

Simple and Flexible Universal Composability: Definition of a Framework and Applications

Von der Fakultät 5 (Informatik, Elektrotechnik und Informationstechnik)
der Universität Stuttgart zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Daniel Rausch

aus Trier

Hauptberichter:

Prof. Dr. Ralf Küsters

Mitberichter:

Prof. Dr. Dominique Schröder

Tag der mündlichen Prüfung: 17.07.2020

Institut für Informationssicherheit (SEC) der Universität Stuttgart

2020

Acknowledgements

I am very grateful to my advisor Ralf Küsters for inviting me to his chair and for providing his excellent support and guidance. He offered continuous feedback that was invaluable in overcoming the challenges that I faced in writing this thesis. I could not have wished for a better advisor.

I also extend my gratitude to all of my colleagues in Trier and Stuttgart. They created a friendly environment not just during work but also for activities outside the office. I further want to thank all of the fellow researchers that have cooperated with me on joint work. I greatly appreciate their input which helped me in creating this thesis.

Furthermore, I would like to thank the *Deutsche Forschungsgemeinschaft (DFG)* for its financial support (Grant KU 1434/9-1).

Last but not least I thank my family for their support and encouragement over the past decades. Without them, I would have never reached the point of writing a PhD thesis.

Contents

Abstract / Kurzzusammenfassung	11
1. Introduction	15
2. The IITM Model in a Nutshell	29
3. The IITM Model With Responsive Environments	39
3.1. The Non-Responsiveness Problem and Its Consequences	39
3.1.1. Underspecified and Ill-Defined Protocols	39
3.1.2. Unintended State Changes and Race Conditions	42
3.1.3. Problems Expressing the Desired Properties	43
3.1.4. The Reentrance Problem	44
3.1.5. Unnatural Specifications of Higher-Level Protocols	48
3.2. Introducing Responsive Environments to the IITM Model	50
3.2.1. Overview	50
3.2.2. Recalling Notions from the IITM Model	51
3.2.3. Restricting Messages	56
3.2.4. Strong Simulatability for Responsive Environments	58
3.2.5. Proving Responsiveness of Simulators	65
3.2.6. Security Notions	66
3.2.7. Parallel Composition of a Constant Number of Protocol Systems	72
3.2.8. Unbounded Self-Composition	75
3.2.9. Combining Security Results for Different Restrictions	98
3.2.10. Discussion	100
3.3. Applying Responsive Environments	103
3.4. Responsive Environments Beyond the IITM Model	105
3.4.1. UC Model	106
3.4.2. GNUC Model	107
3.5. Related Work	109
4. The iUC Framework	111
4.1. Preliminaries	111
4.2. Protocols in the iUC Framework	113
4.2.1. Structure of Protocols	113

4.2.2.	Modeling Corruption	118
4.2.3.	Specifying Protocols	120
4.2.4.	Composing Protocol Specifications	126
4.2.5.	Realization Relation and Composition Theorems	127
4.2.6.	Notation for Specifying Protocols	128
4.3.	A Small Illustrative Example	134
4.4.	Discussion	141
4.4.1.	Core Concepts	141
4.4.2.	Composition Types	143
4.4.3.	Protocols	149
4.4.4.	Capturing the SUC Model	154
4.5.	Related Work	156
5.	Case study	159
5.1.	An Ideal Functionality for Cryptographic Primitives	159
5.1.1.	The ideal functionality $\mathcal{F}_{\text{crypto}}$ from [97]	160
5.1.2.	Extending $\mathcal{F}_{\text{crypto}}$ to Support Diffie-Hellman Key Exchange	168
5.1.3.	Remarks and Discussion of our Extension	173
5.1.4.	Realization $\mathcal{P}_{\text{crypto}}$ of the Ideal Functionality for Cryptographic Primitives	174
5.1.5.	Proving that $\mathcal{P}_{\text{crypto}}$ realizes $\mathcal{F}_{\text{crypto}}$	177
5.2.	Ideal Functionalities for Key Exchange With Key Usability	197
5.3.	Security Analysis of Real World Key Exchange Protocols	203
5.3.1.	ISO Key Exchange Protocol	203
5.3.2.	SIGMA Key Exchange Protocol	211
5.3.3.	OPTLS Key Exchange Protocol	218
5.4.	Discussion	225
5.5.	Related Work	228
6.	Conclusion	229
— Appendix —		230
A.	Dealing With Non-Responsiveness: Queuing of Intermediate Requests with Notifications to the Adversary	231
B.	Single Session Security Analysis in iUC	233
C.	Example: Joint State Realization in iUC	241
D.	Mapping iUC Protocols to the IITM Model	247
D.1.	Notation for the Formal Specification of IITMs	247

D.2. Mapping Templates/Machines	249
D.2.1. System of machines	250
D.2.2. Tapes	250
D.2.3. Message format	251
D.2.4. Check address mode of protocol machines	252
D.2.5. Compute mode of protocol machines	253
D.2.6. Mapping the syntax from Section 4.2.6	259
D.3. Mapping Protocols	261
E. Security Definitions for Cryptographic Primitives	263
E.1. Symmetric Encryption	263
E.2. Public-Key Encryption	264
E.3. Message Authentication Codes (MACs)	265
E.4. Digital Signature Schemes	266
E.5. Decisional Diffie-Hellman	267
E.6. Pseudo-Random Functions	267
F. Formal Specifications of $\mathcal{F}_{\text{crypto}}$ and $\mathcal{P}_{\text{crypto}}$	269
G. Formal Specifications of $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ and $\mathcal{F}_{\text{key-use}}^{\text{UA}}$	281
H. Formal Specifications of our Case Study Key Exchange Protocols	287
Bibliography	313
Academic Curriculum and Publications	315

List of Figures

1.1.	The security experiment in universal composability models.	17
2.1.	The security notion of strong simulatability.	33
3.1.	Example structure of a hybrid protocol.	40
3.2.	The Setup instruction of $\mathcal{F}_{\text{SoK}}(L)$ from [57].	41
3.3.	The Verify instruction of $\mathcal{F}_{\text{D-Cert}}$ from [116].	43
3.4.	The init instruction of $\mathcal{F}_{\text{NIKE}}$ from [72].	43
4.1.	Examples of static and dynamic structures of various protocol types in iUC.	114
4.2.	The static structures of $\mathcal{F}_{\text{sig-CA}}$ and its realization $\mathcal{P}_{\text{sig-CA}}$	116
4.3.	Template for specifying protocols in iUC.	121
4.4.	The ideal signature protocol $\mathcal{F}_{\text{sig-CA}}$ with public key infrastructure.	135
4.5.	The ideal signature protocol $\mathcal{F}_{\text{sig-CA}}$ with public key infrastructure (continued).	136
4.6.	The ideal certificate authority functionality \mathcal{F}_{CA}	136
4.7.	The real signature protocol $\mathcal{P}_{\text{sig-CA}}$ with public key infrastructure.	137
4.8.	The static structures of \mathcal{P}_{KE} using $\mathcal{F}_{\text{sig-CA}}$ respectively $\mathcal{P}_{\text{sig-CA}}$	138
5.1.	The static structure of $\mathcal{F}_{\text{key-use}}^{\text{MA}}$	199
5.2.	The ISO key exchange protocol.	204
5.3.	The static structures of real and ideal world for the ISO protocol.	206
5.4.	The SIGMA protocol with identity protection.	212
5.5.	The 1-RTT non-static mode of OPTLS.	219
5.6.	Key derivation in the 1-RTT non-static mode of OPTLS.	219
B.1.	Example structure of a highest-level protocol with disjoint-sessions.	238
B.2.	Example structure of subroutine protocols with disjoint-sessions.	239
C.1.	An example of combined protocols that are covered by Corollary C.1	242
C.2.	The ideal signature protocol \mathcal{F}_{sig}	244
C.3.	The joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ of \mathcal{F}_{sig} (part I).	245
C.4.	The joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ of \mathcal{F}_{sig} (part II).	246
D.1.	The CheckAddress mode of protocol machines in our framework.	253
D.2.	The Compute mode of protocol machines (part I).	255
D.3.	The Compute mode of protocol machines (part II).	256

D.4. The Compute mode of protocol machines (part III).	257
F.1. The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part I).	270
F.2. The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part II).	271
F.3. The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part III).	272
F.4. The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part IV).	273
F.5. The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part V).	274
F.6. The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part VI).	275
F.7. The ideal functionality for cryptographic primitives $\mathcal{P}_{\text{crypto}}$ (Part I).	276
F.8. The ideal functionality for cryptographic primitives $\mathcal{P}_{\text{crypto}}$ (Part II).	277
F.9. The ideal functionality for cryptographic primitives $\mathcal{P}_{\text{crypto}}$ (Part III).	278
F.10. The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part IV).	279
G.1. The ideal functionality $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ (Part I).	282
G.2. The ideal functionality $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ (Part II).	283
G.3. The ideal functionality $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ (Part I).	284
G.4. The ideal functionality $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ (Part II).	285
G.5. The ideal functionality $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ (Part III).	286
H.1. The ISO protocol for mutually authenticated key exchange \mathcal{P}_{ISO} (Part I).	288
H.2. The ISO protocol for mutually authenticated key exchange \mathcal{P}_{ISO} (Part II).	289
H.3. The ISO protocol for mutually authenticated key exchange \mathcal{P}_{ISO} (Part III).	290
H.4. The ISO protocol for mutually authenticated key exchange \mathcal{P}_{ISO} (Part IV).	291
H.5. The SIGMA protocol for mutually authenticated key exchange $\mathcal{P}_{\text{SIGMA}}$ (Part I).	292
H.6. The SIGMA protocol for mutually authenticated key exchange $\mathcal{P}_{\text{SIGMA}}$ (Part II).	293
H.7. The SIGMA protocol for mutually authenticated key exchange $\mathcal{P}_{\text{SIGMA}}$ (Part III).	294
H.8. The SIGMA protocol for mutually authenticated key exchange $\mathcal{P}_{\text{SIGMA}}$ (Part IV).	295
H.9. The modified OPTLS protocol $\mathcal{P}'_{\text{OPTLS}}$ (Part I).	296
H.10. The modified OPTLS protocol $\mathcal{P}'_{\text{OPTLS}}$ (Part II).	297
H.11. The modified OPTLS protocol $\mathcal{P}'_{\text{OPTLS}}$ (Part III).	298
H.12. The modified OPTLS protocol $\mathcal{P}'_{\text{OPTLS}}$ (Part IV).	299
H.13. The subroutine $\mathcal{F}_{\text{uncorruptDB}}$ of the modified OPTLS protocol $\mathcal{P}'_{\text{OPTLS}}$.	300

Abstract

Security protocols, such as TLS, SSH, IEEE 802.11, and DNSSEC, have become crucial tools in modern society to protect people, data, and infrastructure. They are used throughout virtually all electronic devices to achieve a wide range of different security goals, such as confidentiality, authentication, and integrity. As the long history of attacks on security protocols illustrates, it is indispensable to perform a formal security analysis of such protocols.

A central tool in cryptography for taming the complexity of the design and the analysis of modern protocols is modularity, provided by security models for universal composability. Such models allow for designing and analyzing small parts of a protocol in isolation and then reusing these security results in the context of the overall protocol. This is not just easier than analyzing the whole protocol as a monolithic block but also reduces the overall effort required in building and analyzing multiple different protocols based on the same underlying components, such as cryptographic primitives.

Ideally, a model for universal composability should support a protocol designer in easily creating full, precise, and detailed specifications as well as sound security proofs of various protocols for various types of adversarial models, instead of being an additional obstacle one has to overcome during a security analysis. In particular, such a model should be *sound*, *flexible/expressive*, and *easy to use*. Unfortunately, despite the wide spread use of models for universal composability, existing models and frameworks are still unsatisfying in these respects as none combines all of these requirements simultaneously.

In this thesis we therefore develop a model for universal composability, called the *iUC framework*, which combines *soundness*, *usability*, and *flexibility* in a so far unmatched way, and hence constitutes a solid framework for designing and analyzing essentially any protocol and application in a modular, universally composable, and sound manner.

We use our model in a case study to analyze multiple different key exchange protocols precisely as they are deployed in practice. This illustrates the combination of both flexibility and usability of our model. This case study is also an important independent contribution as this is the first faithful security analysis of these unmodified protocols in a universal composability setting.

Kurzzusammenfassung

Sicherheitsprotokolle, wie TLS, SSH, IEEE 802.11 und DNSSEC, sind ein wichtiges Werkzeug der modernen Gesellschaft um Personen, Daten und Infrastruktur zu schützen. Sie werden in nahezu allen elektronischen Geräten genutzt um eine große Bandbreite von verschiedenen Sicherheitszielen zu gewährleisten, wie etwa Geheimhaltung, Authentifizierung, und Integrität. Die lange Geschichte von Angriffen auf Sicherheitsprotokolle zeigt dabei, dass es unerlässlich ist, eine formale Sicherheitsanalyse solcher Protokolle durchzuführen.

Ein zentrales Werkzeug um die Komplexität des Designs und der Analyse von modernen Protokollen zu bewältigen ist Modularität, welche von Modellen für Universal Composability geboten wird. Diese Modelle ermöglichen es, kleine Teile eines Protokolls in Isolation zu designen und zu analysieren, und diese Resultate dann im Kontext des ganzen Protokolls wiederzuverwenden. Dies macht die Analyse nicht nur einfacher als eine direkte Analyse des ganzen Protokolls. Zusätzlich wird es dadurch auch leichter, mehrere verschiedene Protokolle basierend auf den gleichen Komponenten, wie etwa kryptographische Primitive, zu bauen und zu analysieren.

Im Idealfall sollte ein Modell für Universal Composability einen Protokolldesigner dabei unterstützen, möglichst einfach vollständige, präzise, und detaillierte Spezifikationen sowie Sicherheitsbeweise von verschiedensten Protokollen in diversen Angreifermodellen zu erstellen; auf keinen Fall sollte ein Modell ein weiteres Hindernis darstellen, das es in der Sicherheitsanalyse zu bewältigen gilt. Daher sollte solch ein Modell insbesondere *formal korrekt*, *flexibel/ausdrucksstark*, sowie gleichzeitig *leicht zu benutzen* sein. Trotz der weiten Verbreitung von Modellen für Universal Composability in der Literatur, existiert leider noch kein Modell das diese Eigenschaften zufriedenstellend kombiniert.

In dieser Doktorarbeit entwickeln wir daher ein Modell für Universal Composability, genannt das *iUC framework*, welches *formale Korrektheit*, *Flexibilität* und *einfache Benutzung* auf bisher nicht erreichte Weise miteinander kombiniert. Dieses Modell stellt damit eine solide Grundlage für das modulare Design sowie die modulare Analyse von beliebigen Protokollen und Anwendungen dar.

In einer Fallstudie benutzen wir unser Modell um mehrere Schlüsselaustauschprotokolle exakt wie sie in der Praxis eingesetzt werden zu analysieren. Damit illustrieren wir sowohl Flexibilität als auch die einfache Benutzung unseres Modells. Unsere Fallestudie bietet aber auch einen unabhängigen Beitrag: Dies ist die erste modulare Analyse dieser unmodifizierten Protokolle in einem Modell für Universal Composability.

1. Introduction

Security protocols have become crucial tools in modern society to protect people, data, and infrastructure. Prominent examples of such protocols include TLS, SSH, IPSec, Kerberos, IEEE 802.11, 3GPP 5G, Bluetooth, DNSSEC, DANE, ACME, OAuth 2.0, OpenID Connect, Universal 2nd Factor, EMV, and Signal. Security protocols are used throughout virtually all electronic devices to achieve a wide range of different security goals, such as confidentiality, authentication, integrity, availability, privacy, deniability, non-repudiation, verifiability, and accountability. However, the long history of attacks on security protocols found in the literature (see, e.g., [4, 5, 7, 8, 17, 22, 54, 65, 68–71, 106, 113]) has shown that building secure protocols is very hard: while designing a protocol one can easily miss a seemingly minor detail or some unexpected special case that in turn undermines the security of the protocol in its entirety. It is therefore indispensable to perform a formal security analysis of protocols. This not only allows for systematically finding attacks but also establishes trust in a protocol by proving, in a strict mathematical sense, the absence of attacks within the considered model, thereby excluding whole classes of attack vectors.

There are several different approaches for formally analyzing and proving security of protocols, with the main approaches being symbolic, game-based, implementation-based, and universal composability. These approaches are often augmented with tools for computer-aided (partially) automated analysis, which can increase trust into results by making them machine verifiable and potentially also reduce the effort required for performing a formal security analysis. All of these approaches have different advantages and shortcomings that complement each other. There is no silver bullet, as can also be seen by the fact that important protocols, such as TLS, have been studied in the literature using all of these approaches, taking different views and making use of the specific merits thereof (see, e.g., [12, 17–19, 60, 66, 79, 81, 84]). More specifically:

- Symbolic (Dolev-Yao-style) approaches abstract from low level cryptographic details in order to offer a very high degree of automation (see, e.g., [20, 61, 105] for automated tools).
- Game-based security models are very expressive and flexible in defining individual security properties of a protocol (see, e.g., [15, 47, 62] and [14, 21] for tools). While game-based models do not enjoy built-in modularity for proofs and security results, efforts have been made to improve the modularity provided by these models (see, e.g., [24–26]).
- Implementation-based analysis captures details of the actual implementations of protocols, which is very desirable, but also makes the analysis much more involved. These approaches are typically combined with tools that allow for handling the additional level of detail more

easily (see, e.g., [16, 92, 93, 109, 110]).

- Universal composability approaches come with built-in modularity and allow one to show that protocols are secure in arbitrary (polynomially bounded) environments, and hence, they provide very strong security guarantees that hold true within arbitrary contexts (see, e.g., [36, 75, 87, 102] and [45, 53] for tools). However, these very strong security requirements also impose some limitations on the settings that can be considered in security proofs; for example, the so-called commitment problem might sometimes prevent considering full dynamic corruption of parties (cf. Sections 5.2 and 5.3 for more details)

In this thesis, we focus on the universal composability approach due to its strong security guarantees and in particular its modularity. Modular security analysis is a crucial tool for taming the complexity of modern protocols as it allows for analyzing a small part of a protocol in isolation and then reusing this security result in the context of the overall protocol. This is not just easier than analyzing the whole protocol as a monolithic block but the reduced complexity also lowers the potential for errors in the security proof. Furthermore, once the security of one component has been shown, the same result can be reused within arbitrarily many contexts, thus drastically reducing the overall effort required in building and analyzing multiple different protocols based on the same underlying components, such as cryptographic primitives.

Universal Composability

Universal composability is a concept that has found wide spread use for the modular design and analysis of not just cryptographic primitives and protocols (see, e.g., [3, 6, 27, 33, 34, 58, 59, 64, 80, 115]) but also in other areas and for other types of security protocols, such as for modeling and analyzing OpenStack [77], network time protocols [46], OAuth v2.0 [56], the integrity of file systems [38], as well as privacy in email ecosystems [55].

The concept of universal composability was developed independently by Canetti [36] and Pfitzmann and Waidner [107] in 2001; both of these works were in turn inspired by and have evolved from the concept of simulation-based security as introduced by Goldreich, Micali, and Wigderson in their work on secure multiparty computation [73]. The general idea of universal composability is that one first defines an *ideal protocol* (also called *ideal functionality*) \mathcal{F} that specifies the intended behavior of a target protocol/system and abstracts away implementation details. For a concrete realization (often called real protocol) \mathcal{P} , one then proves that “ \mathcal{P} behaves just like \mathcal{F} ” in arbitrary contexts/environments. More specifically, one shows that for every malicious network attacker \mathcal{A} attacking \mathcal{P} , there exists a benign network attacker \mathcal{S} , called simulator, attacking \mathcal{F} such that no environment \mathcal{E} (playing the role of arbitrary higher-level protocols and communicating with the network attacker) can distinguish whether it is running with \mathcal{A} and \mathcal{P} or \mathcal{S} and \mathcal{F} (cf. Figure 1.1). Intuitively, this implies that every advantage gained by attacking \mathcal{P} can be also be gained by attacking \mathcal{F} instead, where the latter is secure by definition. Hence, the protocol \mathcal{P} enjoys at least the security properties specified by the ideal protocol \mathcal{F} .

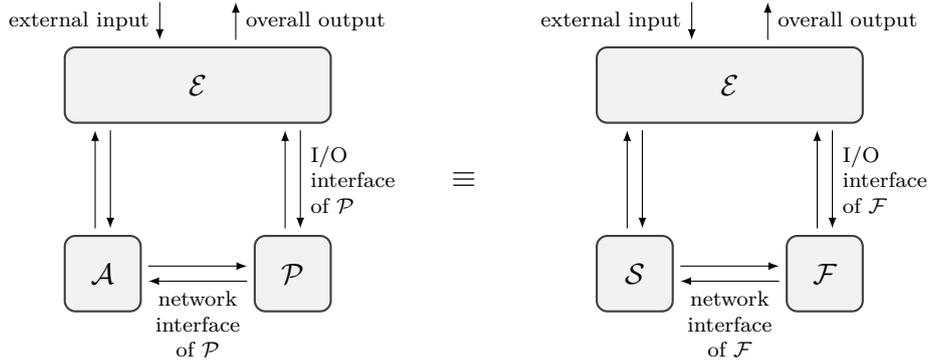


Figure 1.1.: Setup of the security experiment in universal composability models. The I/O interface of \mathcal{P}/\mathcal{F} serves as a direct connection to higher-level protocols that want to use \mathcal{P}/\mathcal{F} as a subroutine, whereas the network interface is used for sending messages over an insecure network that is controlled by a network adversary. In the security experiment, the environment simulates arbitrary higher-level protocols of \mathcal{P}/\mathcal{F} while communicating with the network attacker.

At the core of every model for universal composability lies a so-called composition theorem that enables the modular design and analysis of protocols. Such a composition theorem, intuitively, allows for replacing a subroutine \mathcal{F} used by some higher-level protocol \mathcal{Q} with a realization \mathcal{P} of \mathcal{F} while retaining all security and functional properties of the overall protocol. Hence, a protocol designer can analyze \mathcal{Q} based on an ideal functionality \mathcal{F} and show that the combined protocol $\mathcal{Q}|\mathcal{F}$ is secure, i.e., realizes some other ideal functionality \mathcal{F}' . Analyzing \mathcal{Q} based on \mathcal{F} is usually much easier than analyzing \mathcal{Q} based directly on \mathcal{P} : ideal protocols provide absolute security guarantees, i.e., security cannot be broken by definition. Furthermore, specifications of ideal protocols typically are smaller and more concise, making them easier to handle. Once one has shown that $\mathcal{Q}|\mathcal{F}$ realizes \mathcal{F}' , then the composition theorem directly implies that \mathcal{Q} running with an actual implementation \mathcal{P} of \mathcal{F} , i.e. $\mathcal{Q}|\mathcal{P}$, also realizes \mathcal{F}' . This process can be further iterated by building new protocols on top of \mathcal{F}' ; again, the analysis of such protocols based on a smaller and perfectly secure functionality \mathcal{F}' is much easier than using the whole realization $\mathcal{Q}|\mathcal{P}$ directly.

Many different models for universal composability have been proposed in the literature [9, 11, 35, 36, 39–41, 51, 75, 87, 99, 102, 103, 107, 114]. Today the most relevant models for universal composability are the UC model by Canetti [36] (see [35] for the latest version), the IITM model by Küsters [87] (see [99] for a full and revised version), the GNUC model by Hofheinz and Shoup [75], as well as the CC model by Maurer [102]. The first three models are closely related as they consider the same basic setting: protocols are modeled via interactive Turing machines which satisfy a certain polynomial runtime notion, which can be copied/instantiated arbitrarily often during a run of a protocol, and which include some addressing mechanism for sending and receiving messages to different copies/instances of the same machine. Furthermore, the class of environments considered by those models is fixed to be the set of all machines that satisfy a polynomial runtime notion. In what follows, we say “UC-like models” to refer to

the class of universal composability models that are built in this fashion, i.e., the UC, GNUC, IITM and similar closely related models. This class of models is used by protocol designers to model and analyze protocols in the vast majority of the universal composability literature. In contrast to UC-like models, the CC model takes a much more abstract point of view: it does not fix a machine model, an instantiation and addressing mechanism for creating and sending messages to copies of the same machine, a runtime notion, nor a class of environments that is considered for security proofs. Due to this very abstract viewpoint that drastically differs from UC-like models, it is currently still an open research question whether and how typical protocol specifications, proofs, and arguments from the universal composability literature can be performed in the CC model. This question is orthogonal to this thesis; we therefore focus on UC-like models in the following.

Ideally, a model for universal composability should support a protocol designer in easily creating full, precise, and detailed specifications as well as sound security proofs of various protocols for various types of adversarial models, instead of being an additional obstacle one has to overcome during a security analysis. In particular, such a model should satisfy at least the following requirements:

Soundness: The model itself, including all of its theorems such as the composition theorem, must be sound. This is a necessary requirement for obtaining any form of security statement: if the underlying model is not sound, e.g., because the composition theorem is flawed, then security analyses performed in that model do not carry any meaning. In the worst case, the security analysis might miss serious attacks on the analyzed protocol.

Flexibility: The model must be sufficiently flexible and expressive to allow for the precise design and modular analysis of a wide range of protocols and applications as well as security models, e.g., in terms of corruption, setup assumptions, and globally available information. This property is crucial to make the model widely applicable, therefore allowing more protocol designers to benefit from the modularity and strong security guarantees of the universal composability approach.

Usability: It should be easy to perform a sound security analysis and security proof within the model. This includes not just the proof itself, for which there should be only few technical requirements to show and few edge cases to take care of. But it should also be easy to precisely and fully formalize protocols, which is an important prerequisite for carrying out formally/mathematically correct proofs. For this purpose, there should exist (easy to use) modeling conventions that allow a protocol designer to focus on the core logic of protocols instead of having to deal with technical details of the underlying model or repeatedly taking care of recurrent issues, such as modeling parties, sessions, and standard corruption behavior. Altogether, good usability not just simplifies security analyses, thereby reducing the potential for errors in proofs, but also makes the universal composability approach more accessible to protocol designers.

Unfortunately, despite the wide spread use of the universal composability approach, existing

models and frameworks are still unsatisfying in these respects as none combines all of these requirements simultaneously (we discuss this in detail in Section 4.5).

Goal and Contributions of This Thesis

The main goal of this thesis is therefore to provide the first universal composability framework that is *sound*, *flexible*, and *easy to use*, and hence constitutes a solid framework for designing and analyzing essentially any protocol and application in a modular, universally composable, and sound way. Developing such a security framework is a difficult and very delicate task that takes multiple years if not decades as the history on models for universal composability shows.

We use the IITM model as a starting point for achieving this goal. This UC-like model already is *sound* and *very flexible/expressive*. That is, protocols in the IITM model are defined in a very general way, i.e., they are essentially just arbitrary sets of interactive Turing machines, which may be connected in some way. In addition, the model offers a general addressing mechanism for machine instances. This gives great flexibility as arbitrary protocols can be specified; all theorems, such as composition theorems, are proven for this very general class of protocols. Furthermore, the IITM model offers a very general and at the same time simple runtime notion so that protocol designers do not have to care much about runtime issues and do not have to resort to, e.g., artificial padding of messages, making sound *proofs easier to carry out*. Unfortunately, this generality hampers *usability*, in particular when specifying a protocol. The model does not directly provide design conventions and modeling tools, for example, to deal with party IDs, sessions, subroutine relationships, shared state, or (different forms of) corruption; all of this is left to the protocol designer to manually specify for every design and analysis task, distracting from modeling the actual core logic of a protocol and raising the bar of entry for protocol designers coming from other models and approaches.

Based on the IITM model, we proceed in two major steps. First, we identify the so-called *non-responsiveness problem*, which negatively affects both usability and flexibility/expressiveness of *all* universal composability models. As further explained below, this problem causes more involved protocol specifications and proofs, even leading to flawed security proofs in the literature, ideal functionalities that might not be reusable as subroutine for other protocols, and certain properties of protocols that cannot be expressed. We solve this problem in its entirety by extending and generalizing the IITM model to support our concept of *responsive environments*. In a second step, we propose the *iUC framework*, which is an instantiation of the extended IITM model. The iUC framework addresses the lack of usability of the IITM model by adding the missing set of modeling conventions and tools, while at the same time preserving both soundness and flexibility. Hence, the iUC framework achieves our goal of combining soundness, flexibility, and usability within a single universal composability model.

We then illustrate the iUC framework via a case study in a third step. This case study is also of independent interest as it includes several ideal functionalities that can be used to further simplify security proofs, thereby improving accessibility and ease of use of the universal

composability approach. In the following, we explain each of the three steps in more detail.

Part I: Solving the Non-Responsiveness Problem

In the specifications of protocols, it is often required for the adversary on the network (and in turn the environment connected to the network adversary) to provide some meta-information via the network interface to the protocol, such as cryptographic algorithms, cryptographic values of signatures, ciphertexts, and keys, or corruption-related messages (cf. Figure 1.1 for an illustration of interfaces and connections between protocols, attackers, and environments). Conversely, protocols often have to provide the adversary with meta-information, for example, signaling information (e.g., the existence of machines) or again corruption-related messages (e.g., information leaks). Such messages for transmitting meta-information do not correspond to any real network messages but are merely used for modeling purposes. Giving the adversary/environment the option to not respond immediately to such modeling-specific messages typically does not translate to any real attack scenario. Hence, often it is natural for protocol designers to expect that the adversary/environment answers and returns control back to the protocol immediately whenever the protocol requests meta-information from or provides meta-information to the adversary via a message on the network interface. In the following, we call such messages *urgent messages* or *urgent requests*.

Urgent requests occur in many protocols (including both real and ideal protocols) from the literature, see, e.g., [2, 10, 28, 36, 44, 50, 52, 57, 63, 67, 72, 74, 86, 100, 101, 111, 116] as well as *all* protocols based on the UC model in the version from 2013 [35] (cf. Footnote 4 in Section 3.1.4). This is not surprising as the exchange of meta-information between the adversary/environment and the protocol is an important mechanism for protocol designs in any UC-like model. For example, this mechanism allows for specifying the behaviour of cryptographic values or algorithms by an ideal functionality in a natural manner without having to worry about how these values are generated or the parameters for the algorithms are set up. To be more concrete, consider an ideal digital signature functionality \mathcal{F}_{sig} . This functionality is typically defined in such a way that it requests the signing and verification algorithms as well as key pairs from the attacker. During signature verification, \mathcal{F}_{sig} manually checks whether a signature has been forged (i.e., verification succeeds but the message was never signed before). In particular, security properties provided by \mathcal{F}_{sig} are independent of the specific signing and verification algorithms or key distributions, allowing for a wide range of different realizations with varying algorithms and key distributions. To give another example for use cases of urgent requests, by sending some meta-information from the protocol to the adversary one can easily and naturally model information leaks (e.g. in the case of honest-but-curious corruption or for an ideal encryption functionality) or signal an event such as the creation of a new machine instance (as might be required by the underlying universal composability model). In general, it seems impossible to dispense with urgent requests altogether, and certainly such requests are very convenient and widely used in the literature (see Sections 3.1.1 to 3.1.5 for examples).

In all existing universal composability models, it currently is not guaranteed that urgent requests are answered immediately by the adversary: when receiving an urgent request on the network interface, adversaries and environments can freely activate protocols in between, on network and I/O interfaces, without or before answering the urgent request. In what follows, we refer to this problem as the *problem of non-responsive adversaries/environments* or the *non-responsiveness problem*.

This problem formally does not invalidate any of the universal composability models. It, however, often makes both the specification of protocols and proving the realization relation much harder, and it makes the models less expressive (see below), therefore limiting both usability and flexibility of those models. Most disturbingly, the non-responsiveness problem is really an artificial problem: urgent requests do not correspond to any real messages, and the adversary not responding promptly to such requests does not reflect any real attack scenario. Hence, non-responsiveness forces protocol designers to take care of artificial adversarial behavior that was unintended in the first place and is merely a modeling artifact.

More specifically, protocol designers currently have to deal with a number of delicate problems: (i) While waiting for a response to an urgent request, a protocol might receive other requests, and hence, protocol designers have to take care of interleaving and dangling requests. (ii) While a protocol is waiting for an answer from the adversary to an urgent request, other parties and other parts of the protocol can be activated in the meantime, which might change their state, even their corruption status, which in turn might lead to race conditions. (iii) The adversary might provide an unexpected response to an urgent request, or he might not provide a response at all, thus interrupting the intended execution and potentially blocking parts of the protocol.

This, as further discussed in Section 3.1, makes it difficult to deal with the non-responsiveness problem and results in unnecessarily complex and artificial specifications of protocols, which, in addition, are then hard to reuse. In some cases, one cannot even express certain desired properties. As explained in Section 3.1.4, there is no generic and generally applicable way to deal with the non-responsiveness problem, and hence, one has to resort to solutions specifically tailored to the protocols at hand, if at all possible.

Importantly, the non-responsiveness problem propagates to higher-level protocols as they might not get responses from their subprotocols as expected. The security proofs also become more complex because one, again, has to deal with runs having various dangling and interleaving requests as well as unexpected and unintuitive state changes which do not translate into anything in the real world but are just an artifact of the modeling.

We emphasize that, clearly, in the context of actual network messages one has to deal with many of the above problems in the specifications of protocols. But, in contrast to the non-responsiveness problem, dealing with the asynchronous nature of networks has a real counterpart, and these two types of interactions with the adversary should not be confused.

In the literature, urgent requests and the non-responsiveness problem occur in many protocols. Nevertheless, protocol designers frequently ignore this problem (see, e.g., [2, 10, 28, 52, 57,

74, 86, 100, 101, 111, 116]), i.e., they seem to implicitly assume that urgent request *are* answered immediately, probably, at least as far as ideal functionalities are concerned, because their simulators typically promptly respond to these kinds of requests. Due to this assumption, protocol specifications often do not specify their behavior or might expose unexpected behavior when an urgent request is not answered immediately. This in turn leads to (ideal) protocols that cannot be reused as subroutines since their exact behavior when running with a malicious network attacker is underspecified. This assumption also often leads to flawed security proofs and false security statements as protocol designers did not consider cases where the adversary does not respond immediately to urgent requests. When those cases are properly taken into account, it might even turn out that the security statement cannot be shown at all due to unexpected protocol behavior in such edge cases (see Section 3.1 for more details, including examples from the literature).

Our contributions: In Chapter 3, we propose a generalized IITM model with the novel concept of *responsive environments* and *responsive adversaries*. This model completely avoids and, by this, solves the non-responsiveness problem as it guarantees that urgent requests *are* answered immediately. This really is the most obvious and most natural solution to the problem: there is no reason that protocol designers should have to take care of the non-responsiveness problem and its many negative consequences.

More specifically, the main idea behind our model is as follows. When a protocol sends what we call a *restricting* message to the adversary/environment via the network interface, then the adversary/environment is forced to be responsive, i.e., he is forced to reply with a valid response before sending any other message to the protocol. This requires careful definitions and non-trivial proofs to ensure that all properties and features that are expected from a universal composability model are lifted to the setting with responsive environments and adversaries. While we provide full definitions and formal proofs for the IITM model, we note that our concepts of restricting messages and responsive environments and adversaries are in fact a model independent contribution that should be applicable to all existing UC-like models. We exemplify this for the UC and GNUC models in Section 3.4.

By using our IITM model with responsive environments, protocols can be modeled in a very natural way: protocol designers can simply declare urgent requests to be restricting messages, which hence have to be answered immediately. This elegantly and completely solves the non-responsiveness problem. In particular, protocol designers no longer have to worry about this problem, and specifications of protocols and ideal functionalities are greatly simplified as one can dispense with artificial solutions. In fact, as illustrated in Section 3.3, with our concepts we can easily fix existing specifications from the literature in which the non-responsiveness problem has not been dealt with properly or has simply been ignored as protocol designers often implicitly assumed responsiveness for urgent messages. In some cases, we can now even express certain (properties of) protocols in a natural and elegant way that could not have been expressed before (see Sections 3.1.3 and 3.3). Of course, with simplified and more natural

protocols, also security proofs become easier because the protocol designer does not have to consider irrelevant and unrealistic adversarial behavior and execution orders.

Part II: Unifying Soundness, Flexibility, and Usability

As detailed above, the main goal of this thesis is to provide a universal composability framework that is *sound*, *flexible*, and *easy to use*, and hence constitutes a solid framework for designing and analyzing essentially any protocol and application in a modular, universally composable, and sound way. Such a model supports a protocol designer in easily creating full, precise, and detailed specifications as well as sound security proofs of various applications for various types of adversarial models, instead of being an additional obstacle one has to overcome during a security analysis. Solving the non-responsiveness problem was a necessary prerequisite for achieving this goal as this problem limits both flexibility and usability of universal composability models in general. Building on our solution to the non-responsiveness problem from Part I, we are now ready to unify soundness, flexibility, and usability within a single model for universal composability.

Our contributions: In Chapter 4, we propose a new universal composability framework called iUC (“IITM based Universal Composability”). This framework builds on top of our IITM model with responsive environments proposed in Part I of this thesis. As explained above, this model already meets our goals of *soundness* and *flexibility*, but lacks *usability* due to missing modeling tools and conventions, for example, to deal with party IDs, sessions, subroutine relationships, shared state, or (different forms of) corruption.

In essence, iUC is an instantiation of the IITM model that provides a convenient and powerful framework for specifying protocols. In particular, iUC greatly improves upon *usability* of the IITM model by adding missing conventions and tools for many of the above mentioned repetitive aspects of modeling a protocol, while also abstracting from some of the (few) technical aspects of the underlying model; see Section 4.5 for the comparison of iUC with other frameworks.

At the core of iUC is *one* convenient template that supports protocol designers in specifying arbitrary types of protocols in a precise, intuitive, and compact way. This is made possible by new concepts, including the concept of entities as well as public and private roles. The template comes with a clear and intuitive syntax which further facilitates specifications and allows others to quickly pick up protocol specifications and use them as subroutines in their higher-level protocols.

A key difficulty in designing iUC is to preserve the *flexibility* of the IITM model in expressing (and composing) arbitrary protocols while still improving *usability* by fixing modeling conventions for many repetitive aspects. We solve this tension between flexibility and usability by, on the one hand, allowing for a high degree of customization and, on the other hand, by providing sensible defaults for repetitive and standard specifications. Indeed, as further explained and discussed in Section 4.4 and also illustrated by several examples (cf. Sections 4.3 and 5 as well

as Appendix C), iUC preserves flexibility and supports a wide range of protocol types, protocol features, and composition operations, such as: ideal as well as globally available functionalities with arbitrary protocol structures, i.e., rather than being just monolithic machines, functionalities may, for example, contain subroutines; protocols with joint state and/or global state; shared state between multiple protocol sessions (without resorting to joint state realizations); subroutines that are partially globally available while other parts are only locally available; realizing global functionalities with other protocols (including joint state realizations that combine multiple global functionalities); different types of addressing mechanisms via globally unique and/or locally chosen session IDs; global functionalities that can be changed to be local when used as a subroutine; many different highly customizable corruption types (including incorruptability, static corruption, dynamic corruption, corruption only under certain conditions, automatic corruption upon subroutine corruption); a corruption model that is fully compatible with joint state realizations; arbitrary protocol structures that are not necessarily hierarchical trees and which allow for, e.g., multiple highest-level protocols that are accessible to the environment.

Importantly, all of the above is supported by just *a single template* and *two* composition theorems (one for parallel composition of multiple protocols and one for unbounded self-composition of independent instances of the same protocol). This makes iUC quite user friendly as protocol designers can leverage the full flexibility with just the basic framework; there are no extensions or special cases required to support a wide range of protocol types, unlike in many other models.

We emphasize that we do not claim specifications done in iUC to be shorter than the informal descriptions commonly found in the universal composability literature. A full, non-ambiguous specification cannot compete with such informal descriptions in terms of brevity, as these descriptions are often underspecified and ignore details, including model specific details and the precise corruption behavior. iUC is rather meant as a *powerful and sound tool for protocol designers that desire to specify protocols fully, without sweeping or having to sweep anything under the rug, and at the same time without being overburdened with modeling details and technical artifacts*. Such specifications are crucial for being able to understand, reuse, and compose results and to carry out sound proofs.

Part III: Case Study

To illustrate usability and flexibility of our iUC framework proposed in Part II, we perform a security analysis of several key exchange protocols from practice, namely one of the ISO 9798-3 key exchange protocol family [78], the SIGMA protocol with identity protection [82], and the non-static mode of the OPTLS protocol [85]. Importantly, due to the high level of flexibility of the iUC framework, which is inherited and preserved from the underlying IITM model, we are able to model all protocols precisely as they are deployed in reality. This degree of fidelity is not supported by other UC-like models. In particular, previous analyses of the ISO and SIGMA protocols in such models had to consider modified versions of the protocols with potentially different security properties due to a lack of flexibility of the underlying models and

their respective conventions (see also related work in Section 5.5).

As part of our case study, we also identify and address a further obstacle that designers of cryptographic protocols have to face: for every protocol one has to carry out reduction proofs from the security notions of the protocols to the cryptographic primitives employed time and again. This stands in stark contrast to the idea of modular security analysis: even in universal composability models protocol designers typically have to carry out (tedious, repetitive, and error-prone) reduction proofs.

One major goal of our case study, besides illustrating the iUC framework, is therefore to leverage the modularity of universal composability to get rid of reduction proofs as much as possible or ideally even altogether. This should lead to proofs that are shorter, without being imprecise, as well as easier to understand and carry out. The main idea to achieve this goal, which builds on and extends work by Küsters and Tuengerthal [96,97], is as follows. We provide an ideal functionality $\mathcal{F}_{\text{crypto}}$ which covers various cryptographic primitives, including standard Diffie-Hellman (DH) key exchanges based on the DDH assumption, symmetric/asymmetric encryption, key derivation, MACing, and signing. All primitives provided by $\mathcal{F}_{\text{crypto}}$ can be used with each other in an idealized way. For example, a protocol \mathcal{P} using $\mathcal{F}_{\text{crypto}}$ as a subroutine can first exchange a key via an ideal Diffie-Hellman key exchange where some messages are (ideally) signed and then derive a MAC and a symmetric encryption key from the DH key. Importantly, both keys can still be used in an idealized way, i.e., one can perform ideal MACing and encryption using these keys. Therefore, when \mathcal{P} is analyzed based on $\mathcal{F}_{\text{crypto}}$, one does not have to perform any reductions, hybrid arguments, or in fact use any probabilistic reasoning (at least not for the primitives supported by $\mathcal{F}_{\text{crypto}}$); purely information-theoretic arguments are sufficient.

We show that $\mathcal{F}_{\text{crypto}}$ can be realized by standard cryptographic assumptions. While the proof involves several reductions and hybrid arguments, this step is a once and for all effort thanks to the modularity provided by universal composability models. In particular, if we have analyzed some protocol \mathcal{Q} based on $\mathcal{F}_{\text{crypto}}$, then by the composition theorem $\mathcal{F}_{\text{crypto}}$ can be replaced by its realization $\mathcal{P}_{\text{crypto}}$ so that the ideal cryptographic primitives are replaced by their real counterparts. This step does not involve any reductions but is directly implied by the composition theorem. This is unlike essentially all other approaches for protocol analysis, where reductions to the cryptographic assumptions of primitives have to be carried out time and again for every protocol \mathcal{Q} one wants to analyze.

In addition to $\mathcal{F}_{\text{crypto}}$, we also provide new functionalities for ideal key exchange that allow a higher-level protocol to still use a session key in an idealized way. Altogether, when using our functionalities as subroutines, the need for reduction proofs is greatly reduced or such proofs are avoided completely in many cases. Protocol designers can argue on an intuitive information-theoretic level while being able to analyze a protocol in a very modular way with universally composable security guarantees.

Our contributions: More precisely, our case study in Chapter 5 not only illustrates the iUC framework but also makes several additional contributions.

- We extend the ideal functionality $\mathcal{F}_{\text{crypto}}$ from [97] to also support standard DH key exchange with two key shares g^a and g^b . This is a crucial step as many real world protocols use Diffie-Hellman key exchanges and thus could not have been analyzed before using $\mathcal{F}_{\text{crypto}}$. Designing such an extension requires care in order for the extension to, on the one hand, provide all expected properties to higher-level protocols and, on the other hand, still be realizable under standard cryptographic assumptions.
- Our functionality $\mathcal{F}_{\text{crypto}}$ ensures that the adversary on the network cannot interfere with higher-level protocols while they use $\mathcal{F}_{\text{crypto}}$ to perform local computations. While this is expected and natural for such an ideal functionality, it previously was impossible to model this property. Leveraging the concept of responsive environments introduced in Part I of this thesis, we can now indeed provide this property for $\mathcal{F}_{\text{crypto}}$, which further simplifies security proofs.
- We propose and prove a realization $\mathcal{P}_{\text{crypto}}$ for $\mathcal{F}_{\text{crypto}}$ based on standard cryptographic assumptions. The proof is quite involved, with several hybrid arguments, as $\mathcal{F}_{\text{crypto}}$ allows for a wide range of (combinations of) operations. But, as explained above, due to the modularity of iUC this is a once and for all effort.
- Inspired by an ideal functionality from [96], we propose two new functionalities for both mutually and unilaterally authenticated key exchange with perfect forward secrecy. Unlike most other key exchange functionalities, which output the session key, our functionalities allow higher-level protocols to still use the exchanged key in an ideal way, namely for idealized key derivation, symmetric encryption, and MACing. Hence, as mentioned, one can avoid reduction proofs also for the higher-level protocols, such as secure channel protocols. Further discussion and comparison with other key exchange functionalities is provided in Section 5.2.
- We illustrate the usefulness of our proposed functionalities by showing for three different real world key exchange protocols that they realize our key exchange functionalities with mutual or unilateral authentication. Due to the use of $\mathcal{F}_{\text{crypto}}$, none of the security proofs require any reductions, hybrid arguments, or even probabilistic reasoning:
 - We provide the first security proofs of unaltered versions of the ISO 9798-3 [78] and the SIGMA [82] key exchange protocols in an universal composability model. This is made possible due to the great flexibility offered by the iUC framework, which is inherited and preserved from the underlying IITM model (see also Sections 5.4 and 5.5).
 - We analyze the non-static mode of the OPTLS key exchange protocol [85] and find a subtle bug in the original game-based reduction proof. We show that, under the original security assumptions, a slight variation of the protocol is a secure unilaterally authenticated and universally composable key exchange protocol.

Publications

This thesis is based on three works that were published at leading international security and cryptography conferences [29, 31, 89]. Each of these publications roughly corresponds to one of the three parts explained above. We note that the authors of these publications are listed in alphabetical order. The main technical contributions of each work are due to the author of this thesis.

In addition to the above three works the author of this thesis has also collaborated on [88, 91, 98, 99] while he was a PhD student. A full list of all publications of the author is included at the end of this thesis.

Structure of This Thesis

We first briefly recall the original IITM model in Chapter 2. We then illustrate the non-responsiveness problem in more detail and introduce our generalized IITM model with responsive environments in Chapter 3. Building on this, we propose the iUC framework in Chapter 4 with our case study given in Chapter 5. We conclude in Chapter 6. Related work is discussed within each of the Chapters 3 to 5. Further details are available in the Appendix.

2. The IITM Model in a Nutshell

We now briefly recap the IITM model as defined in [99]. This overview is mostly taken from [99] with some small modifications, such as an additional overview of common terminology for various types of protocols from the universal composability literature. This chapter remains on an informal level that is sufficient to understand the rest of this thesis; more technical details are available in Section 3.2.2 where we propose a generalized version of the IITM model. We start by explaining the computational model and then recall universal composability security notions and composition theorems.

The General Computational Model. The general computational model is defined in terms of systems of IITMs. An *inexhaustible interactive Turing machine (IITM)* is a probabilistic Turing machine with named input and output tapes as well as an associated polynomial. The tape names determine how different machines are connected in a system of IITMs (see below). An IITM runs in one of two modes, **CheckAddress** and **Compute**. The **CheckAddress** mode is used as a generic mechanism for addressing different instances of IITMs in a system of IITMs, as explained below. In this mode, an IITM may perform, in every activation, a deterministic polynomial time computation in the length of the security parameter plus the length of the current input plus the length of its current configuration, where the polynomial is the one associated with the IITM. The IITM outputs “accept” or “reject” at the end of the computation in this mode, indicating whether the received message is processed further or ignored. The actual processing of the message, if accepted, is done in mode **Compute**. In mode **Compute**, a machine may output only at most one message on an output tape (and hence, only at most one other machine is triggered). The runtime in this mode is not a priori bounded. Later the runtime of systems and their subsystems will be defined in such a way that the overall runtime of a system of IITMs is polynomially bounded in the security parameter plus the length of the external input. We note that in both modes, an IITM cannot be exhausted (hence, the name): in every activation it can perform actions, such as reading the full input, and cannot be forced to stop. This property, while not satisfied in all other models, is crucial to obtain a reasonable model for universal composability (see [99] for a detailed discussion).

A *system* \mathcal{S} of IITMs is of the form $\mathcal{S} = M_1 | \dots | M_k | !M'_1 | \dots | !M'_{k'}$, where M_i , $i \in \{1, \dots, k\}$, and M'_j , $j \in \{1, \dots, k'\}$, are IITMs such that, for every tape name c , at most two of these IITMs have a tape named c and if two IITMs have a tape named c , then c is an input tape in one of the machines and an output tape in the other. Furthermore, there may be at most one tape named **start**, which must be an input tape, as well as at most one tape

named **decision**, which must be an output tape; these special tapes are used for providing external inputs to and obtaining outputs from runs of the system \mathcal{S} (see below). The IITMs M'_j are said to be in the scope of a bang operator. This operator indicates that in a run of a system an unbounded number of (fresh) instances of these machines can be generated. Conversely, if a machine is not in the scope of a bang operator, there may be at most one instance of the machine in every run of the system. Systems in which multiple instances of a machine may be generated are often needed, e.g., in the case of multi-party protocols or in the case a system describes the concurrent execution of multiple instances/sessions of a protocol.

Before explaining runs of systems, we would like to emphasize the difference between a *description* of a machine and an *instance* (or *copy*) of a machine. The difference is the same as the one between program code and a process in an operating system: a process has state and performs the actual actions in a run of a system; it does so following its program code. In our setting, the description of a machine M specifies the behavior of a machine (its program code) and is part of the specification of a system \mathcal{S} . In a run of \mathcal{S} , *instances* of M are created. These instances have a specific state (or configuration), receive input on their input tapes, process the input according to their specifications (program code), thereby updating their state, and produce output. In what follows, for simplicity, we often use the terms *IITMs* and *machines* to denote both static descriptions and instances, depending on the context. More specifically, in the context of systems we mean the static descriptions of machines. In the context of runs of systems or in the context of the runtime behavior of machines, we refer to instances of machines.

In a run of a system \mathcal{S} at any time only one instance of an IITM is active and all other instances wait for new input. The first instance to be activated in a run of \mathcal{S} is an instance of the so-called master IITM, where the master IITM is the one with an input tape named **start**. A system has at most one master IITM, which may get external input (on tape **start**); a run may have several instances of the master IITM, though, if this machine is in the scope of a bang operator (see below). By the definition of IITMs, the active machine may output only at most one message on one of its output tapes, and hence, at most one other machine is triggered after the activation of the currently active machine. To illustrate runs of systems, consider, for example, the system $\mathcal{S} = M_1 |! M_2$ and assume that M_1 has an output tape named c , M_2 has an input tape named c , and M_1 is the master IITM. (There may also be other tapes connecting M_1 and M_2 .) Furthermore, assume that in the run of \mathcal{S} executed so far, two instances of M_2 , say M'_2 and M''_2 , have been generated, with M'_2 generated before M''_2 , and that M_1 just sent a message m on tape c . This message is delivered to M'_2 (as the first copy of M_2). First, M'_2 runs in mode **CheckAddress** with input m ; as mentioned, this is a deterministic polynomial time computation which outputs “accept” or “reject”. If M'_2 accepts m , then M'_2 gets to process m in mode **Compute** and could, for example, send a message back to M_1 . Otherwise, m is given to M''_2 which then runs in mode **CheckAddress** with input m . If M''_2 accepts m , then M''_2 gets to process m in mode **Compute**. Otherwise (if both M'_2 and M''_2 do not accept m), a new copy M'''_2 of M_2 with fresh randomness is generated and M'''_2 runs in mode **CheckAddress** with input

m . (Note that a new copy of M_2 is generated only because in the system description \mathcal{S} M_2 is in the scope of a bang.) If M_2''' accepts m , then M_2''' gets to process m . Otherwise, M_2''' is removed again, the message m is dropped, and an instance of the master IITM is activated with empty input on the tape named `start`, in this case M_1 . More precisely, (the single instance of) M_1 first runs in mode `CheckAddress` to determine whether it accepts the empty input. If M_1 does, it gets to process the empty input. Otherwise, no new instance of M_1 is generated, because M_1 is not in the scope of a bang operator. Now, since M_1 is a master IITM, this means that the run stops. In general, master IITMs can also be in the scope of a bang operator, and hence, analogously to M_2 in the example, new instances of such an IITM can be created. If none of the existing instances of a master IITM nor a newly created master instance accepts an input message, the system run stops. A run also stops if a master instance active in mode `Compute` does not produce output (and hence, does not trigger another machine). If a currently active non-master instance running in mode `Compute` does not produce output, a master instance is triggered (with empty input). In particular, this does not (immediately) terminate a run. Finally, a run also terminates if an instance of a machine (not necessarily a master instance) outputs a message on an output tape named `decision`.¹ Such a message is considered to be the *(overall) output of the system*.

As mentioned in the definition of systems, tape names of machines are unique in the context of the static description of a system: there are no two machines in a system which have an input tape with the same name; analogously for output tapes. For every tape name c , there may, however, exist at most two machines in a system where one has an input tape named c and the other has an output tape named c , which, as explained, means that the two machines are connected. For *instances* of machines, we do not have uniqueness of names, though. For example, both instances M_2' and M_2'' from above have an input tape named c as they are copies of M_2 . Because (the static description of) M_1 has an output tape named c and (the static description of) M_2 has an input tape name c , (an instance of) M_1 can send a message via c to an instance of M_2 . As explained, the `CheckAddress` mode of M_2 is then used to determine which copy of M_2 actually gets to process the message sent by M_1 . If M_1 did not have any output tape whose name coincides with the name of an input tape of M_2 , then M_1 could not send a message to (an instance of) M_2 .

Two systems \mathcal{P} and \mathcal{Q} are called *indistinguishable* ($\mathcal{P} \equiv \mathcal{Q}$) if and only if the difference between the probability that the output of \mathcal{P} is 1 and the probability that the output of \mathcal{Q} is 1 is negligible.

Note that the computational model does not fix details such as addressing of machines by party/session IDs or corruption. It also does not impose a specific structure, e.g., a hierarchical structure with protocols and subroutines, on systems. Those details can be freely instantiated by protocol designers as needed. This makes the IITM model both simple and very expressive.

¹Note that, by the definition of systems, at most one of the specified IITMs can have such an output tape. The instances of that IITM are therefore the only instances that can determine an overall output for the run.

Universal Composability Security Notions. We need the following terminology. For a system \mathcal{S} , the input/output tapes of IITMs in \mathcal{S} that do not have a matching output/input tape in \mathcal{S} are called *external*, where an input/output tape of one IITM in \mathcal{S} matches an output/input tape of another IITM in \mathcal{S} if both tapes have the same name. External tapes are grouped into *I/O* and *network tapes*. I/O tapes are used to model secure direct connections between two machines, for example, to model subroutine relationships where one machines locally calls another one, whereas network tapes model untrusted (network) communication with the adversary/environment or communication with a simulator (see below). We often refer to the sets of I/O and network tapes of \mathcal{S} by *I/O and network interface*, respectively. We consider three different types of systems: protocol systems, adversarial systems, and environmental systems, modeling (i) real and ideal protocols/functionalities, (ii) adversaries and simulators, and (iii) environments, respectively. *Protocol systems*, *adversarial systems*, and *environmental systems* are systems which have an I/O and network interface, i.e., they may have external I/O and network tapes. Adversarial systems may only connect to the network interface of a protocol system but not to its I/O interface. Environmental systems are the only system type that may contain a master machine with input tape **start** and may produce overall output on the output tape **decision**.

So far, we have not restricted the runtime of IITMs in mode **Compute** in any way. The following constraints will be used to enforce that systems run in polynomial time (possibly except with negligible probability): (i) Every environmental system \mathcal{E} has to be *universally bounded*, i.e., there exists a polynomial p such that for all systems \mathcal{S} which connect only to the external tapes of \mathcal{E} , we have that the overall runtime of \mathcal{E} in mode **Compute** in runs of the system $\mathcal{E} | \mathcal{S}$, with security parameter η and external input a , is bounded by $p(\eta + |a|)$. (ii) A protocol system \mathcal{P} has to be *environmentally bounded*, i.e., for all environmental systems \mathcal{E} there exists a polynomial p such that the overall runtime of \mathcal{P} in mode **Compute** in runs of the system $\mathcal{E} | \mathcal{P}$, with security parameter η and external input a , is bounded by $p(\eta + |a|)$ (in all runs except for a negligible set of runs). Since the runtime in mode **CheckAddress** is already polynomially bounded, this guarantees that, for a protocol system \mathcal{P} and environmental system \mathcal{E} , the overall runtime of $\mathcal{E} | \mathcal{P}$ is polynomially bounded in the security parameter plus the length of the external input (except with negligible probability). This runtime notion for protocol systems is very general. We claim that it includes all reasonable protocol systems that occur in applications, as further explained in [76, 99]; we are in fact not aware of any natural protocol that does not meet this runtime notion. In most other models, such as Canetti’s UC model, the runtime notions are more restricted and more complex.

We now informally define the security notion of strong simulatability; other equivalent security notions, such as universal composability (UC), dummy UC, and reactive simulatability, can be defined in a similar way (cf. [99] for definitions of these notions in the IITM model. We also define and show equivalence of all of these notions for our extension of the IITM model in Section 3.2.6). The systems considered in the following definition are illustrated in Figure 2.1.

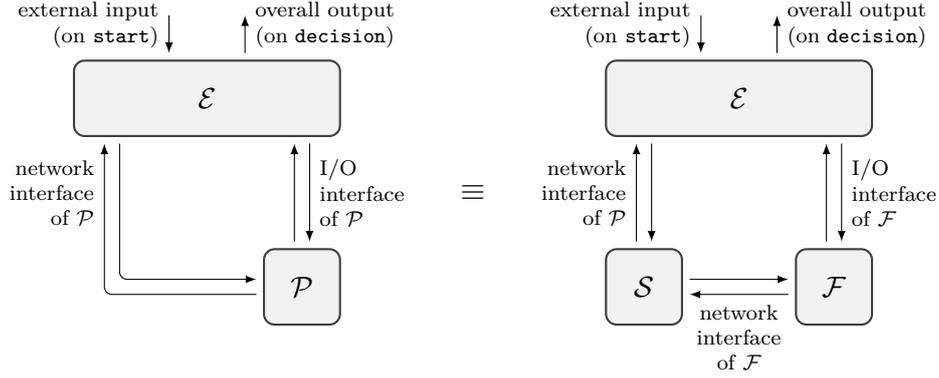


Figure 2.1.: The security notion of strong simulatability (SS). We note that \mathcal{P} and \mathcal{F} have the same I/O interface.

Definition 2.1 (informal). *Let \mathcal{P} and \mathcal{F} be protocol systems with the same I/O interface (i.e., with the same set of I/O tapes), usually called the real and the ideal protocol, respectively. Then, \mathcal{P} realizes \mathcal{F} ($\mathcal{P} \leq \mathcal{F}$) if and only if there exists an adversarial system \mathcal{S} (called a simulator or an ideal adversary) such that \mathcal{S} connects only to the network interface of \mathcal{F} , the systems \mathcal{P} and $\mathcal{S} \mid \mathcal{F}$ have the same external interface, $\mathcal{S} \mid \mathcal{F}$ is environmentally bounded, and for all environmental systems \mathcal{E} , connecting only to the external interface of \mathcal{P} (and hence, $\mathcal{S} \mid \mathcal{F}$), it holds that $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$.*

In the above definition, the setting $\mathcal{E} \mid \mathcal{P}$ is often referred to as the *real world*, whereas $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ is often called the *ideal world*.² Note that the security notion of strong simulatability, in contrast to the notion of universal composability presented in the introduction (cf. Chapter 1 and Figure 1.1 in that chapter), does not consider a network attacker \mathcal{A} in the real world but instead lets the environment directly subsume the role of the network attacker. We are justified in using the conceptually simpler notion of strong simulatability since, as mentioned, both notions are actually equivalent within the IITM model.

Composition Theorems. Composition theorems allow for the modular analysis and design of systems and are one of the main features of the universal composability paradigm. The first and main composition theorem handles concurrent composition of a fixed number of (different) protocol systems. A second theorem guarantees secure composition of an unbounded number of instances of a protocol system.

Theorem 2.1 (informal). *Let $k \geq 1$. Let $\mathcal{Q}, \mathcal{P}_1, \dots, \mathcal{P}_k, \mathcal{F}_1, \dots, \mathcal{F}_k$ be protocol systems such that they connect only via their I/O interfaces, $\mathcal{Q} \mid \mathcal{P}_1 \mid \dots \mid \mathcal{P}_k$ is environmentally bounded, and $\mathcal{P}_i \leq \mathcal{F}_i$, for $i \in \{1, \dots, k\}$. Then, $\mathcal{Q} \mid \mathcal{P}_1 \mid \dots \mid \mathcal{P}_k \leq \mathcal{Q} \mid \mathcal{F}_1 \mid \dots \mid \mathcal{F}_k$.*

²We sometimes also use the term *real world* to denote (the behavior of) a protocol in reality, i.e., when an actual implementation runs on physical computers, as opposed to (the behavior of) a protocol as modeled for the security analysis. The meaning is always clear from the context.

Note that this theorem does not require that the protocols $\mathcal{P}_i/\mathcal{F}_i$ are subprotocols of \mathcal{Q} , i.e., that \mathcal{Q} has matching external I/O tapes for all of these protocols. How these protocols connect to each other via their I/O interfaces, if at all, is not restricted in any way; even the environment could connect directly to the parts of the I/O interfaces of these protocols that are not taken by another protocol.

The second composition theorem, informally speaking, allows for analyzing a single session of a protocol \mathcal{P} in isolation, provided that the sessions of \mathcal{P} are disjoint, i.e., do not directly interact with each other and in particular do not share any state. The composition theorem then implies that an unbounded number of sessions running concurrently is still secure. To state this theorem, we first have to define “protocol with disjoint sessions”, which we do via *protocol session identifier functions* and the notion of a *session version*.

We consider a (polynomially computable) *protocol session identifier function* σ which, given a message and a tape name, outputs a protocol session identifier (PSID, which is simply a bit string)³ or \perp . For example, the following function takes the prefix of a message as its PSID, i.e., the prefix of a message identifies a session of a protocol run: $\sigma_{\text{prefix}}(m, c) := s$ if $m = (s, m')$ for some s, m' and $\sigma_{\text{prefix}}(m, c) := \perp$ otherwise, for all m, c . Clearly, many more examples are conceivable. The reason that σ , besides a message, also takes a tape name as input is that the way SIDs are extracted from messages may depend on the tape name a message is received from.

Now, we say that an IITM M is a σ -*session machine* (or a σ -*session version*) if the following conditions are satisfied: (i) M rejects (in mode `CheckAddress`) a message m on tape c if $\sigma(m, c) = \perp$. (ii) If m_0 is the first message that M accepted (in mode `CheckAddress`), say on tape c_0 , in a run, then, M will reject all messages m received on some tape c (in mode `CheckAddress`) with $\sigma(m, c) \neq \sigma(m_0, c_0)$. (iii) Whenever M outputs a messages m on tape c (in mode `Compute`), then $\sigma(m, c) = \sigma(m_0, c_0)$, with m_0 and c_0 as before. We say that a system \mathcal{Q} is a σ -*session system* (or a σ -*session version*) if every IITM occurring in \mathcal{Q} is a σ -session machine. Intuitively, such a σ -session system guarantees that instances belonging to different sessions do not interact with each other in any way, neither by sending messages across sessions nor by using the same state or machine instances in multiple sessions.

We call an environmental system \mathcal{E} σ -*single session* if it only outputs messages with the same SID according to σ . Hence, when interacting with a σ -session version, such an environmental system invokes at most one protocol session.

Let \mathcal{P} and \mathcal{F} be protocol systems, which in the setting considered here would typically describe multiple sessions of a protocol. Moreover, we assume that \mathcal{P} and \mathcal{F} are σ -session versions. Now, we define what it means that a single session of \mathcal{P} realizes a single session of \mathcal{F} . This is defined just as $\mathcal{P} \leq \mathcal{F}$, with the difference that we consider only σ -single session environments, and hence, environments that invoke at most one session of \mathcal{P} and \mathcal{F} .

³We note that in [99] these functions were just called *session identifier (SID) functions*. To avoid any confusion with the concept of session identifiers (SIDs) introduced in the iUC framework in Chapter 4, which are used for a different purpose, we have renamed *SID functions* to *PSID functions* in this thesis.

Definition 2.2 (informal). *Let \mathcal{P} , \mathcal{F} , and σ be as above. Then, \mathcal{P} single-session realizes \mathcal{F} w.r.t. σ ($\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$) if and only if there exists an adversarial system \mathcal{S} (a simulator or an ideal adversary) such that $\mathcal{E}|\mathcal{P} \equiv \mathcal{E}|\mathcal{S}|\mathcal{F}$ for every σ -single session environmental system \mathcal{E} . (The details concerning runtime and interfaces are similar to Definition 2.1.).*

Now the following theorem says that if \mathcal{P} realizes \mathcal{F} w.r.t. a single session, then \mathcal{P} realizes \mathcal{F} w.r.t. multiple sessions.

Theorem 2.2 (informal). *Let σ , \mathcal{P} , and \mathcal{F} be as above. Then, $\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$ implies $\mathcal{P} \leq \mathcal{F}$.*

This theorem can also be combined with the theorem for concurrent composition to construct more and more complex systems. For example, if we already know that $\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$ and $\mathcal{Q}|\mathcal{P}$ is environmentally bounded, then by the combination of both composition theorems we can conclude that $\mathcal{Q}|\mathcal{P} \leq \mathcal{Q}|\mathcal{F}$.

We emphasize that details such as addressing of machines by party/session IDs, corruption, and structure of protocols are still not, and do not need to be fixed in order to prove the above composition theorems. In other words, these theorems hold true for all specific choices, and hence, are very general. Instead of fixing these details, as done in other models, the IITM model leaves their definition to the protocol designer, hence providing a greater degree of freedom for protocol specifications.

Protocol types. The universal composability literature often considers certain types/classes of protocols which are supposed to model certain situations. Here, we briefly recap standard terminology for those types of protocols and explain what they mean in the context of the IITM model. We emphasize that all of the following protocol types are mere special cases of the general protocol definition and hence are directly supported by the composition theorems of the IITM model.

- **Real protocol:** A real protocol models an actual implementation of a protocol as it would run in reality. This is typically the protocol that one wants to analyze and prove secure, i.e., the protocol \mathcal{P} from the realization relation (cf. Definition 2.1).
- **Ideal protocol/ideal functionality:** An ideal protocol provides an ideal specification of a certain task, i.e., provides expected security and functional properties by definition. Generally, these protocols cannot be implemented in reality and instead need to first be realized via a real protocol. Typically the protocol \mathcal{F} in the realization relation is an ideal protocol, i.e., the ideal protocol defines the expected (perfect) behavior that the (usually real) protocol \mathcal{P} is supposed to achieve.

Traditionally, ideal protocols in many UC-like models are built from two components/ machines: a so-called ideal functionality that contains the actual logic of the protocol, and so-called dummy parties whose only purpose is to forward messages on I/O connections between the environment and the ideal functionality. These dummy parties are necessary

in some models due to rigid addressing mechanisms which do not allow the same machine instance to directly accept messages meant for different protocol participants. In the IITM model with its very flexible addressing mechanism one does not need dummy parties, i.e., an ideal protocol can consist only of an ideal functionality that directly accepts all messages for multiple parties. Hence, we use the term “ideal functionality” synonymously to the term “ideal protocol” in this thesis.

- **Hybrid protocol:** A hybrid protocol is a real protocol that uses an ideal protocol as a subroutine. The intention of a hybrid protocol is either to model a setup assumption via an ideal subroutine, such as a common reference string that is available to all participants of a protocol, or to perform a modular analysis of a protocol by first proving security based on an ideal subroutine and then replacing that ideal subroutine with a realization using the composition theorem (as explained in the introduction, this is usually much easier than directly using the realization as ideal protocols typically are simpler and provide absolute security guarantees).
- **Protocol with disjoint sessions:** A protocol has disjoint sessions if all instances can be grouped into separate sets that do not interact with each other, where each set forms a protocol session. In the IITM model, this corresponds to the notion of a session version protocol and hence allows for using the unbounded self-composition theorem.
- **Joint state realization/protocol:** This is a (typically hybrid) protocol that realizes another (typically ideal) protocol with disjoint sessions, but reuses some state throughout all protocol sessions such as signature keys. That is, a joint state realization behaves just as a protocol with disjoint sessions, even though it actually reuses some state in multiple sessions.

Historically, this type of protocol was proposed because most universal compossibility models only supported (real/ideal/hybrid) protocols with disjoint sessions (cf., e.g., [51, 75]), i.e., those models needed some extension that allowed them to also capture reusing some state. The IITM model supports joint state realizations by default, as this type of protocol is just another special case of the general protocol definition. Detailed discussions and examples of how joint state can be modeled in the IITM model are available in [94, 99]. We also provide a discussion of joint state in our instantiation iUC in Section 4.4, including an example joint state realization which is given in Appendix C. We note, however, that joint state realizations are not necessary in the IITM model to express shared state; instead, protocol designers can also directly model (real/ideal/hybrid) protocols that share state in arbitrary ways. In particular, such protocols do not need to behave as if they had disjoint sessions, i.e., the behavior of one protocol session can strongly depend on and directly influence other sessions.

- **Protocol with global state:** A protocol with global state (also called global subroutine, global functionality, or global setup) shares one or more of its subroutines with other protocols/the environment. For example, a global subroutine of some protocol \mathcal{P} could be used to

model a common reference string that can then be reused by other independent protocols, which might not even be known by the time that the protocol \mathcal{P} is analyzed.

Similar to joint state, protocols with global state were historically introduced because most universal composability models did not support protocols with subroutines that are available to and shared with other protocols/the environment (cf., e.g., [41, 75]). Just as for joint state protocols, in the IITM model protocols with global state are also a mere special case of the general protocol definition; in particular, as already highlighted after Theorem 2.1, there are no limitations on which parts of a protocol can offer free I/O tapes for other protocols/the environment to connect to. A detailed discussion of how global state can be modeled in the IITM model is available in [99]. We also provide a discussion of global state in our instantiation iUC in Section 4.4, including an example protocol with global state which is given in Section 4.3.

3. The IITM Model With Responsive Environments

In this chapter, we propose our IITM model with responsive environments, which was originally published in [29]. As already explained in the introduction (cf. Chapter 1), this is an extension and generalization of the original IITM model which solves and gets rid of the non-responsiveness problem and all of its consequences.

This chapter is structured as follows. In Section 3.1, we first discuss the non-responsiveness problem and its consequences in more detail, including examples from the literature. We then present our IITM model with responsive environments in Section 3.2, including formal definitions as well as full proofs of the composition theorems. We discuss applications of responsive environments in Section 3.3 and discuss the concept of responsive environments in the context of other universal composability models in Section 3.4. Related work is presented in Section 3.5.

3.1. The Non-Responsiveness Problem and Its Consequences

Recall from the introduction that universal composability models currently do not guarantee that urgent requests are answered immediately by the adversary, which we call the *non-responsiveness problem*. In the following, we illustrate the non-responsiveness problem and its consequences in more detail by examples from the literature, showing that this problem is very hard to deal with and can even prevent modeling certain expected properties of protocols. We also point to concrete cases in which this problem has been ignored (i.e., immediate answers to urgent requests were assumed implicitly) and where this has led to ill-defined protocols and functionalities as well as invalid proofs and statements. Furthermore, we argue that there is no simple and general way to deal with the non-responsiveness problem in existing universal composability models.

3.1.1. Underspecified and Ill-Defined Protocols

In many papers, the non-responsiveness problem is ignored in the specifications of protocols. We discuss a number of typical cases in the following.

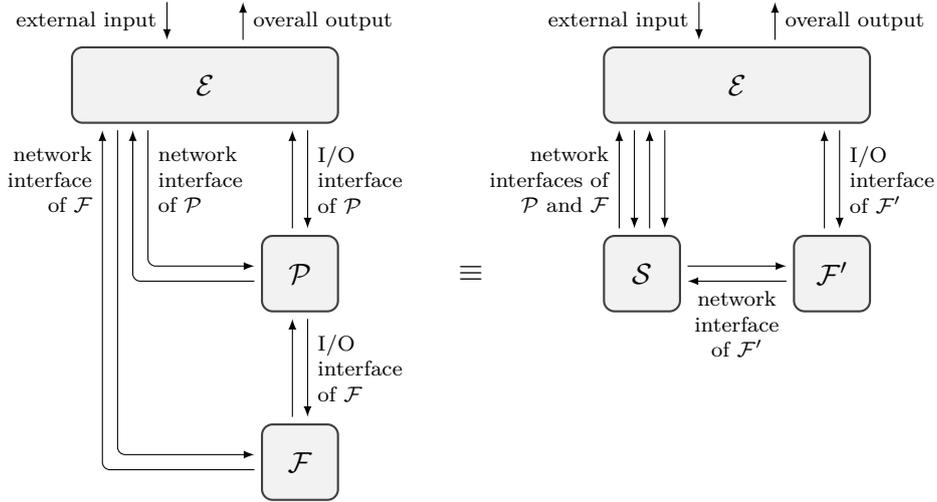


Figure 3.1.: A hybrid protocol \mathcal{P} using the ideal subroutine \mathcal{F} to realize some other ideal functionality \mathcal{F}' .

Ideal functionalities. An example of a statement that one often finds in specifications of ideal functionalities is one like the following (see, e.g., [2, 10, 52, 74, 86, 100, 101]):

$$\begin{aligned} & \text{“send <some message> to the adversary;} \\ & \text{upon receiving <some answer> from the adversary do <something>”,} \end{aligned} \tag{3.1}$$

where the message sent to the adversary, in our terminology, is an urgent request, i.e., some meta-information provided to the adversary or a request for some meta-information the adversary is supposed to provide. For example, ideal functionalities might ask for cryptographic material (cryptographic algorithms and keys, ciphertexts, or signatures), ask whether the adversary wants to corrupt a party, or simply signal their existence.

In specifications containing formulations as in (3.1) it is not specified what happens if the adversary does not respond immediately, but, for example, other requests on the I/O interface are received; intermediate state changes in other parts might also occur, which might require different actions (cf. Section 3.1.2). There does not seem to exist a generic solution to handle such problems in existing models (cf. Section 3.1.4). It rather seems to be necessary to find solutions tailored to the specific protocol and ideal functionality at hand, making it even more important to precisely specify the behavior in case the adversary does not respond immediately to urgent requests.

Many research papers on universal composability focus on proposing new functionalities and realizations thereof, including proofs that a realization actually realizes a functionality; to a lesser extent the functionalities are then used in higher-level protocols. In realization proofs, one might not notice that formulations as that in (3.1) are problematic because for such proofs an ideal functionality \mathcal{F} runs alongside a (friendly) simulator and this simulator might indeed provide answers to urgent requests immediately (cf. Figure 2.1). However, if used in

Upon receiving a value (Setup, sid) from any party P , verify that $sid = (M_L, sid')$ for some sid' . If not, then ignore the request. Else, if this is the first time that (Setup, sid) was received, **hand (Setup, sid) to the adversary; upon receiving $(\text{Algorithms}, sid, \text{Verify}, \text{Sign}, \text{Simsign}, \text{Extract})$ from the adversary**, where Sign , Simsign , Extract are descriptions of PPT TMs, and Verify is a description of a deterministic polytime TM, store these algorithms. Output the stored $(\text{Algorithms}, sid, \text{Sign}, \text{Verify})$ to P .

Figure 3.2.: The Setup instruction of $\mathcal{F}_{\text{SoK}}(L)$ from [57].

a hybrid protocol (cf. Figure 3.1), an ideal functionality \mathcal{F} runs alongside a (hostile) adversary/environment controlling the network. In this case, it is important that specifications also capture the case that urgent requests are not answered immediately. If this is ignored or not handled correctly, it yields (i) underspecified protocols, with the problem that they cannot be reused in hybrid protocols, which in turn prevents modular security proofs and hence defeats the main purpose of the universal composability approach, and (ii) possibly false statements.

To illustrate these points by a concrete example, we consider the “signature of knowledge” functionality $\mathcal{F}_{\text{SoK}}(L)$ proposed by Chase and Lysyanskaya [57]. This functionality contains a Setup instruction (reproduced in Figure 3.2), where the adversary provides the keys and algorithms, and signing and verification instructions that then use those keys and algorithms without requiring interaction with the adversary - a very common mechanism in the literature (see, e.g., [2, 28, 50, 63, 67, 111]). This functionality is explicitly intended to be used in a hybrid setting to realize delegatable credentials.

If the adversary does not respond to the first (Setup, sid) request, all subsequent requests (e.g., a Setup request by a different party) will cause the functionality to use or output the undefined Sign and Verify algorithms, which is a problem: Chase and Lysyanskaya provide a hybrid protocol that uses $\mathcal{F}_{\text{SoK}}(L)$ as a subroutine for realizing delegated credentials, i.e., an ideal functionality for signatures on signatures. They then prove that this protocol realizes the functionality. They, however, missed the fact that $\mathcal{F}_{\text{SoK}}(L)$ may interact with a non-responsive adversary in the hybrid world which might not respond to the setup query, thus forcing $\mathcal{F}_{\text{SoK}}(L)$ to use undefined algorithms. Chase and Lysyanskaya do not handle that case in their simulation in the ideal world. It is thus easy to distinguish the real from the ideal world. Hence, their proof is flawed, and in fact it seems that the statement cannot be proven at all.

Other types of protocols and higher-level protocols. As already mentioned in the introduction, urgent requests are not specific to ideal protocols but other types of protocols, including real and hybrid ones, also often also send urgent requests to the adversary (e.g., signaling their existence or asking whether the adversary wants to corrupt). This results in the same issues as described above for ideal protocols. In addition, one often finds specifications of higher-level protocols containing formulations of the following form to make requests to (typically ideal)

subprotocols (see, e.g., [28, 57, 111, 116]):

$$\begin{aligned} & \text{“send <some message> to } \mathcal{F}; \\ & \text{upon receiving <some answer> from } \mathcal{F} \text{ do <something>.”} \end{aligned} \tag{3.2}$$

Intuitively, the subroutine \mathcal{F} might indeed model some non-interactive task. For example, \mathcal{F} might be an ideal protocol for computing ideal signatures. However, because of the common use of urgent requests in protocols, even when completely uncorrupted, \mathcal{F} might not return answers right away. So, again, formulations as the one in (3.2) are greatly underspecified. What happens if other requests are received at the network or I/O interface? Should they be ignored? Or may they be queued somehow? Also, the state and status (such as corruption) of other parts of the protocol or subprotocols might change while waiting for answers from \mathcal{F} . Again, as illustrated in the following subsections, dealing with this is not easy and often requires solutions tailored to the specific protocols at hand, making a full specification of the behavior particularly important.

3.1.2. Unintended State Changes and Race Conditions

As mentioned before, a general problem one has to take care of when dealing with the non-responsiveness problem is that while a protocol is waiting for answers to urgent requests, the adversary might cause changes in the state of other parts of the protocol/functionality and of subprotocols, which in turn influences the behavior of the protocol. Keeping track of the actual current overall state often is tricky, and race conditions are possible. The following is a simple example which illustrates that the problem can occur already locally within a single functionality. It can often become even trickier in higher-level protocols which use urgent requests themselves and where possibly several subroutines use urgent requests.

We consider the dual-authentication certification functionality $\mathcal{F}_{\text{D-Cert}}$ from [116]. In this functionality, the adversary needs to be contacted when verifying a signature (a common mechanism to verify cryptographic values that is also used in many other functionalities [37, 52, 86]). Such requests are urgent as this is supposed to model local computations. However, the adversary may not answer immediately.

More specifically, Figure 3.3 shows the Verify instruction of $\mathcal{F}_{\text{D-Cert}}$. Assume now that party S' has received a message m and a signature σ for this message, which supposedly was created by an honest party P with SID sid . Now, if the signature actually was not created by P , the verification should fail as P is not corrupted. However, as the adversary gets activated during this allegedly local task, it could corrupt the signer during the verification process, return $\phi = 1$, and therefore let the functionality *accept* σ . This behavior is certainly unexpected and counterintuitive.

Such a functionality also considerably complicates the security analysis of any higher-level application that uses $\mathcal{F}_{\text{D-Cert}}$ as a subroutine, as one has to also consider the possibility of a party getting corrupted during the invocation of a subroutine modeling a local task, which, even worse, in that case returns unexpected answers.

Upon receiving a value $(\text{Verify}, sid, m, \sigma)$ from some party S' , **hand** $(\text{Verify}, sid, m, \sigma)$ to the adversary. Upon receiving $(\text{Verified}, sid, m, \phi)$ from the adversary, do:

1. If $(m, \sigma, 1)$ is recorded then set $f = 1$.
2. Else, if the signer is not corrupted, and no entry $(m, \sigma', 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, \sigma, 0)$.
3. Else, if there is an entry (m, σ, f') recorded, then set $f = f'$.
4. Else, set $f = \phi$, and record the entry (m, σ, ϕ) .

Output $(\text{Verified}, sid, m, f)$ to S' .

Figure 3.3.: The Verify instruction of $\mathcal{F}_{\text{D-Cert}}$ from [116].

Upon input (init, P_i, P_j) from P_i , if $P_j \notin \Lambda_{\text{ref}}$, return (P_i, P_j, \perp) to P_i . If $P_j \in \Lambda_{\text{reg}}$, we consider two cases:

- Corrupted session mode: if there exists an entry $(\{P_i, P_j\}, K_{i,j})$ in Λ_{keys} , set $key = K_{i,j}$. Else, **send** (init, P_i, P_j) to the adversary. **After receiving** $(\{P_i, P_j\}, K_{i,j})$ from the adversary, set $key = K_{i,j}$ and add $(\{P_i, P_j\}, K_{i,j})$ to Λ_{keys} .
- Honest session mode: if there exists an entry $(\{P_i, P_j\}, K_{i,j})$ in Λ_{keys} , set $key = K_{i,j}$, else choose $key \leftarrow \{0, 1\}^k$ and add $(\{P_i, P_j\}, K_{i,j})$ to Λ_{keys} .

Return (P_i, P_j, key) to P_i .

Figure 3.4.: The init instruction of $\mathcal{F}_{\text{NIKE}}$ from [72].

3.1.3. Problems Expressing the Desired Properties

The following is an example where the authors struggled with the non-responsiveness problem in that it finally led to a functionality that, as the authors acknowledge, is not completely satisfying. This functionality, denoted $\mathcal{F}_{\text{NIKE}}$, is supposed to model a non-interactive key exchange and was proposed by Freire et al. [72]. Figure 3.4 shows a central part of this functionality, namely, the actual key exchange. A party P_i may ask for the key that is shared between the parties P_i and P_j . If this session of P_i and P_j is considered corrupted, namely, because one of the parties is corrupted, and no key has been recorded for this session yet, the adversary is allowed to freely choose the key that is shared between the two parties. The functionality uses an urgent request to model this, i.e., it directly sends a message to the adversary if he is allowed to choose a key.

As the authors state, they would have liked to also model “immediateness” of the functionality, i.e., a higher-level protocol that requests a key should be able to expect an answer without the adversary being able to interfere with the protocol in the meantime. This indeed would be expected and natural because $\mathcal{F}_{\text{NIKE}}$ models a *non-interactive* key exchange. However, this is in conflict with allowing the adversary to choose the key of a corrupted session. The authors suggest that one option to also model immediateness might be to let the adversary choose an

algorithm upon setup, which is then used to compute the keys for corrupt parties. Nevertheless, they chose the non-immediate modeling because the other solution would lead to “technical complications”; it would also limit the adaptiveness of the adversary and might add other problems. Indeed, code upload constructs (see also Section 3.1.4), in general, do not solve the non-responsiveness problem.

As a consequence of the formulation chosen in $\mathcal{F}_{\text{NIKE}}$, the adversary can now, e.g., block requests, which again also needs to be considered in any higher-level protocol using $\mathcal{F}_{\text{NIKE}}$ as a subroutine, even though in the real world the honest party would always obtain some key due to the non-interactivity of the primitive.

More generally, ideal functionalities that use urgent requests (which in current models are not answered immediately) might have *weaker security guarantees than their realizations*, in particular when the functionality is supposed to model a non-interactive task, because the realization might not give control to the adversary. So for hybrid protocols one might not be able to prove certain properties when using an ideal functionality, whereas the same protocol using the realization of the ideal functionality instead might enjoy such properties.

This is in contrast to one of the goals of universal composability models, namely, reducing the complexity of security analyses by enabling the use of conceptually simpler ideal functionalities as subroutines.

3.1.4. The Reentrance Problem

As explained in Section 3.1.1, a protocol designer has to specify the behavior of protocols upon receiving another input (on the I/O interface) while they are waiting for a response to an urgent request on the network. In other words, protocols have to be *reentrant*. Note that, as pointed out, a protocol has to be reentrant not only when it uses urgent requests itself, but also if a subroutine uses such messages.

As explained next, making a protocol reentrant can be very difficult. We call this the *reentrance problem*. Approaches to solve this problem complicate the specifications of protocols and none of those approaches is sufficiently general to be applicable in every case.

We now illustrate this by an example ideal functionality. However, similar issues occur in specifications for other types of protocols, including real and hybrid ones. Let \mathcal{F} be any ideal functionality which sends an urgent request to the adversary upon its first creation, say, to retrieve some modeling-related information. This is a common situation. For example, ideal functionalities often require some cryptographic material such as keys and algorithms from the adversary before they can continue their execution (e.g., functionalities for digital signatures or public-key encryption). We also assume that \mathcal{F} is meant to be realized by a real protocol \mathcal{P} consisting of two independent parties/roles A and B (e.g., signer and verifier). We further assume that both of these parties also send an urgent request to the adversary upon their first activation and expect an answer before they can continue with their computation. Again, this is a common situation as, for example, real protocols often ask for their corruption status or

notify the adversary of their creation.⁴ While the above is only one illustrative example, it already describes a large and common class of real and ideal protocols often encountered in the literature.

We now present several potential approaches to try to make \mathcal{F} reentrant in the above sense, i.e., to deal with intermediate I/O requests while waiting for a response to an urgent request on the network. We show that the obvious approaches in general cannot be used. In particular, with most of these approaches \mathcal{F} cannot even be realized by A and B in the setting outlined above. This in turn shows that solutions that are tailored to the specific functionality at hand and even the envisioned realization are required, which is very unsatisfactory as this leads to more complex and yet less general functionalities and protocols.

Ignore requests or return error messages. After sending an urgent request to the adversary, the most straightforward approach would be to ignore all incoming messages until a response from the adversary is received.⁵ This, however, is not only an unexpected behavior in many cases – for example, why should a request silently fail if the ideal functionality models a local computation? – but the ideal functionality in fact might no longer be realizable by some classes of real protocols:

If \mathcal{F} , in our example functionality, would simply ignore incoming messages, an environment can distinguish \mathcal{F} (with a simulator) from the realization A and B . It first sends a message to A which, as we assume, then in turn sends an urgent request to the dummy adversary and hence to the environment. Now the environment, which does not have to respond to urgent requests immediately, sends a message to B which in turn also sends an urgent request to the adversary and hence to the environment. Consider the behavior of the ideal world in this case: After receiving the message for A , \mathcal{F} will send an urgent request to the simulator. The simulator, however, generally cannot answer this urgent request because it has to simulate A by sending an urgent request to the environment. (This might be the case because the simulator first has to consult the environment before answering the urgent request by \mathcal{F} or because \mathcal{F} does not return control to the simulator after receiving an answer to the urgent request.) The environment then sends the second message (for B) to \mathcal{F} , which is ignored because \mathcal{F} still waits for an answer to its urgent request. This behavior is different from the real world, and thus, the environment can distinguish the real world from the ideal one.

This illustrates that an ideal functionality that simply blocks *all* requests while waiting for a response to an urgent request can in general not be realized by two or more *independent* parties that also send urgent requests to the adversary. Instead one needs to adjust the blocking approach to the specific protocols at hand. For example, sometimes it might be possible to

⁴ Such notifications are, for example, required by the 2013 version of the UC model [35] (see the definition of “subroutine respecting protocols”). While prompt responses by the adversary are formally not required, they would be very convenient for all of the reasons discussed in this Section.

⁵ Alternatively, one could send error messages as response to intermediate requests. However, the exact same problems discussed for the approach of ignoring requests occur. For simplicity of presentation, we mention only ignoring requests in the following.

block messages that would be processed by a single party in the real protocol, while messages for other parties are still processed. But this does not work if, for instance, \mathcal{F} cannot process messages for any party before receiving a response to its urgent requests, e.g., because \mathcal{F} first needs to receive cryptographic material (algorithms, keys, etc.). Thus, in this case yet another workaround is required.

Queuing of intermediate requests. Another general approach to try to deal with the reentrance problem is to store all incoming messages to process them later on. The simplest implementation of this approach would be the following: Upon receiving another input while still waiting for a response to an urgent request, the ideal functionality stores the input in a queue and then ends its activation. After receiving a response from the adversary, the ideal functionality processes the messages stored in the queue.

This approach is vulnerable to the same attack as the previous approaches: if the environment executes this attack in the real world, it will eventually receive an urgent request from B . This, however, cannot be simulated in the ideal world. The simulator does not get control when B is activated as the ideal functionality simply ends its activation after queuing the input for B .

Another problem with this approach is that in all current universal composability models, a machine is allowed to send only one message per activation. Hence, the ideal functionality will never be able to catch up with the inputs that have been stored. Every time it is activated by another input, it will have to process both the new input and several older inputs that are still stored in the queue. But it can only answer one of these messages at a time. This observation leads to another more advanced and complex approach based on the queuing of unanswered requests which we discuss in Appendix A. This approach, which does not seem to have been used in the literature so far, is, however, very complex and weakens the security of the ideal functionality to an extent that for some tasks is unacceptable: it allows the adversary to determine the order in which requests are processed by an ideal functionality.

Using a default. To solve the non-responsiveness of the adversary at least for those cases when the functionality \mathcal{F} asks for some meta-information, such as algorithms or cryptographic keys, one might be tempted to add a default to the functionality such that this default will be used if the next message received is not from the adversary or does not contain the expected information.

While this definition seems straightforward at first, it actually is much more complex as soon as one tries to implement it. Note that after the ideal functionality has sent its urgent request, the next activation may be due to a second input on the I/O interface. The ideal functionality would then have to process and send responses to two requests at the same time (while using the default), which is not possible in any of the current universal composability models. Dropping one of the two requests generally is not an option, as this leads to exactly the same problems as described for the *ignore requests* approach. Hence, a queuing approach is necessary in which messages are processed one after another. As discussed above, the simplest form of a queue

does not work in general either, whereas a much more complex form (cf. Appendix A) severely weakens the security guarantees of ideal functionalities and hence is usually unacceptable.

Besides the problems mentioned above, there are further issues with this approach. First, it might not always be possible to find a sensible default. Second, in some settings, the environment can force the use of the default algorithm and thus distinguish the real and the ideal world. To see this, consider the previous running example and the attack presented in the *ignore requests* paragraph. In that setting, \mathcal{F} would be forced to use the default; the simulator has no way to first provide other values because it has to simulate the network traffic of A before \mathcal{F} produces output on the I/O interface (which happens as soon as the simulator provides the algorithms). Hence, any simulation that depends on the simulator being able to choose the information that is used by \mathcal{F} will fail.

Not asking for information at all (code upload constructs). Alternatively to using a default, one could try to eliminate those urgent requests from \mathcal{F} that ask for some information from the adversary by requiring that the adversary actively sends the required information before it is needed. Let us be more precise: In this approach, \mathcal{F} accepts a special message containing some data from the adversary at any point in time. Then, as soon as the ideal functionality has to process a request from the I/O interface, it uses the data it has previously received. If it has not received any data prior to the request, it resorts to a default (and stays with this default even if the adversary later provides different data). One extreme form of this variant are code upload constructs, in which the adversary first provides some algorithm, which is then internally simulated by the ideal functionality to generate answers to any (urgent) request the functionality might have sent otherwise.

However, this approach, which tries to eliminate one type of urgent requests altogether, also cannot be used in general. In particular, an environment might again be able to force the use of the default to be able to distinguish between the real and the ideal world. To see this, assume that \mathcal{F} requires some algorithm at the beginning (e.g., signing and verification algorithms in the case of an ideal signature functionality \mathcal{F}_{sig}) and otherwise does not contact the adversary/simulator at all because it models a local computation. Also assume that there can be multiple instances of \mathcal{F} , say, to model multiple sessions running in parallel, where each instance is addressed using a session identifier (SID). Let \mathcal{P} be an intended realization of \mathcal{F} that uses a specific fixed cryptographic algorithm. To simulate \mathcal{P} , the simulator has to provide the same algorithm (or a variant of it) to \mathcal{F} . As \mathcal{F} does not send an urgent request asking for the algorithm but rather uses a code upload construct, the simulator now has to somehow figure out when and to which instance of \mathcal{F} to send the required algorithm. In particular, the simulator must send this information before (the instance of) \mathcal{F} is used for the first time. Even if it is guaranteed that the simulator is activated before \mathcal{F} (such as in the UC model, where the adversary/simulator is always activated directly after the environment), the simulator would still have to know which session of \mathcal{F} is going to be used by the environment. However, the environment is not forced to tell the simulator which SID it is going to use for the challenge

session, and thus a simulator can only guess at this point (note that the simulator will be wrong most of the time because it can only guess a polynomial number of sessions, whereas the environment is free to choose from an exponential number). It is also not possible to provide the algorithm later on because in this approach \mathcal{F} does not contact the simulator at all. Hence, as soon as the environment regains control, it can access \mathcal{F} directly for some random SID, and thus force (the instance of) \mathcal{F} to use the default algorithm (with overwhelming probability), which is easily distinguishable from \mathcal{P} in general.

A variant of this approach defines a special instance with a fixed ID which gets all data from the adversary/simulator. This instance is then used as a subroutine by all ideal functionalities that need to access this data. While this fixes the above issue by defining a single, known ID for the instance to which the simulator has to send its data, it comes at a hefty price: It is no longer possible to use the composition theorem for multiple sessions. In this setting, all sessions access *the same* subroutine instance, and hence sessions are no longer disjoint, as required by the composition theorem. In fact, many UC-like models require sessions to not share any state at all and therefore do not support this variant of the data/code upload approach in the first place.

We emphasize that both the *use default* and *code upload constructs* approaches only seek to address the issue of urgent requests in which \mathcal{F} wants to retrieve information *from* the adversary. However, urgent requests that are meant to provide information *to* the adversary are not addressed at all. So even in cases where one of these two approaches can be applied, which depends on the concrete functionalities and intended realizations, it only fixes part of the problem.

In summary, we conclude that there is no general solution for dealing with the reentrance problem in existing models. Instead, if a solution exists at all, it must be tailored towards the protocols at hand, which makes protocols less general while also increasing the overall complexity.

3.1.5. Unnatural Specifications of Higher-Level Protocols

As explained above, higher-level protocols have to deal with the non-responsiveness problem for two reasons. First, they might use urgent requests themselves. Second, subprotocols might use urgent requests, and hence, if requests are sent to subprotocols (even for those that intuitively should model non-interactive primitives), the adversary might get control. In both cases, higher-level protocols have to deal with the problem that while waiting for answers, the state of other parts of them and of any of their subprotocols might change and new requests might have to be processed. This can lead to unnecessarily complex and often unnatural specifications, even beyond having to find a solution for making the higher-level protocol reentrant (cf. Section 3.1.4), if the non-responsiveness problem is actually taken into account rather than being ignored (which in turn would result in underspecified, and hence, unusable protocols, as discussed in Section 3.1.1).

We illustrate this by a joint state realization, which represents one form of a higher-level protocol: Consider a digital signature functionality \mathcal{F}_{sig} . Let us assume that \mathcal{F}_{sig} is specified in such a way that at the beginning it asks the adversary for signing and verification algorithms and keys before it answers other requests; as already mentioned, this is a very common design pattern. Because the adversary might not answer requests for the cryptographic material right away (non-responsiveness), \mathcal{F}_{sig} might receive further requests while waiting for the answer. Let us assume that \mathcal{F}_{sig} ignores/drops all such requests (this seems to be the option mainly used in the literature, see, e.g., [10, 94]).⁶

In a joint state realization of \mathcal{F}_{sig} , one instance of \mathcal{F}_{sig} (per party) is used to realize all sessions of \mathcal{F}_{sig} (for one party) in the ideal world. The idea behind the joint state realization is that if in session sid a message m is to be signed/verified, then one would instead sign/verify the message (sid, m) . In this way, messages of different sessions cannot interfere even when they are signed using the same instance of \mathcal{F}_{sig} and hence the same signing key. In the realization proof, a simulator would provide an instance \mathcal{F}_{sig} in session sid with a signing and verification algorithm that exactly mimics the behavior of the joint state realization in session sid (i.e., signing/verifying first prefix messages with sid). Unfortunately, because of the non-responsiveness problem, the joint state realization is more complex than that, even if, for the purpose of the discussion, we ignore the handling of corruption. To see this, assume that the environment sends a signing request for some message m in session sid . The joint state realization would now invoke its subroutine \mathcal{F}_{sig} with a signing request for (sid, m) . Before \mathcal{F}_{sig} can answer, \mathcal{F}_{sig} asks the adversary for the cryptographic material. Hence, the adversary/environment gets activated again, and the environment can send a new, say, signing request for message m' in session sid' . As \mathcal{F}_{sig} is still waiting for the adversary to provide the cryptographic material, this later request will be ignored by \mathcal{F}_{sig} and hence will never be answered. To mimic this behavior in the ideal world, the simulator should not provide the cryptographic material to the instance of \mathcal{F}_{sig} in session sid' (otherwise, \mathcal{F}_{sig} in session sid' would return a signature for m'). But then, this instance of \mathcal{F}_{sig} is blocked completely. Hence, in turn, the joint state realization also has to block all further requests for session sid' . That is, it has to store all SIDs for which it received requests while waiting for \mathcal{F}_{sig} to respond, and all future requests for all such SIDs have to be dropped.

This is very unnatural and certainly would not correspond to anything one would do in actual implementations: there one would simply prefix messages with SIDs, but one would never block requests for certain SIDs. This is just an artifact caused by the non-responsiveness problem, i.e., the fact that, in current models, urgent requests (in this case the request for cryptographic material by \mathcal{F}_{sig}) might not be answered immediately.

⁶As explained in Section 3.1.4, this approach, just as all other approaches discussed in Section 3.1.4, does not work in general, e.g., when the signer and verifier are independent and send urgent requests to the adversary upon first activation. It really depends on the details of both \mathcal{F}_{sig} and its intended realization.

3.2. Introducing Responsive Environments to the IITM Model

The non-responsiveness problem and the resulting complications shown in Section 3.1 are artificial problems. As urgent requests exist only for modeling purposes but do not model any real network traffic, a real adversary would not be able to use them to carry out attacks. Still, in all current universal composability models, the non-responsiveness of adversaries enables attacks that do not correspond to anything in reality. If we could force the adversary to answer urgent requests immediately, which, as already mentioned before, would be the natural and expected behavior, there would not be any need for coming up with workarounds that try to solve the non-responsiveness problem in the specifications of protocols and functionalities and one would not have to consider such artificial attacks in security proofs.

In this section, we propose our IITM model with responsive environments, which extends and generalizes the original IITM model from [99]. This extension allows protocol designers to specify messages that have to be answered immediately by (responsive) environments and adversaries, thus solving the non-responsiveness problem entirely. This section is structured as follows: We first give an overview of the general idea of responsive environments in Section 3.2.1. In Section 3.2.2 we recall several formal definitions from the original IITM model that we reuse in our extension, including formal definitions of elements that were only sketched in the high-level overview given in Chapter 2. In Sections 3.2.3 to 3.2.9 we define our model and prove two composition theorems for parallel composition of a constant number of systems as well as unbounded self-composition of multiple sessions of the same system. Finally, Section 3.2.10 discusses our extension. We note that, while our extension is defined and proven only for the IITM model, the general concepts of responsive environments are of independent interest and can be applied to other universal composability models as well. This is exemplified for the UC and GNUC models in Section 3.4.

3.2.1. Overview

To avoid the non-responsiveness problem altogether, we introduce the concepts of *responsive environments* and *responsive adversaries*. In a nutshell, when these environments and adversaries receive specific messages from the network (we call these messages *restricting*) then they have to respond to these messages immediately, i.e., without activating other parts of the protocol before sending an answer. Furthermore, depending on the restricting message, they may send an answer from a specific set of messages only. Restricting messages and possible answers are not hardwired into the model but can rather be specified by the protocol designer, which provides a high level of flexibility. More specifically, restricting messages and possible responses are specified by a binary relation $R \subseteq \{0, 1\}^+ \times \{0, 1\}^+$ over non-empty messages, called a *restriction*. If $(m, m') \in R$, then m is a restricting message and m' a possible answer to m . That is, if an environment/adversary receives m on its network interface, then it has to answer immediately with some m' such that $(m, m') \in R$.

This allows a protocol designer to specify all urgent requests as restricting messages by defining a restriction R appropriately; such requests are then answered not only immediately but also with an expected answer. Therefore the adversary can no longer interfere with the protocol run in an unintended way by activating other parts of the protocol or sending unexpected inputs before answering an urgent request.

Note that this concept is very powerful and needs to be handled with care: While, as motivated above, it does not weaken security results if one models urgent requests as restricting messages, one generally should not use such messages when modeling real network traffic, as real network messages are not guaranteed to be answered immediately in reality.

3.2.2. Recalling Notions from the IITM Model

Before we can extend the IITM model to work with responsive environments, we have to recall some parts of the original IITM model from [99]. We start by recalling the formal definitions of IITMs and systems of IITMs. Then we introduce several terms that describe specific properties of I/O and network interfaces of machines, such as, e.g., two machines being able to be connected to each other (in a system) via those interfaces. Finally, we recall the original runtime definitions of the IITM model in detail and then formally define environments, adversaries, and protocols.

Systems of IITMs

We start by recalling the definition of inexhaustible interactive Turing machines.

Definition 3.1 (Inexhaustible Interactive Turing machine (IITM)). *An (inexhaustible) interactive Turing machine (IITM) M is a probabilistic Turing machine with a polynomial q associated with it, where q bounds the runtime of M in mode `CheckAddress`, and the following tapes: a read-only tape on which the mode the IITM M is supposed to run is written (the mode tape) — the possible modes are `CheckAddress` and `Compute` — , a read-only tape on which the random coins are written (the random tape), a read-only tape on which the security parameter is written (the security parameter tape), a write-only tape (the address decision tape, where the output of mode `CheckAddress` is written), zero or more input and output tapes, and work tapes. The input and output tapes have names, and we require that different tapes of M have different names. We also require that only input tapes may be named `start` and only output tapes may be named `decision`.*

As already explained in Chapter 2, the `CheckAddress` mode is used as a generic mechanism for addressing specific instances (also called copies) of an IITM in a system of IITMs. In this mode, an IITM may perform, in every activation, a deterministic polynomial time computation in the length of the security parameter plus the length of the current input plus the length of its current configuration, where the polynomial is the one associated with the IITM. The IITM outputs “accept” or “reject” at the end of the computation in this mode, indicating whether the received message is processed further or ignored. The actual processing of the message, if

accepted, is then done in mode **Compute**. In mode **Compute**, a machine may only output at most one message on an output tape, which then ends its activation.

Input and output tapes, identified by their names, are classified as either I/O or network tapes. I/O tapes are used to model secure direct connections between (sub-)programs and network tapes are used to model insecure network connections that are controlled by a (network) adversary. For brevity, we define **NET** to be the set of names of all network tapes.

Several machines can connect to each other via their input and output tapes to form a system of machines, where instances of these machines can send messages to each other via those tapes. Two machines in a system are connected via a tape if one of them has an input tape named n and the other has an output tape named n . In a system, there are at most two tapes with the same name and if there are exactly two, they have opposite directions. This uniquely defines how machines connect to each other. More formally:

Definition 3.2 (Systems of IITMs). *A system of IITMs is a set of IITMs. An input/output tape with name n of one of the IITMs in the system is called internal if there is another machine in the same system with a tape named n of opposite direction. Input and output tapes that are not internal are called external. Using this terminology, a system is defined recursively as follows:*

- *Every single IITM M also is a system of IITMs.*
- *If \mathcal{Q} is a system, then $!\mathcal{Q}$ is also a system.*
- *For two systems of IITMs \mathcal{Q} and \mathcal{R} , the composition $\mathcal{Q}|\mathcal{R}$, where all internal tapes of \mathcal{Q} and \mathcal{R} are renamed such that they only connect via external tapes, is a system iff there is no external tape in both \mathcal{Q} and \mathcal{R} with the same name and direction.*

In the above definition, for $!\mathcal{Q}$, we say that \mathcal{Q} is in the scope of a bang. This means that every IITM that is contained in \mathcal{Q} will be allowed to have more than one instance in a run of the system \mathcal{Q} ; in contrast to this, IITMs which are not (in systems that are) in the scope of a bang will only have at most one instance in every run (see below). We note that the bang operator $!$ binds more strongly than the composition operator $|$, i.e., the systems $(!\mathcal{Q})|\mathcal{R}$ and $!\mathcal{Q}|\mathcal{R}$ are equivalent, whereas $!(\mathcal{Q}|\mathcal{R})$ is a different system. Also note that, by definition of the behavior of the bang operator (see below), we have that $!(\mathcal{Q}|\mathcal{R})$ is the same as $!\mathcal{Q}|\mathcal{R}$ and that $!(!\mathcal{Q})$ is the same as $!\mathcal{Q}$.

Running a System

In a run of a system \mathcal{Q} , several instances of each IITM in \mathcal{Q} may be spawned, where the **CheckAddress** mode is used to determine which instance gets to process an incoming message. The actual processing is then done in mode **Compute**. More specifically, in a run of a system $\mathcal{Q}(1^\eta, a)$ with security parameter η and external input a , only one instance is active at any time and all other instances wait for new input. The first machine to be activated is the IITM in \mathcal{Q} with an input tape named **start**, also called *master IITM*, by writing the external input a on

start (if no external input is considered, then the empty message is written on **start** instead); if no master IITM exists, the run of \mathcal{Q} terminates immediately. If a message m is written by some instance on one of its output tapes, say on t (initially, as mentioned, the external input is written on **start**), and there is a machine, say M , in \mathcal{Q} , with an input tape named t , then which instance of M gets to process m is decided as follows. All existing instances of M are run in **CheckAddress** mode in the order of their creation, until one instance accepts m . This instance (if any) then runs in **Compute** mode with input m written on its input tape t . If no instance accepted m and M is in the scope of a bang, or if there is no instance of M yet, a fresh instance of M is spawned and run in mode **CheckAddress** and if it accepts m , it gets to process m on its input tape t . Otherwise, the freshly created instance is deleted again, m is dropped, and the empty message is written on **start** in order to trigger the master IITM (of which there might also be several instances, where again their **CheckAddress** mode is used to decide which one gets to process the message). After running an instance in mode **CheckAddress**, the configuration is set back to the state before it was run in **CheckAddress**; thus this mode does not and cannot change the configuration of a machine.

When an instance of M processes a message in mode **Compute**, it may write at most one non-empty message, say m , on one of its output tapes, say t , and then stop. If there is an IITM with an input tape named t in the system, the message m is delivered to one instance of that IITM on tape t as described above. If the instance of M stops without producing output (sometimes also called empty output as all output tapes contain the empty message, i.e., $m = \epsilon$) or there is no IITM with an input tape t , then (an instance of) the master IITM is activated by writing the empty message on **start**. (Just as described above, the **CheckAddress** mode is used to determine the specific instance of the master IITM that processes the empty input message.) A run stops as soon as a non-empty message is written on **decision**, no master instance accepted the empty input message on **start**, or, in mode **Compute**, a master instance did not produce output. An informal example of a run was given in Chapter 2.

The overall output of a finite run is defined to be the message that is output on **decision**. If no message was written to **decision**, the overall output is the empty message. If a run does not terminate, the overall output is undefined. In slight abuse of notation, by writing $\mathcal{Q}(1^\eta, a)$ we denote both the run of the system \mathcal{Q} and its overall output distribution.

If two systems output 1 with the same probability (except for a negligible difference), these systems are called *indistinguishable*. The notion of indistinguishability is fundamental in universal composability models; in particular, it is used in the definition of the realization relation (cf. Definition 3.18) which states that two systems, one containing a (real) protocol and one an (ideal) protocol, must be indistinguishable. More formally:

Definition 3.3 (Negligible Functions With External Input, cf. Canetti [36]). *A function $f : \mathbb{N} \times \{0, 1\}^* \rightarrow \mathbb{R}_{\geq 0}$ is called negligible if for all $c, d \in \mathbb{N}$ there exists $\eta_0 \in \mathbb{N}$ such that for all $\eta > \eta_0$ and all $a \in \{0, 1\}^{\leq \eta^d}$: $f(\eta, a) < \eta^{-c}$. A function $f : \mathbb{N} \times \{0, 1\}^* \rightarrow [0, 1]$ is called overwhelming if $1 - f$ is negligible.*

Definition 3.4 (Equivalence/Indistinguishability of Systems). *Two systems \mathcal{Q} and \mathcal{R} are called equivalent or indistinguishable ($\mathcal{Q} \equiv \mathcal{R}$) if and only if there exists a negligible function $f : \mathbb{N} \times \{0, 1\}^* \rightarrow \mathbb{R}_{\geq 0}$ such that for every security parameter η and external input $a \in \{0, 1\}^*$ the following holds true:*

$$|\Pr[\mathcal{Q}(1^\eta, a) = 1] - \Pr[\mathcal{R}(1^\eta, a) = 1]| \leq f(\eta, a)$$

We also write $\mathcal{R} \equiv_f \mathcal{Q}$ to say that $|\Pr[\mathcal{Q}(1^\eta, a) = 1] - \Pr[\mathcal{R}(1^\eta, a) = 1]|$ is upper bounded by a specific (not necessarily negligible) function f .

Interfaces and Connectivity of Systems

For lemmas and theorems we often have to require specific properties of the external interfaces of some systems \mathcal{Q} and \mathcal{R} . This section recalls several terms that describe these properties.

We start by introducing a term which describes systems \mathcal{Q} and \mathcal{R} that can be connected to each other, i.e., the system $\mathcal{Q}|\mathcal{R}$ is well defined. However, merely considering the case of two systems is not sufficient: Observe that by Definition 3.2 the composition and hence the runtime behavior of three or more systems can depend on the order of the systems. This is the case, e.g., for systems \mathcal{A} and \mathcal{C} that have an external input tape named n and a system \mathcal{B} that has an external output tape named n . The systems $(\mathcal{A}|\mathcal{B})|\mathcal{C}$ and $\mathcal{A}|(\mathcal{B}|\mathcal{C})$ are well defined, but not identical. Hence, to avoid confusion, the following terminology also deals with the case of multiple systems:

Definition 3.5. *Two systems of IITMs \mathcal{Q} and \mathcal{R} are connectable iff each common external tape of \mathcal{Q} and \mathcal{R} has complementary directions (i.e., in one system the tape is an input tape, while in the other it is an output tape). We also say that \mathcal{Q} can be connected to \mathcal{R} (and vice-versa). The systems $\mathcal{Q}_1, \dots, \mathcal{Q}_n$ are connectable iff they are pairwise connectable.*

Note that, if the systems $\mathcal{Q}_1, \dots, \mathcal{Q}_n$ are connectable, then the system $\mathcal{Q}_1 | \dots | \mathcal{Q}_n$ is uniquely defined. In particular, for connectable systems, the composition operator $|$ is associative and commutative. In the following we will only use $|$ in this context, i.e., we will always require that the composition is uniquely defined. Sometimes we want to stress that two connectable systems connect only via their external I/O-tapes, but not via their external network tapes. We will use the term *I/O-connectable* to describe such systems.

Besides the term “connectable”, we also recall a term that describes systems with identical (external) tape interfaces. This property is usually needed when replacing one system \mathcal{Q} with another system \mathcal{R} , as this operation should not change the machines to which \mathcal{Q} and \mathcal{R} , respectively, can connect.

Definition 3.6. *Two systems of IITMs \mathcal{Q} and \mathcal{R} are compatible iff the set of (names of) external input tapes of \mathcal{Q} is the same as the set of (names of) external input tapes of \mathcal{R} , and the set of (names of) external output tapes of \mathcal{Q} is the same as the set of (names of) external output tapes of \mathcal{R} .*

Runtime Definitions

As already mentioned in Chapter 2, the IITM model, just as other UC-like models, ensures that systems run in polynomial time in the security parameter η and the length of the external input. This section recalls the formal runtime definitions of the IITM model.

The IITM model ensures this with the three runtime notions of *almost bounded*, *universally bounded*, and *environmentally bounded* systems. These definitions differ in the setting they consider: A system \mathcal{Q} is almost bounded if its runtime is bounded by a polynomial, it is universally bounded if its runtime is still bounded by a polynomial even when running with an arbitrary system \mathcal{R} , and it is environmentally bounded if the combined runtime of \mathcal{Q} and any universally bounded system \mathcal{R} is still bounded by a polynomial. Note that all of these notions consider runtime in mode **Compute** only; the **CheckAddress** mode is already bounded by the polynomial associated with the IITM. More formally:

Definition 3.7. A system \mathcal{Q} is called almost bounded if there exists a polynomial p such that

$$f(\eta, a) = \Pr[\text{Time}(\mathcal{Q}(1^\eta, a)) > p(\eta, |a|)] \text{ for all } \eta \in \mathbb{N} \text{ and } a \in \{0, 1\}^*$$

is negligible, where $\text{Time}(\mathcal{Q}(1^\eta, a))$ is the random variable which denotes the combined runtime of all instances of \mathcal{Q} (in a run of $\mathcal{Q}(1^\eta, a)$) in mode **Compute**.

Definition 3.8. A system \mathcal{E} is called universally bounded if there exists a polynomial p such that for every security parameter $\eta \in \mathbb{N}$, external input $a \in \{0, 1\}^*$, and any system \mathcal{Q} that can be connected to \mathcal{E} it holds true that the combined runtime in mode **Compute** of all instances of \mathcal{E} in every run of $(\mathcal{E} \mid \mathcal{Q})(1^\eta, a)$ is at most $p(\eta, |a|)$.

Definition 3.9. A system \mathcal{P} is called environmentally bounded if for every universally bounded \mathcal{E} that can be connected to \mathcal{P} the system $\mathcal{E} \mid \mathcal{P}$ is almost bounded.

Environments, Adversaries, Protocols

Recall from Chapter 2 that there are three special classes of systems, namely, environments, (network) adversaries, and protocols. This section precisely defines the properties of these systems. This includes both runtime and connectivity requirements: Environments are required to be universally bounded, i.e., they will not exceed a polynomial runtime no matter in which system they are used. Protocols and adversaries only have to be polynomial when running with an (universally bounded) environment and hence when receiving inputs of at most polynomial length. Furthermore, protocols and adversaries are not allowed to start or end a run; environments are expected to do this.

Definition 3.10. A system \mathcal{E} is called environmental system (or just environment) if it is universally bounded. For some system \mathcal{Q} , we denote by $\text{Env}(\mathcal{Q})$ the set of all environments \mathcal{E} that can be connected to \mathcal{Q} .

Definition 3.11. A system \mathcal{P} is called *protocol system* (or just *protocol*) if (i) no tape in \mathcal{P} is named `start` or `decision`, (ii) \mathcal{P} is environmentally bounded and (iii) for every IITM M occurring in \mathcal{P} such that M is not in the scope of a bang, we require that M accepts every incoming message in mode `CheckAddress`.⁷

Definition 3.12. A system \mathcal{A} is called *adversarial system* (or just *adversary*) if no tape of \mathcal{A} is named `start` or `decision`.⁸

3.2.3. Restricting Messages

We now formally define the term *restriction* from Section 3.2.1. Recall that a *restriction* is a set of pairs of messages. The first component of a pair is a message that, when sent on a network tape by a protocol, has to be answered “immediately”, i.e., an answer should be received on the corresponding network tape before any other machine of the protocol is activated on a different tape. The second component of a pair describes one possible answer. Additionally, we require that it is possible to efficiently decide whether an environment violates the restriction. More formally:

Definition 3.13 (Restriction). Let $R \subseteq \{0,1\}^+ \times \{0,1\}^+$ be a set of tuples of non-empty messages. We define $R[0] := \{m \mid \exists m' : (m, m') \in R\}$.

The set R is called a *restriction* if and only if the following holds true: There exists an algorithm A which for all inputs of the form (m, m') runs in at most polynomial time in the length of m' and outputs 1 iff $m \in R[0]$ but $(m, m') \notin R$, outputs 2 iff $(m, m') \in R$, and outputs 0 otherwise. A message $m \in R[0]$ is called a *restricting message*.

In the following, we will say that R is *decidable in polynomial time in the second component* or *efficiently decidable in the second component* when we refer to the property defined in the above definition. Note that this is different from the usual meaning of *decidable in polynomial time*: First, the algorithm A does not decide whether $(m, m') \in R$, but it decides whether, given some initial message m , a response m' is allowed by the restriction: If A outputs 0, the answer is allowed because $m \notin R[0]$; if it outputs 1, the answer is not allowed, and if it outputs 2, the answer is allowed (iff it is sent on the correct tape, i.e., the answer is sent to the correct machine; see below). Second, A does not run in polynomial time in the length of (m, m') , but only in the length of m' . This property is needed in the proof of Lemma 3.1. Technically, in Lemma 3.1, we can guarantee only that the length of the second component m' is polynomially bounded in the security parameter η and the external input a (if considered) as it is produced by the (universally bounded) environment; the first component m might not be as it is generated

⁷The third requirement is used in the proof of the unbounded self-composition theorem. It also makes intuitive sense: the only purpose of the `CheckAddress` mode is to allow for addressing multiple instances of the same machine. Hence, if there is just a single instance, there is no need to use the `CheckAddress` mode.

⁸Note that, just as for environmental systems and protocol systems, there are also runtime conditions imposed on adversarial systems. However, these conditions depend on the system that is connected to \mathcal{A} and thus will be presented in later definitions. Informally, if \mathcal{A} is connected to another (typically protocol) system \mathcal{Q} , then $\mathcal{A} \mid \mathcal{Q}$ will be required to be environmentally bounded.

by the protocol, which in Lemma 3.1 we do not require to be bounded in any way. Still the environment needs to be able to decide in polynomial time in η and a whether $m \in R[0]$ and whether $(m, m') \in R$. This property does not seem to be a strong requirement for practical purposes, as can be seen by the following example restriction:

$$\begin{aligned}
R := & \{(m, m') \mid m = (id, \text{AmICorrupted?}), \\
& \quad m' = (id, \text{CorruptionState}, b), \\
& \quad id \in \{0, 1\}^*, b \in \{\text{false}, \text{true}\}\} \\
& \cup \{(m, m') \mid m = (id, \text{Info}, m''), \\
& \quad m' = (id, \text{OK}), \\
& \quad id, m'' \in \{0, 1\}^*\} \\
& \cup \{(m, m') \mid m = (id, \text{Respond}, m''), \\
& \quad m' = (id, m'''), \\
& \quad id, m'', m''' \in \{0, 1\}^*\}
\end{aligned}$$

This restriction consists of three types of restricting messages, each identified by a fixed bit string (written in `typewriter front`). Each of the three types of restricting messages is prefixed with some identifier id , which has to be repeated in the answer. This can be used to ensure that the same instance which sent a message will also receive the answer, e.g., by defining mode `CheckAddress` such that every instance only accepts messages that are prefixed with its own (unique) ID. In fact, one of the main reasons we did not simply hardwire a specific restriction into our framework is that one can adjust restrictions to different definitions of the `CheckAddress` mode. In the above example, the first type of restricting messages can be used to ask the adversary for the corruption state of a fresh instance upon its first activation; the adversary then has to provide a bit b immediately. The second type can be used to provide the adversary with some information such that the adversary has to give back control immediately by sending an acknowledgement. The third type is meant to exchange arbitrary data between the protocol and the adversary, i.e., it could be used to ask the adversary for some initialization, such as keys and algorithms (the string m'' can be used to distinguish different requests), while the adversary can provide an arbitrary answer m''' but has to do so immediately. Jumping ahead, this third type is actually sufficient to build a so-called “generic restriction” that allows for modeling all cases in which restricting messages are needed (see Section 3.3). In particular, one can omit the first two types and only use the third type instead. We expect that restrictions will usually be variants of the generic restriction that have been adapted to the definition of the `CheckAddress` mode such that responses will be received by the correct machine instance.

The above relation R is indeed a restriction as defined in Definition 3.13 for an appropriate message encoding.⁹ Given some initial message m and an answer m' , it is possible to decide

⁹In particular, we need an encoding such that it is possible to check the number of components in a tuple and read single components without having to read the full string. Such an encoding can easily be established, so we will keep it implicit for simplicity of presentation.

in polynomial time (in the length of m') whether m' is an incorrect response to a restricting message. The algorithm A first checks whether m is a restricting message by checking the number of components in tuples and reading components with a fixed length. Note that this is possible in constant time, as we assumed an appropriate encoding. If $m \notin R[0]$, then A outputs 0. Otherwise, A reads m' and checks whether m' is a possible answer to m , i.e., whether $(m, m') \in R$. Depending on the result, A outputs either 1 or 2. It is easy to see that A can do this in polynomial time in the length of m' .

In the following, we assume that every IITM has a corresponding output tape for every input tape and vice versa (except for tapes named `start` and `decision`), i.e., we actually consider pairs of tapes. For an input/output tape $t \notin \{\text{start}, \text{decision}\}$ of a machine, we denote by t^{-1} the corresponding output/input tape of the same machine such that (t, t^{-1}) is a pair of tapes. We require that two machines connect via tape pairs only. That is, if (t, t^{-1}) is a tape pair of a machine M , with t being an input tape and t^{-1} the corresponding output tape, and M connects to other machines, then if M connects to an output (input) tape of another machine M' with t (t^{-1}), then M also connects to the corresponding input (output) tape of M' with t^{-1} (t). The intuition behind this is that machines, after having received some input from a machine M , usually send a response back to M at some point, so they need a corresponding output tape. In particular, there has to be a corresponding output tape for every input tape of the environment such that the environment is able to send responses to restricting messages to the correct machine.

Note that this is only a formal requirement, as every system \mathcal{Q} that does not fulfill this requirement can easily be adjusted by adding additional tapes. The behavior of \mathcal{Q} does not change as these new tapes will never be used.

3.2.4. Strong Simulatability for Responsive Environments

We now define *responsive environments*, *protocols for responsive environments*, and *responsive simulators*. We also define the security notion *strong simulatability for responsive environments*.

Definition 3.14 (Responsive environments). *Let R be a restriction. Let \mathcal{P} be a system of IITMs, and let $\mathcal{E} \in \text{Env}(\mathcal{P})$ be an environment. We define the event E to be the set of all runs of $\mathcal{E} \mid \mathcal{P}$ where the following holds true: If \mathcal{P} sends a restricting message $m \in R[0]$ on an external tape $t \in \text{NET}$ and if there exists a message m' such that m' is the first message received on an external tape t' (not necessarily in NET) of \mathcal{P} after m was sent, then $t' = t^{-1}$ and $(m, m') \in R$.*

The environment \mathcal{E} is called a responsive environment for \mathcal{P} (with respect to R) if and only if $\Pr[E]$ is overwhelming. We denote the set of all responsive environments for an IITM system \mathcal{P} by $\text{Env}_R(\mathcal{P})$.

Note that if a responsive environment receives a restricting message, it may still perform arbitrary computations and even send messages on tapes that are not connected to \mathcal{P} (e.g., `decision` or any internal tape). The environment is also free to not respond to \mathcal{P} at all and

instead to, e.g., end the run. Note also that this definition does not imply that \mathcal{E} is actually connected to the tape t on which \mathcal{P} sends a restricting message; it is entirely possible that a restricting message activates a master IITM with an empty string on tape `start`. Nevertheless, the definition guarantees that even in this case the next message sent to \mathcal{P} will be on the corresponding answer tape and will be of a form that satisfies R (except with negligible probability). Typically, however, \mathcal{E} will be connected to all external tapes of \mathcal{P} , where \mathcal{P} could be a real protocol or a simulator connected to an ideal protocol.

In our proofs, we will often say that “*in a run of the system $\mathcal{E} | \mathcal{P}$, restricting messages from \mathcal{P} are answered correctly*” to say that such a run belongs to the event E specified in Definition 3.14, i.e., after \mathcal{P} sends a restricting message there is either no answer or the answer is of an expected form on the correct tape. Analogously we say that “*in a run of the system $\mathcal{E} | \mathcal{P}$, a restricting message from \mathcal{P} is answered incorrectly*” to say that such a run does not belong to E , i.e., at least one restricting message was answered with an unexpected answer or on the wrong tape.

While responsive environments are defined with respect to a specific system \mathcal{P} , the following lemma shows that responsive environments are also responsive for systems that are indistinguishable from \mathcal{P} .

Lemma 3.1. *Let R be a restriction. Let \mathcal{P} and \mathcal{P}' be two systems of IITMs such that neither of them has a `start` or `decision` tape, both systems have the same external interface, and $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{P}'$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$. Then, $\text{Env}_R(\mathcal{P}) = \text{Env}_R(\mathcal{P}')$.¹⁰*

Proof. Let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$. We can assume that `start` is a tape of \mathcal{E} ; otherwise there would be no master IITM and every run would directly terminate with empty output, which implies $\mathcal{E} \in \text{Env}_R(\mathcal{P}')$. If $\mathcal{E} \notin \text{Env}_R(\mathcal{P}')$, then there must be a non-negligible set of runs of $\mathcal{E} | \mathcal{P}'$ in which a restricting message of \mathcal{P}' is not answered correctly. Let $\mathcal{E}' \in \text{Env}(\mathcal{P})$ be the single IITM that accepts all messages in mode `CheckAddress` and then simulates \mathcal{E} . Note that this is possible by Lemma 6 from [99], which states that we can simulate every system with a single IITM that has the same external interface and accepts all messages in mode `CheckAddress`. Furthermore, we add additional tapes c to \mathcal{E}' for each external network tape of \mathcal{P} that does not connect to \mathcal{E} . The system \mathcal{E}' also has a `decision` tape even if \mathcal{E} does not have one. Note that these additional tapes do not interfere with the simulation: If \mathcal{P} in a run of $\mathcal{E} | \mathcal{P}$ sends a message on an external network tape that is not connected to \mathcal{E} , the master IITM (which is part of \mathcal{E}) will be activated with empty input. The system \mathcal{E}' can easily simulate this as soon as a message on one of these new tapes is received.

Every time a message m' is about to be sent by \mathcal{E} to \mathcal{P} (or \mathcal{P}'), \mathcal{E}' checks whether the condition of responsive environments is about to be violated by this message and, if that is the case, outputs 1 on `decision` instead. Note that this is easy to check: First, \mathcal{E}' connects

¹⁰Note that this lemma does not impose any runtime requirements on the systems \mathcal{P} and \mathcal{P}' . We need such a general version of this lemma as often there are cases where we have constructed a new system \mathcal{P}' from a system \mathcal{P} and still have to prove that the runtime of \mathcal{P}' is bounded when running with a responsive environment. To prove this, we first need the results of this lemma.

to all external network tapes of \mathcal{P} , i.e. \mathcal{E}' “sees” all restricting messages m from \mathcal{P} . Second, because R is decidable in polynomial time in the second component, \mathcal{E}' can use algorithm A (see Definition 3.13) to decide in polynomial time in $|m'|$ whether m' is an allowed answer: If A outputs 0, m' is allowed because $m \notin R[0]$. If A outputs 1, m' is an incorrect answer since $m \in R[0] \wedge (m, m') \notin R$. If A outputs 2, m' is an allowed answer iff it is sent on the correct tape, because $m \in R[0] \wedge (m, m') \in R$. Because m' was outputted by the simulated \mathcal{E} and \mathcal{E} is universally bounded, $|m'|$ is bounded by a polynomial in the security parameter η and the length of the external input a (if any). So, overall A runs in polynomial time in η and $|a|$. If \mathcal{E} wants to output something on `decision` or the master IITM of \mathcal{E} stops with empty output, \mathcal{E}' outputs 0 on `decision` instead. Now it is easy to see that \mathcal{E}' is universally bounded as \mathcal{E}' only simulates the universally bounded system \mathcal{E} and additionally performs some checks which can be carried out in polynomial time. In particular, as \mathcal{E}' uses algorithm A to decide whether it must output 1, it does not have to read the entire messages m that are sent by \mathcal{P} but only polynomial many bits (as many as required by the polynomial time algorithm A and the universally bounded system \mathcal{E}).

Clearly, we have that $\mathcal{E}' \in \text{Env}_R(\mathcal{P})$ because \mathcal{E}' ends the run before a restricting message from \mathcal{P} is answered incorrectly. Furthermore, as every restricting message from \mathcal{P} (or \mathcal{P}') will activate the simulated \mathcal{E} (either on a connected tape or on tape `start`), it is easy to see that \mathcal{E}' will output 1 if and only if a restricting message from \mathcal{P} or \mathcal{P}' in the simulated run of $\mathcal{E} | \mathcal{P}$ or $\mathcal{E} | \mathcal{P}'$, respectively, is answered incorrectly. In other words, \mathcal{E}' is able to “see” all incorrect answers because \mathcal{E} must have been the system that sent the incorrect answer. Overall, because $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ and (by assumption from above) $\mathcal{E} \notin \text{Env}_R(\mathcal{P}')$, we have that $\mathcal{E}' | \mathcal{P} \not\equiv \mathcal{E}' | \mathcal{P}'$, in contradiction to the assumption that $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{P}'$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ and the fact that $\mathcal{E}' \in \text{Env}_R(\mathcal{P})$. We conclude that $\mathcal{E} \in \text{Env}_R(\mathcal{P}')$ and thus $\text{Env}_R(\mathcal{P}) \subseteq \text{Env}_R(\mathcal{P}')$.

The other inclusion follows by the same argument. Note in particular that \mathcal{E}' as constructed above is a responsive environment for \mathcal{P} also when simulating $\mathcal{E} \in \text{Env}_R(\mathcal{P}')$. \square

As we will consider only responsive environments in our extension of the IITM model, it is not necessary (and generally also not possible) to make any runtime guarantees for a system \mathcal{P} if it is combined with an arbitrary environment (i.e., any universally bounded system that can be connected to \mathcal{P}). Thus we need to relax the notion of environmentally bounded systems.

Definition 3.15 (R-environmentally bounded systems). *Let \mathcal{P} be a system of IITMs. Then \mathcal{P} is called environmentally bounded for responsive environments or simply R-environmentally bounded if and only if $\mathcal{E} | \mathcal{P}$ is almost bounded for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$.*

Definition 3.16 (Protocol systems for responsive environments). *Let R be a restriction. Let \mathcal{P} be a system of IITMs. Then \mathcal{P} is called a protocol system for responsive environments if and only if \mathcal{P} has no tapes named `start` or `decision`, \mathcal{P} is R-environmentally bounded, and every IITM in \mathcal{P} that is not in the scope of a bang accepts all messages in mode `CheckAddress`.*

The only difference between *protocol systems* and *protocol systems for responsive environments* is that the runtime condition of the latter is relaxed, i.e., the set of all protocol systems for responsive environments is a superset of the set of all protocol systems. From now on, we will use *protocol system* (or just *protocol*) to denote a protocol system for responsive environments as our model is based only on those systems.

Definition 3.17 (Responsive simulators). *Let R be a restriction. Let \mathcal{P}, \mathcal{F} be protocol systems. Let \mathcal{S} be an adversarial system such that \mathcal{S} can be connected to \mathcal{F} , the set of external tapes of \mathcal{S} is disjoint from the set of I/O-tapes of \mathcal{F} , $\mathcal{S} | \mathcal{F}$ and \mathcal{P} have the same external interface, and $\mathcal{S} | \mathcal{F}$ is R -environmentally bounded.*

Let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ be a responsive environment. We define the event E to be the set of all runs of $\mathcal{E} | \mathcal{S} | \mathcal{F}$ in which the following holds true: If \mathcal{F} sends a restricting message $m \in R[0]$ on an external tape $t \in \text{NET}$ and if there exists a message m' such that m' is the first message received on an external tape t' of \mathcal{F} after m was sent, then $t' = t^{-1}$ and $(m, m') \in R$.

The simulator \mathcal{S} is called a responsive simulator (with respect to R) if and only if $\Pr[E]$ is overwhelming for all responsive environments $\mathcal{E} \in \text{Env}_R(\mathcal{P})$. We denote the set of all responsive simulators for protocol systems \mathcal{P} and \mathcal{F} by $\text{Sim}_R^{\mathcal{P}}(\mathcal{F})$.

This definition ensures that restricting messages from \mathcal{F} are answered without activating another machine of \mathcal{F} (and with an expected response), even if \mathcal{F} is connected to a simulator (on its network interface). Analogously to responsive environments, the terms *restricting messages are answered correctly* and *a restricting message is answered incorrectly* can also be defined for responsive simulators.

In the above definition the event E which determines responsiveness of a simulator is defined for environments $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ (instead of $\mathcal{E} \in \text{Env}_R(\mathcal{S} | \mathcal{F})$). This follows the intuition that the simulator should work for every environment that is “valid” for the real protocol \mathcal{P} that is being analyzed. However, responsive simulators are used only in settings in which we have/require $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{S} | \mathcal{F}$ for every $\mathcal{E} \in \text{Env}_R(\mathcal{P})$. By Lemma 3.1, in these settings it holds true that $\text{Env}_R(\mathcal{P}) = \text{Env}_R(\mathcal{S} | \mathcal{F})$. Hence, defining responsive simulators for environments $\mathcal{E} \in \text{Env}_R(\mathcal{S} | \mathcal{F})$ actually yields an equivalent model, i.e., a protocol designer is free to quantify over either all environments from $\text{Env}_R(\mathcal{P})$ or all environments from $\text{Env}_R(\mathcal{S} | \mathcal{F})$ when proving responsiveness of a simulator.

Now we can define strong simulatability for responsive environments.

Definition 3.18 (Strong simulatability with responsive environments). *Let R be a restriction. Let \mathcal{P} and \mathcal{F} be protocol systems. Then, \mathcal{P} realizes \mathcal{F} with respect to responsive environments ($\mathcal{P} \leq_R \mathcal{F}$) if and only if there exists a responsive simulator $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$ such that $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{S} | \mathcal{F}$ for every responsive environment $\mathcal{E} \in \text{Env}_R(\mathcal{P})$.*

We show in Section 3.2.5 that the additional requirement of simulators being responsive is typically a property that is easy to see/prove. But first, we will prove two fundamental

properties of \leq_R , namely, reflexivity and transitivity. For this purpose, we have to recall two lemmas that were originally shown in [99]. We need to adjust one these lemmas to the case of responsive environments and show how the original proof has to be changed. We also prove two new lemmas that are based on those other lemmas.

In the following lemma, for an almost bounded system $\mathcal{R} \mid \mathcal{Q}$ where \mathcal{R} contains a master IITM, we consider an IITM, denoted by $[\mathcal{R}]_{\mathcal{Q}}$, which accepts all incoming messages (i.e., there is only ever at most one instance of $[\mathcal{R}]_{\mathcal{Q}}$) and internally simulates \mathcal{R} up to the polynomial runtime bound of $\mathcal{R} \mid \mathcal{Q}$. If the runtime bound is exceeded, $[\mathcal{R}]_{\mathcal{Q}}$ terminates the run with empty output. Note that $[\mathcal{R}]_{\mathcal{Q}}$ is universally bounded, i.e., an environment. The following lemma states that we can replace \mathcal{R} (in $\mathcal{R} \mid \mathcal{Q}$) with $[\mathcal{R}]_{\mathcal{Q}}$ such that the resulting system is indistinguishable from the original one.

Lemma 3.2 (Lemma 7 of [99]). *Let \mathcal{R} and \mathcal{Q} be connectable systems such that $\mathcal{R} \mid \mathcal{Q}$ is almost bounded and \mathbf{start} is a tape of \mathcal{R} (i.e. \mathcal{R} contains a master IITM, and hence, \mathcal{Q} does not). Then there exists an IITM $[\mathcal{R}]_{\mathcal{Q}}$ such that the following conditions are satisfied:*

1. $[\mathcal{R}]_{\mathcal{Q}}$ and \mathcal{R} are compatible.
2. $[\mathcal{R}]_{\mathcal{Q}}$ accepts every message in mode `CheckAddress`.
3. $[\mathcal{R}]_{\mathcal{Q}}$ is universally bounded
4. $[\mathcal{R}]_{\mathcal{Q}} \mid \mathcal{Q}$ is almost bounded
5. $[\mathcal{R}]_{\mathcal{Q}} \mid \mathcal{Q} \equiv \mathcal{R} \mid \mathcal{Q}$

Next, Lemma 3.3 states that $[\mathcal{E} \mid \mathcal{S}]_{\mathcal{F}}$ is a responsive environment for \mathcal{F} that simulates the behavior of $\mathcal{E} \mid \mathcal{S}$, where \mathcal{E} , \mathcal{S} , and \mathcal{F} are as in the definition of the realization relation (cf. Definition 3.18). We require simulators to be responsive to get this property. Without requiring simulators to be responsive, one could not move a simulator into the environment and still hope to obtain a responsive environment, a property which is important in many places, such as the transitivity of \leq_R and the composition theorems.

Lemma 3.3. *Let R be a restriction. Let \mathcal{P}, \mathcal{F} be protocol systems with $\mathcal{P} \leq_R \mathcal{F}$. Let $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$ be the responsive simulator that is used in the definition of $\mathcal{P} \leq_R \mathcal{F}$. Let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ be a responsive environment such that \mathbf{start} is a tape of \mathcal{E} . Then $[\mathcal{E} \mid \mathcal{S}]_{\mathcal{F}} \in \text{Env}_R(\mathcal{F})$.*

Proof. Let $\mathcal{P}, \mathcal{F}, \mathcal{S}, \mathcal{E}$ be systems as required by the lemma. First, we observe that Lemma 3.2 can be applied to $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$: As $\mathcal{S} \mid \mathcal{F}$ is R -environmentally bounded and $\mathcal{E} \in \text{Env}_R(\mathcal{S} \mid \mathcal{F})$ by Lemma 3.1, we have that $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ is almost bounded and $\mathcal{E} \mid \mathcal{S}$ contains a master IITM.

By Lemma 3.2, we know that $[\mathcal{E} \mid \mathcal{S}]_{\mathcal{F}}$ is universally bounded, and hence $[\mathcal{E} \mid \mathcal{S}]_{\mathcal{F}} \in \text{Env}(\mathcal{F})$. Suppose there was a non-negligible set of runs of $[\mathcal{E} \mid \mathcal{S}]_{\mathcal{F}} \mid \mathcal{F}$ where a restricting message of \mathcal{F} is not answered correctly. As $[\mathcal{E} \mid \mathcal{S}]_{\mathcal{F}} \mid \mathcal{F}$ and $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ behave identical except for a negligible

set of runs (in which a certain runtime is reached), it follows that there must also be a non-negligible set of runs of $\mathcal{E} | \mathcal{S} | \mathcal{F}$ in which a restricting message from \mathcal{F} is not answered correctly. However, this is a contradiction to $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$. This implies that restricting messages from \mathcal{F} are answered correctly in an overwhelming set of runs of $[\mathcal{E} | \mathcal{S}]_{\mathcal{F}} | \mathcal{F}$, i.e., $[\mathcal{E} | \mathcal{S}]_{\mathcal{F}} \in \text{Env}_R(\mathcal{F})$. \square

Note that Lemma 3.3 is essentially a special case of the following more general lemma.

Lemma 3.4. *Let R be a restriction and let \mathcal{R}, \mathcal{Q} be connectable systems of IITMs such that $\mathcal{R} | \mathcal{Q}$ is almost bounded and **start** is a tape of \mathcal{R} , then $[\mathcal{R}]_{\mathcal{Q}} \in \text{Env}_R(\mathcal{Q})$ if restricting messages from \mathcal{Q} are answered correctly in an overwhelming set of runs of $\mathcal{R} | \mathcal{Q}$.*

Proof. The proof is analogous to the proof of Lemma 3.3. \square

The following lemma corresponds to Lemma 8 in [99] but for responsive environments. It allows us to replace the simulation $[\mathcal{R}]_{\mathcal{Q}}$ by the original system \mathcal{R} when running with some system \mathcal{Q}' that is indistinguishable from \mathcal{Q} . This, again, is crucial for many proofs such as transitivity of the \leq_R relation and the composition theorems as we have to show statements for the original system \mathcal{R} , not for some (close) simulation of \mathcal{R} .

Lemma 3.5. *Let R be a restriction. Let \mathcal{R} and \mathcal{Q} be connectable systems such that $\mathcal{R} | \mathcal{Q}$ is almost bounded, **start** is a tape of \mathcal{R} (i.e., \mathcal{R} contains a master IITM, and hence \mathcal{Q} does not), and **decision** is not a tape of \mathcal{Q} . Furthermore, let \mathcal{Q}' be a system that is compatible with \mathcal{Q} and satisfies the following condition: $\mathcal{E} | \mathcal{Q} \equiv \mathcal{E} | \mathcal{Q}'$ for every $\mathcal{E} \in \text{Env}_R(\mathcal{Q})$ such that $\mathcal{E} | \mathcal{Q}$ is almost bounded. If $[\mathcal{R}]_{\mathcal{Q}} \in \text{Env}_R(\mathcal{Q})$, then*

$$[\mathcal{R}]_{\mathcal{Q}} | \mathcal{Q}' \equiv \mathcal{R} | \mathcal{Q}' .$$

Moreover, if $[\mathcal{R}]_{\mathcal{Q}} \in \text{Env}_R(\mathcal{Q})$ and $[\mathcal{R}]_{\mathcal{Q}} | \mathcal{Q}'$ is almost bounded, then also $\mathcal{R} | \mathcal{Q}'$ is almost bounded.

Proof. We may assume that \mathcal{R} is a single IITM that accepts every message in mode **CheckAddress** (by Lemma 6 from [99]). Now, recall that by definition, $[\mathcal{R}]_{\mathcal{Q}}$ exactly simulates all transitions of \mathcal{R} up to a certain polynomial bound and that this bound is exceeded only with negligible probability in runs of $[\mathcal{R}]_{\mathcal{Q}} | \mathcal{Q}$. If $[\mathcal{R}]_{\mathcal{Q}} \in \text{Env}_R(\mathcal{Q})$, then the probability that this bound is exceeded when running the system $[\mathcal{R}]_{\mathcal{Q}} | \mathcal{Q}'$ is also negligible. Otherwise, one could easily construct a system \mathcal{E} such that $\mathcal{E} \in \text{Env}_R(\mathcal{Q})$, $\mathcal{E} | \mathcal{Q}$ is almost bounded, and $\mathcal{E} | \mathcal{Q} \not\equiv \mathcal{E} | \mathcal{Q}'$, in contradiction to the assumption: The system \mathcal{E} is defined to simulate $[\mathcal{R}]_{\mathcal{Q}}$ and output 1 on **decision** if and only if the bound is exceeded (note that this is only possible because **decision** is not a tape of \mathcal{Q} or \mathcal{Q}'). As we assume $[\mathcal{R}]_{\mathcal{Q}} \in \text{Env}_R(\mathcal{Q})$, we directly obtain that $\mathcal{E} \in \text{Env}_R(\mathcal{Q})$. Furthermore, because $[\mathcal{R}]_{\mathcal{Q}} | \mathcal{Q}$ is almost bounded, it is easy to see that $\mathcal{E} | \mathcal{Q}$ is also almost bounded. Observe that 1 is output on **decision** in runs of $\mathcal{E} | \mathcal{Q}$ or $\mathcal{E} | \mathcal{Q}'$ if and only if the bound is exceeded.

It follows that with overwhelming probability $[\mathcal{R}]_{\mathcal{Q}}$ exactly simulates \mathcal{R} in the system $[\mathcal{R}]_{\mathcal{Q}} | \mathcal{Q}'$. Thus, we obtain $[\mathcal{R}]_{\mathcal{Q}} | \mathcal{Q}' \equiv \mathcal{R} | \mathcal{Q}'$. Since $[\mathcal{R}]_{\mathcal{Q}}$ exactly simulates \mathcal{R} in the system $[\mathcal{R}]_{\mathcal{Q}} | \mathcal{Q}'$ (except with negligible probability), it is also easy to see that if $[\mathcal{R}]_{\mathcal{Q}} | \mathcal{Q}'$ is almost bounded (and $[\mathcal{R}]_{\mathcal{Q}} \in \text{Env}_R(\mathcal{Q})$), then also $\mathcal{R} | \mathcal{Q}'$ is almost bounded. \square

We can now proof reflexivity and transitivity of \leq_R .

Lemma 3.6. \leq_R is reflexive and transitive.

Proof. Reflexivity: Let \mathcal{P} be a protocol system. Let \mathcal{S} be a single IITM without external tapes that does nothing. This directly implies that \mathcal{S} fulfills the conditions of responsive simulators regarding the interfaces and runtime. Let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ be a responsive environment. We have that the systems $\mathcal{E} | \mathcal{P}$ and $\mathcal{E} | \mathcal{S} | \mathcal{P}$ behave identical as \mathcal{S} cannot interact with the other machines, i.e., \mathcal{S} does not influence a run in any way. Thus we directly obtain $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{S} | \mathcal{P}$. Moreover, as $\mathcal{E} \in \text{Env}_R(\mathcal{P})$, we have that restricting messages sent by \mathcal{P} in runs of the system $\mathcal{E} | \mathcal{P}$ are answered correctly by \mathcal{E} . As $\mathcal{E} | \mathcal{S} | \mathcal{P}$ behaves identical to $\mathcal{E} | \mathcal{P}$, the same must also hold true for restricting messages of \mathcal{P} in runs of $\mathcal{E} | \mathcal{S} | \mathcal{P}$. Hence, we obtain $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{P})$. Overall, we have that $\mathcal{P} \leq_R \mathcal{P}$.

Transitivity: Let $\mathcal{P}, \mathcal{P}', \mathcal{P}''$ be protocol systems with $\mathcal{P} \leq_R \mathcal{P}'$ and $\mathcal{P}' \leq_R \mathcal{P}''$. Without loss of generality, we can assume that $\mathcal{P}, \mathcal{P}'$ and \mathcal{P}'' have pairwise disjoint sets of (external) network tapes. This is possible because, if a simulator exists for two systems that share some network tape names, we can rename the tapes of one system and define a new simulator that behaves just as the original one but in addition forwards messages between the renamed tapes and the original tapes. Let $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}'}(\mathcal{P}')$ and $\mathcal{S}' \in \text{Sim}_R^{\mathcal{P}''}(\mathcal{P}'')$ be the simulators that are used in the definition of $\mathcal{P} \leq_R \mathcal{P}'$ and $\mathcal{P}' \leq_R \mathcal{P}''$, respectively. Let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ be a responsive environment. We may assume that \mathcal{E} has a **start** tape; otherwise there would be no master IITM and the run would always directly terminate with empty output, in which case transitivity is trivially fulfilled. As \mathcal{E} contains a master IITM, we can use Lemma 3.2 to get $\mathcal{E} | \mathcal{S} | \mathcal{P}' \equiv [\mathcal{E} | \mathcal{S}]_{\mathcal{P}'} | \mathcal{P}'$. By Lemma 3.3, we have $[\mathcal{E} | \mathcal{S}]_{\mathcal{P}'} \in \text{Env}_R(\mathcal{P}')$ and thus $[\mathcal{E} | \mathcal{S}]_{\mathcal{P}'} | \mathcal{P}' \equiv [\mathcal{E} | \mathcal{S}]_{\mathcal{P}'} | \mathcal{S}' | \mathcal{P}''$. Lemma 3.5 implies $[\mathcal{E} | \mathcal{S}]_{\mathcal{P}'} | \mathcal{S}' | \mathcal{P}'' \equiv \mathcal{E} | \mathcal{S} | \mathcal{S}' | \mathcal{P}''$. Note that $\mathcal{E} | \mathcal{S} | \mathcal{S}' | \mathcal{P}''$ is well defined, i.e., all systems are actually connectable by the initial assumption that the sets of external network tapes of $\mathcal{P}, \mathcal{P}'$, and \mathcal{P}'' are pairwise disjoint. Overall, we have that $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{S} | \mathcal{S}' | \mathcal{P}''$.

It remains to show that $\mathcal{S} | \mathcal{S}' \in \text{Sim}_R^{\mathcal{P}''}(\mathcal{P}'')$. It is easy to see that the conditions regarding interfaces of responsive simulators are fulfilled. As $[\mathcal{E} | \mathcal{S}]_{\mathcal{P}'} \in \text{Env}_R(\mathcal{S}' | \mathcal{P}'')$ by Lemma 3.1 and by the definition of responsive simulators (cf. Definition 3.17), we have that $[\mathcal{E} | \mathcal{S}]_{\mathcal{P}'} | \mathcal{S}' | \mathcal{P}''$ is almost bounded, and thus by Lemma 3.5 the system $\mathcal{E} | \mathcal{S} | \mathcal{S}' | \mathcal{P}''$ is also almost bounded. This holds for every $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ and thus for every $\mathcal{E} \in \text{Env}_R(\mathcal{S} | \mathcal{S}' | \mathcal{P}'')$ by Lemma 3.1, so $\mathcal{S} | \mathcal{S}' | \mathcal{P}''$ is R-environmentally bounded. Let the event E be the set of all runs of $\mathcal{E} | \mathcal{S} | \mathcal{S}' | \mathcal{P}''$ in which a restricting message from \mathcal{P}'' is answered on the wrong tape or with an unexpected response. If $\Pr[E]$ is non-negligible, then there must also be a non-negligible set of runs of

$[\mathcal{E} | \mathcal{S}]_{\mathcal{P}'} | \mathcal{S}' | \mathcal{P}''$ in which a restricting message from \mathcal{P} is not answered correctly.¹¹ However, this is a contradiction to \mathcal{S}' being a responsive simulator, because $[\mathcal{E} | \mathcal{S}]_{\mathcal{P}'}$ is a responsive environment of \mathcal{P}' . Altogether, this implies $\mathcal{S} | \mathcal{S}' \in \text{Sim}_R^{\mathcal{P}}(\mathcal{P}'')$, and thus, $\mathcal{P} \leq_R \mathcal{P}''$. \square

3.2.5. Proving Responsiveness of Simulators

According to the definition of strong simulatability (Definition 3.18), for $\mathcal{P} \leq_R \mathcal{F}$ to hold one has to prove that there exists a simulator \mathcal{S} that fulfills i) certain restrictions concerning connectivity and interfaces, ii) certain runtime conditions, iii) the responsiveness condition (i.e., restricting messages from \mathcal{F} are answered correctly by \mathcal{S}), and most importantly iv) the equivalence $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{S} | \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$. Condition i) is easy to check; condition ii) is also generally easy to check, at least for natural protocols. The following lemma shows that iii) can be checked easily as well. According to the definition, one has to show that if \mathcal{S} receives a restricting message from \mathcal{F} , then \mathcal{S} has to make sure that this message is answered correctly according to the restriction R . So if \mathcal{S} sends out a message on an external tape, then there are two options: i) this message is sent to \mathcal{F} , and in this case, it has to be the correct response according to R , or ii) this message is sent to \mathcal{E} , but then this message has to be a restricting message for \mathcal{E} to ensure that \mathcal{E} answers directly (to \mathcal{S}). If \mathcal{S} sent a non-restricting message to \mathcal{E} , then \mathcal{E} would be free to send a message to \mathcal{F} , which would violate the responsiveness property that \mathcal{S} has to fulfill. Whether or not \mathcal{S} always sends such restricting messages (either to \mathcal{F} or to \mathcal{E}) should be easy to check given the specification of \mathcal{S} . Therefore, condition iii) should be easy to check. This intuition is made precise in the following lemma.

Lemma 3.7. *Let R be a restriction. Let \mathcal{P}, \mathcal{F} be protocol systems. Let \mathcal{S} be an adversarial system such that \mathcal{S} can be connected to \mathcal{F} , the set of external tapes of \mathcal{S} is disjoint from the set of I/O-tapes of \mathcal{F} , $\mathcal{S} | \mathcal{F}$ and \mathcal{P} have the same external interface, and $\mathcal{S} | \mathcal{F}$ is R -environmentally bounded.*

Let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ be a responsive environment. We define the event $E_{\mathcal{E}}$ to be the set of all runs of $\mathcal{E} | \mathcal{S} | \mathcal{F}$ in which the following holds true: The system \mathcal{S} accepts all restricting messages from \mathcal{F} in mode `CheckAddress`. If \mathcal{S} receives a restricting message from \mathcal{F} , it does not produce empty output until it has sent a message to \mathcal{F} . Furthermore, if \mathcal{S} sends a message on an external tape after having received a restricting message m on tape t from \mathcal{F} and before having sent an answer to \mathcal{F} , then the message is of one of two types: either a restricting message n on an external tape of $\mathcal{S} | \mathcal{F}$ (i.e. to the environment) where all possible answers n' with $(n, n') \in R$ will be accepted by \mathcal{S} in mode `CheckAddress`, or a message m' on tape t^{-1} with $(m, m') \in R$.

Then, $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$ and $\mathcal{P} \leq_R \mathcal{F}$ if, for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$, we have that (I) $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{S} | \mathcal{F}$ and (II) the event $E_{\mathcal{E}}$ is overwhelming.

Proof. Let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ be a responsive environment and let \mathcal{S} be given as above such that (I)

¹¹Per construction of $[\mathcal{E} | \mathcal{S}]_{\mathcal{P}'}$, it works just like $\mathcal{E} | \mathcal{S}$ except for cases where a certain runtime is reached. However, this runtime is only reached by $\mathcal{E} | \mathcal{S} | \mathcal{S}' | \mathcal{P}''$ in a negligible set of runs (see the proof of Lemma 3.5). As both systems only differ in a negligible set of runs, the statement follows.

and (II) are satisfied. To show $\mathcal{P} \leq_R \mathcal{F}$, we have to prove only that $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$. For this, it remains to show that \mathcal{S} fulfills the responsiveness condition, i.e., restricting messages from \mathcal{F} are answered correctly in an overwhelming set of runs of $\mathcal{E} | \mathcal{S} | \mathcal{F}$. First, by Lemma 3.1 we have $\mathcal{E} \in \text{Env}_R(\mathcal{S} | \mathcal{F})$. Let E' be the set of all runs of $\mathcal{E} | \mathcal{S} | \mathcal{F}$ in which a restricting message from \mathcal{F} is answered incorrectly, let E'' be the subset of E' in which the first restricting message from \mathcal{F} that is not answered correctly is sent on a network tape that is not connected to \mathcal{S} , and let E''' be the subset of E' where the first restricting message from \mathcal{F} that is not answered correctly is sent on a network tape that is connected to \mathcal{S} . Obviously, E'' and E''' are disjoint, $E' = E'' \cup E'''$, and hence, $\Pr[E'] = \Pr[E''] + \Pr[E''']$.

The event E'' is a subset of all runs of $\mathcal{E} | \mathcal{S} | \mathcal{F}$ in which a restricting message from $\mathcal{S} | \mathcal{F}$ is answered incorrectly. As $\mathcal{E} \in \text{Env}_R(\mathcal{S} | \mathcal{F})$, this set of runs is negligible, and thus, $\Pr[E'']$ must also be negligible.

The event E''' can be divided into two (distinct) subsets E'''_0 and E'''_1 , where we define E'''_0 to contain only those runs of E''' that belong to $E_{\mathcal{E}}$ (as defined in the lemma) and E'''_1 to contain the remaining runs of E''' . As, by assumption, $E_{\mathcal{E}}$ is overwhelming, we have that E'''_1 is negligible. Now, we further divide E'''_0 into two (distinct) subsets E'''_{0a} and E'''_{0b} , where we define E'''_{0a} to contain only those runs of E'''_0 in which restricting messages from \mathcal{S} on an external tape of $\mathcal{S} | \mathcal{F}$ are answered correctly and E'''_{0b} to contain the remaining runs of E'''_0 . Again, we directly obtain that E'''_{0b} is negligible because $\mathcal{E} \in \text{Env}_R(\mathcal{S} | \mathcal{F})$.

We now show that E'''_{0a} is the empty set, and hence, $\Pr[E'''_{0a}] = 0$. Assume that E'''_{0a} contains a run. Consider the first restricting message from \mathcal{F} sent to \mathcal{S} that is answered incorrectly. Such a message exists as $E'''_{0a} \subseteq E'''$. As $E'''_{0a} \subseteq E_{\mathcal{E}}$, \mathcal{S} accepts this message in mode `CheckAddress` and thus gets activated. As \mathcal{S} may not produce empty output, it can only end its activation by sending one of two kinds of messages. If it sends a restricting message to \mathcal{E} , the next message to $\mathcal{S} | \mathcal{F}$ will be on a tape of \mathcal{S} (otherwise the run would be in E'''_{0b}). As \mathcal{S} must accept this message in mode `CheckAddress` (by definition of $E_{\mathcal{E}}$), the system \mathcal{S} will get activated and, again, it may only end this activation by sending one of two kinds of messages. If \mathcal{S} sends a message to \mathcal{F} , it will be of such a form that the answer to the restricting message is correct; otherwise this run would be in E'''_1 . Hence, the restricting message from \mathcal{F} is actually answered correctly, a contradiction. This implies that $E'''_{0a} = \emptyset$.

Altogether, we have that $\Pr[E'] = \Pr[E''] + \Pr[E'''_1] + \Pr[E'''_{0a}] + \Pr[E'''_{0b}]$, where all probabilities on the right-hand side of the equation are negligible. Hence, the claim follows. \square

We note that \mathcal{S} might send several restricting messages to \mathcal{E} before it sends an answer to \mathcal{F} .

3.2.6. Security Notions

In the literature, besides strong simulatability, often the security notions UC, dummy UC, and black-box simulatability are used. In [99], it was shown that these notions are equivalent to strong simulatability within the IITM model, which justifies the use of the conceptually simpler notion of strong simulatability. Furthermore, in [99], equivalence to reactive simulatability, a

notion which is not currently used in the literature, was shown as well; however, this equivalence holds true only with respect to non-uniform environments. In this section, we show that these notions are still equivalent with respect to responsive environments. Hence, again, it suffices to consider strong simulatability only. First, we have to define the other security notions. For this purpose, we have to introduce responsive adversaries and the (responsive) dummy adversary.

Definition 3.19 (Responsive adversaries). *Let R be a restriction. Let \mathcal{P} be a protocol system. Let \mathcal{A} be an adversarial system such that \mathcal{A} can be connected to \mathcal{P} , the set of external tapes of \mathcal{A} is disjoint from the set of I/O-tapes of \mathcal{P} , and $\mathcal{A}|\mathcal{P}$ is R -environmentally bounded.*

Let $\mathcal{E} \in \text{Env}_R(\mathcal{A}|\mathcal{P})$ be a responsive environment. We define the event E to be the set of all runs of $\mathcal{E}|\mathcal{A}|\mathcal{P}$ in which the following holds true: If \mathcal{P} sends a restricting message $m \in R[0]$ on an external tape $t \in \text{NET}$ and if there exists a message m' such that m' is the first message received on an external tape t' of \mathcal{P} after m was sent, then $t' = t^{-1}$ and $(m, m') \in R$.

The adversarial system \mathcal{A} is called a responsive adversary (with respect to R) if and only if $\Pr[E]$ is overwhelming for all responsive environments $\mathcal{E} \in \text{Env}_R(\mathcal{A}|\mathcal{P})$. We denote the set of all responsive adversaries for a protocol system \mathcal{P} by $\text{Adv}_R(\mathcal{P})$.

Note that responsive adversaries are quite similar to responsive simulators as the two definitions mainly differ in the requirements for the external interface of $\mathcal{A}|\mathcal{P}$ and $\mathcal{S}|\mathcal{F}$, respectively. The idea behind this definition is that an adversary should also answer restricting messages from \mathcal{P} correctly, i.e., he should not be more powerful than a responsive environment.

In the following, we will also need a special responsive adversary: Given a protocol system \mathcal{P} , by $D_{\mathcal{P}}$ we denote the *dummy adversary for \mathcal{P}* that connects to all network tapes of \mathcal{P} and has one corresponding network tape per network tape of \mathcal{P} . The dummy adversary simply forwards all messages between a tape of \mathcal{P} and the corresponding tape. Note that the connectivity conditions of responsive adversaries are fulfilled and that $D_{\mathcal{P}}|\mathcal{P}$ is R -environmentally bounded because \mathcal{P} is R -environmentally bounded. Furthermore, it is easy to see that restricting messages of \mathcal{P} are answered correctly in an overwhelming set of runs because the dummy adversary forwards the same restricting message to a responsive environment on a network tape. Thus, we have $D_{\mathcal{P}} \in \text{Adv}_R(\mathcal{P})$.

Definition 3.20. *Let R be a restriction. Let \mathcal{P} and \mathcal{F} be protocol systems, where \mathcal{P} is supposed to realize \mathcal{F} .¹²*

1. Strong Simulatability with responsive environments (SS_R):

$$\mathcal{P} \leq_R^{\text{SS}} \mathcal{F} \text{ iff } \exists \mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F}) \ \forall \mathcal{E} \in \text{Env}_R(\mathcal{A}|\mathcal{P}):$$

$$\mathcal{E}|\mathcal{P} \equiv \mathcal{E}|\mathcal{S}|\mathcal{F} .$$

¹²Strong simulatability has already been defined in Definition 3.18. We here list strong simulatability again to have a complete list of the security notions we consider.

2. Universal Simulatability/Composability with responsive environments (UC_R):

$$\mathcal{P} \leq_R^{UC} \mathcal{F} \text{ iff } \forall \mathcal{A} \in \text{Adv}_R(\mathcal{P}) \exists \mathcal{I} \in \text{Sim}_R^{\mathcal{A}|\mathcal{P}}(\mathcal{F}) \forall \mathcal{E} \in \text{Env}_R(\mathcal{A}|\mathcal{P}):$$

$$\mathcal{E} | \mathcal{A} | \mathcal{P} \equiv \mathcal{E} | \mathcal{I} | \mathcal{F} .$$

3. Dummy Version of UC with responsive environments (dummy UC_R):

$$\mathcal{P} \leq_R^{dumUC} \mathcal{F} \text{ iff } \exists \mathcal{I} \in \text{Sim}_R^{D_{\mathcal{P}}|\mathcal{P}}(\mathcal{F}) \forall \mathcal{E} \in \text{Env}_R(D_{\mathcal{P}}|\mathcal{P}):$$

$$\mathcal{E} | D_{\mathcal{P}} | \mathcal{P} \equiv \mathcal{E} | \mathcal{I} | \mathcal{F} .$$

4. Black-box Simulatability with responsive environments (BB_R):

$$\mathcal{P} \leq_R^{BB} \mathcal{F} \text{ iff } \exists \mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F}) \forall \mathcal{A} \in \text{Adv}_R(\mathcal{P}) \forall \mathcal{E} \in \text{Env}_R(\mathcal{A}|\mathcal{P}):$$

$$\mathcal{E} | \mathcal{A} | \mathcal{P} \equiv \mathcal{E} | \mathcal{A} | \mathcal{S} | \mathcal{F} .$$

5. Reactive Simulatability with responsive environments (RS_R):

$$\mathcal{P} \leq_R^{RS} \mathcal{F} \text{ iff } \forall \mathcal{A} \in \text{Adv}_R(\mathcal{P}) \forall \mathcal{E} \in \text{Env}_R(\mathcal{A}|\mathcal{P}) \exists \mathcal{I} \in \text{Sim}_R^{\mathcal{A}|\mathcal{P}}(\mathcal{F}):$$

$$\mathcal{E} | \mathcal{A} | \mathcal{P} \equiv \mathcal{E} | \mathcal{I} | \mathcal{F} .$$

For strong and black-box simulatability, if \mathcal{P} and \mathcal{F} do not have disjoint network tapes, there might not exist a simulator such that \mathcal{P} and $\mathcal{S} | \mathcal{F}$ are indistinguishable because \mathcal{S} cannot connect to all network tapes of \mathcal{F} in this case. In what follows, we therefore always (implicitly) assume that the network tapes of \mathcal{F} are renamed first, so that the set of network tapes of \mathcal{P} and \mathcal{F} are disjoint. Note that this is only a formal requirement. It does not influence the behavior of \mathcal{P} and \mathcal{F} and therefore the security statement in any way.

We will now prove that strong simulatability with responsive environments is equivalent to the other security notions of Definition 3.20. We note that, except for the equivalence of strong simulatability and reactive simulatability, this theorem holds true also for uniform environments, i.e., environments that do not obtain external input. As mentioned before, an analogous result was originally proven in [99] for arbitrary environments.

Theorem 3.1. *Let R be a restriction. Let \mathcal{P} and \mathcal{F} be protocol systems. Then:*

$$\mathcal{P} \leq_R^{SS} \mathcal{F} \text{ iff } \mathcal{P} \leq_R^{UC} \mathcal{F} \text{ iff } \mathcal{P} \leq_R^{dumUC} \mathcal{F} \text{ iff } \mathcal{P} \leq_R^{BB} \mathcal{F} \text{ iff } \mathcal{P} \leq_R^{RS} \mathcal{F} .$$

Proof. Let \mathcal{P} and \mathcal{F} be protocol systems.

$\mathcal{P} \leq_R^{SS} \mathcal{F} \text{ iff } \mathcal{P} \leq_R^{UC} \mathcal{F} \text{ iff } \mathcal{P} \leq_R^{dumUC} \mathcal{F}$: We will first show the equivalence of SS_R , UC_R , and dummy UC_R . As UC_R trivially implies dummy UC_R , we only have to show that SS_R implies UC_R and that dummy UC_R implies SS_R .

Let $\mathcal{P} \leq_R^{SS} \mathcal{F}$. Then there exists a responsive simulator $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$ such that $\mathcal{E}|\mathcal{P} \equiv \mathcal{E}|\mathcal{S}|\mathcal{F}$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$. Let $\mathcal{A} \in \text{Adv}_R(\mathcal{P})$ be a responsive adversary and let $\mathcal{E} \in \text{Env}_R(\mathcal{A}|\mathcal{P})$ be a responsive environment. We define $\mathcal{I} := \mathcal{A}|\mathcal{S}$. We may assume that **start** is a tape of \mathcal{E} ; if this was not the case, then there would be no master IITM in $\mathcal{E}|\mathcal{A}|\mathcal{P}$ and $\mathcal{E}|\mathcal{I}|\mathcal{F}$, i.e. every run directly terminates with empty output, which trivially fulfills the following statements. As $\mathcal{A}|\mathcal{P}$ is R-environmentally bounded we have that $\mathcal{E}|\mathcal{A}|\mathcal{P}$ is almost bounded. Furthermore, by the definition of responsive adversaries, we know that restricting messages of \mathcal{P} are answered correctly in an overwhelming set of runs of $\mathcal{E}|\mathcal{A}|\mathcal{P}$. This gives the following equation:

$$\begin{aligned}
\mathcal{E}|\mathcal{A}|\mathcal{P} &\equiv [\mathcal{E}|\mathcal{A}]_{\mathcal{P}}|\mathcal{P} && \text{(Lemma 3.2)} \\
&\equiv [\mathcal{E}|\mathcal{A}]_{\mathcal{P}}|\mathcal{S}|\mathcal{F} && \text{(Lemma 3.4 and } \mathcal{P} \leq_R^{SS} \mathcal{F}\text{)} \\
&\equiv \mathcal{E}|\mathcal{A}|\mathcal{S}|\mathcal{F} && \text{(Lemma 3.5)} \\
&= \mathcal{E}|\mathcal{I}|\mathcal{F}
\end{aligned}$$

We also have that $[\mathcal{E}|\mathcal{A}]_{\mathcal{P}} \in \text{Env}_R(\mathcal{S}|\mathcal{F})$ by Lemma 3.1. As $\mathcal{S}|\mathcal{F}$ is R-environmentally bounded, we further obtain that $\mathcal{E}|\mathcal{A}|\mathcal{S}|\mathcal{F} = \mathcal{E}|\mathcal{I}|\mathcal{F}$ is almost bounded by Lemma 3.5. As this holds for all $\mathcal{E} \in \text{Env}_R(\mathcal{I}|\mathcal{F})$ by Lemma 3.1, we have that $\mathcal{I}|\mathcal{F}$ is R-environmentally bounded.

Now observe that there is only a negligible set of runs of $[\mathcal{E}|\mathcal{A}]_{\mathcal{P}}|\mathcal{S}|\mathcal{F}$ in which a restricting message from \mathcal{F} is answered incorrectly because \mathcal{S} is a responsive simulator. Because $[\mathcal{E}|\mathcal{A}]_{\mathcal{P}}|\mathcal{S}|\mathcal{F}$ and $\mathcal{E}|\mathcal{A}|\mathcal{S}|\mathcal{F}$ are the same except for a negligible set of runs (see the proof of Lemma 3.5), the set of runs of $\mathcal{E}|\mathcal{I}|\mathcal{F}$ in which a restricting message from \mathcal{F} is answered incorrectly must also be negligible. Overall, we have $\mathcal{I} \in \text{Sim}_R^{A|\mathcal{P}}(\mathcal{F})$, which shows that $\mathcal{P} \leq_R^{UC} \mathcal{F}$.

For the other implication, let $\mathcal{P} \leq_R^{dumUC} \mathcal{F}$. Then there exists a responsive simulator $\mathcal{I} \in \text{Sim}_R^{D_{\mathcal{P}}|\mathcal{P}}(\mathcal{F})$ such that $\mathcal{E}|D_{\mathcal{P}}|\mathcal{P} \equiv \mathcal{E}|\mathcal{I}|\mathcal{F}$ for all $\mathcal{E} \in \text{Env}_R(D_{\mathcal{P}}|\mathcal{P})$. Let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$. We can construct a new responsive environment $\mathcal{E}' \in \text{Env}_R(D_{\mathcal{P}}|\mathcal{P})$ that works exactly as \mathcal{E} , but for all network tapes c that connect between \mathcal{E} and \mathcal{P} , \mathcal{E}' has instead a network tape c' that connects to the tape of $D_{\mathcal{P}}$ that corresponds to c . Note that $\mathcal{E}|\mathcal{P}$ and $\mathcal{E}'|D_{\mathcal{P}}|\mathcal{P}$ work in exactly the same way because we can rename tapes and insert the dummy adversary without changing the behavior of the system by Lemma 4 and 5 from [99]. This implies $\mathcal{E}' \in \text{Env}_R(D_{\mathcal{P}}|\mathcal{P})$ as restricting messages from \mathcal{P} are answered correctly in an overwhelming set of runs of $\mathcal{E}|\mathcal{P}$ and thus restricting messages of $D_{\mathcal{P}}|\mathcal{P}$ must also be answered correctly in an overwhelming set of runs of $\mathcal{E}'|D_{\mathcal{P}}|\mathcal{P}$.

By assumption, there exists a responsive simulator \mathcal{I} such that $\mathcal{E}'|D_{\mathcal{P}}|\mathcal{P} \equiv \mathcal{E}'|\mathcal{I}|\mathcal{F}$. Analogous to the above, we can construct $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$, which works just as \mathcal{I} but for every external network tape c' that does not connect to \mathcal{F} , \mathcal{S} has instead a network tape c that corresponds

to c' .¹³ Now we have:

$$\begin{aligned}
\mathcal{E} \mid \mathcal{P} &\equiv \mathcal{E}' \mid D_{\mathcal{P}} \mid \mathcal{P} && \text{(Lemma 4 and 5 from [99])} \\
&\equiv \mathcal{E}' \mid \mathcal{I} \mid \mathcal{F} && (\mathcal{P} \leq_R^{dumUC} \mathcal{F}) \\
&\equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F} && \text{(Lemma 4 from [99])}
\end{aligned}$$

It also holds true that $\mathcal{S} \in \text{Sim}_R^{\mathcal{F}}(\mathcal{P})$: The conditions regarding external interfaces are obviously fulfilled. As $\mathcal{E}' \mid \mathcal{I} \mid \mathcal{F}$ and $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ are the same and $\mathcal{E}' \mid \mathcal{I} \mid \mathcal{F}$ is almost bounded, we have that $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ is almost bounded. This holds for all $\mathcal{E} \in \text{Env}_R(\mathcal{S} \mid \mathcal{F})$ by Lemma 3.1, and thus, $\mathcal{S} \mid \mathcal{F}$ is R-environmentally bounded. Finally, observe that restricting messages from \mathcal{F} are answered correctly in an overwhelming set of runs of $\mathcal{E}' \mid \mathcal{I} \mid \mathcal{F}$ because \mathcal{I} is a responsive simulator. This implies that restricting messages from \mathcal{F} are also answered correctly in an overwhelming set of runs of $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$, which shows $\mathcal{S} \in \text{Sim}_R^{\mathcal{F}}(\mathcal{P})$, and thus, $\mathcal{P} \leq_R^{SS} \mathcal{F}$.

$\mathcal{P} \leq_R^{SS} \mathcal{F}$ iff $\mathcal{P} \leq_R^{BB} \mathcal{F}$: We first show $SS_R \Rightarrow BB_R$. Let $\mathcal{P} \leq_R^{SS} \mathcal{F}$. Then there exists $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$ with $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$. Let $\mathcal{A} \in \text{Adv}_R(\mathcal{P})$ and $\mathcal{E} \in \text{Env}_R(\mathcal{A} \mid \mathcal{P})$. Analogously to the proof of “ $SS_R \Rightarrow UC_R$ ” we obtain that $\mathcal{E} \mid \mathcal{A} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{A} \mid \mathcal{S} \mid \mathcal{F}$, which directly implies $\mathcal{P} \leq_R^{BB} \mathcal{F}$ because $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$.

For the other implication let $\mathcal{P} \leq_R^{BB} \mathcal{F}$. Then there exists $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$ with $\mathcal{E} \mid \mathcal{A} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{A} \mid \mathcal{S} \mid \mathcal{F}$ for all $\mathcal{A} \in \text{Adv}_R(\mathcal{P})$ and $\mathcal{E} \in \text{Env}_R(\mathcal{A} \mid \mathcal{P})$. In the following let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$. Analogous to the proof of “dummy $UC_R \Rightarrow SS_R$ ”, we can construct a new environment $\mathcal{E}' \in \text{Env}_R(D_{\mathcal{P}} \mid \mathcal{P})$ by renaming some tapes of \mathcal{E} such that \mathcal{E}' connects to the dummy adversary $D_{\mathcal{P}}$ instead of the network tapes of \mathcal{P} . We obtain the following equation:

$$\begin{aligned}
\mathcal{E} \mid \mathcal{P} &\equiv \mathcal{E}' \mid D_{\mathcal{P}} \mid \mathcal{P} && \text{(Lemma 4 and 5 from [99])} \\
&\equiv \mathcal{E}' \mid D_{\mathcal{P}} \mid \mathcal{S} \mid \mathcal{F} && (D_{\mathcal{P}} \in \text{Adv}_R(\mathcal{P}) \text{ and } \mathcal{P} \leq_R^{BB} \mathcal{F}) \\
&\equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F} && \text{(Lemma 4 and 5 from [99])}
\end{aligned}$$

As $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$, we directly obtain $\mathcal{P} \leq_R^{SS} \mathcal{F}$.

$\mathcal{P} \leq_R^{SS} \mathcal{F}$ iff $\mathcal{P} \leq_R^{RS} \mathcal{F}$: Because dummy UC_R , UC_R and SS_R are equivalent and UC_R trivially implies RS_R , we have to show only that RS_R implies dummy UC_R . The main argument is similar to the one presented in [36].

Let $\mathcal{P} \leq_R^{RS} \mathcal{F}$, then for all $\mathcal{A} \in \text{Adv}_R(\mathcal{P})$ and all $\mathcal{E} \in \text{Env}_R(\mathcal{A} \mid \mathcal{P})$ there exists $\mathcal{I} \in \text{Sim}_R^{\mathcal{A} \mid \mathcal{P}}(\mathcal{F})$ such that

$$\mathcal{E} \mid \mathcal{A} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{I} \mid \mathcal{F} .$$

We choose $\mathcal{A} = D_{\mathcal{P}}$ to be the dummy adversary for \mathcal{P} . We also choose $\mathcal{E} \in \text{Env}_R(D_{\mathcal{P}} \mid \mathcal{P})$ to be a “universal” Turing machine (more precisely, a universal IITM) which takes as external

¹³With *corresponds* we mean that \mathcal{P} has a network tape c , which is mapped to a network tape c' by the dummy adversary.

input (i.e., input on `start`) a tuple of the form $(a, e, 1^t)$, where e is an encoding of some IITM (representing an environmental system \mathcal{E}'), a is interpreted to be external input to \mathcal{E}' , and t is interpreted to be the runtime (by Lemma 6 from [99], we may assume that e encodes a single IITM which accepts every message in mode `CheckAddress`). The universal IITM \mathcal{E} simulates \mathcal{E}' with external input a up to t steps. During this simulation, \mathcal{E} always checks whether \mathcal{E}' is about to send an incorrect answer to a restricting message of $D_{\mathcal{P}} | \mathcal{P}$ and, if that is the case, stops the run with empty output instead. This check is possible in polynomial time, because R is decidable in polynomial time in the second component (see the proof of Lemma 3.1 for a more detailed discussion). Clearly, \mathcal{E} is universally bounded because its runtime is polynomial in the security parameter plus the length of the external input. Furthermore, \mathcal{E} is responsive (for $D_{\mathcal{P}} | \mathcal{P}$) as it never answers a restricting message incorrectly. This implies that there exists $\mathcal{I}_{\mathcal{E}} \in \text{Sim}_R^{D_{\mathcal{P}} | \mathcal{P}}(\mathcal{F})$ such that $\mathcal{E} | D_{\mathcal{P}} | \mathcal{P} \equiv \mathcal{E} | \mathcal{I}_{\mathcal{E}} | \mathcal{F}$.

We observe that, given a security parameter η , external input a , and an environmental system \mathcal{E}' , there exists a tuple $(a, e, 1^t)$ of length polynomial in $\eta + |a|$ (it suffices to choose t polynomial in $\eta + |a|$ because \mathcal{E}' is universally bounded) such that \mathcal{E} with external input $(a, e, 1^t)$ precisely simulates \mathcal{E}' except for runs in which a restricting message is answered incorrectly by \mathcal{E}' . We will refer to this observation by (*).

Now let $\mathcal{E}' \in \text{Env}_R(D_{\mathcal{P}} | \mathcal{P})$ be a responsive environment with representation e . We first show that $\mathcal{E}' \in \text{Env}_R(\mathcal{I}_{\mathcal{E}} | \mathcal{F})$.¹⁴ For this, let $\mathcal{E}'' \in \text{Env}_R(D_{\mathcal{P}} | \mathcal{P})$ be the universally bounded IITM that simulates \mathcal{E}' , checks whether a restricting message is about to be answered incorrectly, and outputs 1 if this is about to happen. In all other cases 0 is output (again, see the proof of Lemma 3.1 which includes a more detailed construction of such an IITM). Let e' be the representation of \mathcal{E}'' . As \mathcal{E}'' never answers a responsive message incorrectly, we can use (*) to obtain that for all $\eta \in \mathbb{N}$ and $a \in \{0, 1\}^*$ there exists a tuple $(a, e', 1^t)$ such that \mathcal{E}'' is perfectly simulated by \mathcal{E} with external input $(a, e', 1^t)$. As $\mathcal{E} | D_{\mathcal{P}} | \mathcal{P} \equiv \mathcal{E} | \mathcal{I}_{\mathcal{E}} | \mathcal{F}$, i.e. both systems differ only with negligible probability, the length of $(a, e', 1^t)$ is polynomial in η and $|a|$, and \mathcal{E}'' outputs 1 with negligible probability in runs of $\mathcal{E}'' | D_{\mathcal{P}} | \mathcal{P}$, we obtain that \mathcal{E}'' also outputs 1 with negligible probability in runs of $\mathcal{E}'' | \mathcal{I}_{\mathcal{E}} | \mathcal{F}$. This implies that $\mathcal{E}' \in \text{Env}_R(\mathcal{I}_{\mathcal{E}} | \mathcal{F})$.

In the following, let g be the negligible function that bounds the probability of an incorrectly answered message of $D_{\mathcal{P}} | \mathcal{P}$ in runs of $\mathcal{E}' | D_{\mathcal{P}} | \mathcal{P}$ and let g' be the negligible function that bounds the probability of an incorrectly answered message of $\mathcal{I}_{\mathcal{E}} | \mathcal{F}$ in runs of $\mathcal{E}' | \mathcal{I}_{\mathcal{E}} | \mathcal{F}$. For all $\eta \in \mathbb{N}$ and $a \in \{0, 1\}^*$, let $(a, e, 1^t)$ be the tuple from (*) such that \mathcal{E} simulates \mathcal{E}' perfectly in runs in which all restricting messages are answered correctly. We obtain the following equation, where f is a negligible function:

¹⁴Observe that we cannot use Lemma 3.1 to show this. This is because the equation $\mathcal{E} | D_{\mathcal{P}} | \mathcal{P} \equiv \mathcal{E} | \mathcal{I}_{\mathcal{E}} | \mathcal{F}$ holds true only for a single environment \mathcal{E} , but the lemma requires this equation to hold true for all responsive environments $\mathcal{E} \in \text{Env}_R(D_{\mathcal{P}} | \mathcal{P})$.

$$\begin{aligned}
& |\Pr[\mathcal{E}' | D_{\mathcal{P}} | \mathcal{P}(1^\eta, a) = 1] - \Pr[\mathcal{E}' | I_{\mathcal{E}} | \mathcal{F}(1^\eta, a) = 1]| \\
& \leq |\Pr[\mathcal{E} | D_{\mathcal{P}} | \mathcal{P}(1^\eta, (a, e, 1^t)) = 1] - \Pr[\mathcal{E} | I_{\mathcal{E}} | \mathcal{F}(1^\eta, (a, e, 1^t)) = 1]| + g(\eta, a) + g'(\eta, a) \\
& \leq f(\eta, (a, e, 1^t)) + g(\eta, a) + g'(\eta, a)
\end{aligned}$$

As the length of $(a, e, 1^t)$ is polynomial in $\eta + |a|$, with e being a fixed bit string and t being a polynomial in η and a , we have that $f'(\eta, a) := f(\eta, (a, e, 1^t))$ is a negligible function. Now, because the sum of three negligible functions is still negligible, we have that there is a negligible function (in η and a) that bounds $|\Pr[\mathcal{E}' | D_{\mathcal{P}} | \mathcal{P}(1^\eta, a) = 1] - \Pr[\mathcal{E}' | I_{\mathcal{E}} | \mathcal{F}(1^\eta, a) = 1]|$. Hence, we have that $\mathcal{E}' | D_{\mathcal{P}} | \mathcal{P} \equiv \mathcal{E}' | I_{\mathcal{E}} | \mathcal{F}$ for all $\mathcal{E}' \in \text{Env}_R(D_{\mathcal{P}} | \mathcal{P})$, which together with $I_{\mathcal{E}} \in \text{Sim}_R^{D_{\mathcal{P}} | \mathcal{P}}(\mathcal{F})$ implies $\mathcal{P} \leq_R^{\text{dumUC}} \mathcal{F}$. \square

3.2.7. Parallel Composition of a Constant Number of Protocol Systems

In this section, we formally define and prove the composition theorem for a constant number of systems in our model. Analogously to [99], to prove this composition theorem we first prove a theorem that states that parallel composition of a constant number of systems holds true for the computational indistinguishability relation. This theorem corresponds to Theorem 4 in [99], which we adjust to the case of responsive environments. From this, we can then easily obtain the full composition theorem for strong simulatability.

Theorem 3.2. *Let R be a restriction. Let $k \geq 1$, $\mathcal{Q}, \mathcal{P}_1, \dots, \mathcal{P}_k, \mathcal{R}_1, \dots, \mathcal{R}_k$ be systems of IITMs without **start** and **decision** tapes such that the following conditions are satisfied:*

1. *For all $j \leq k$: \mathcal{P}_j and \mathcal{R}_j are R -environmentally bounded, \mathcal{P}_j and \mathcal{R}_j have the same (external) interface, and $\mathcal{E} | \mathcal{P}_j \equiv \mathcal{E} | \mathcal{R}_j$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P}_j)$.*
2. *$\mathcal{Q}, \mathcal{P}_1, \dots, \mathcal{P}_k$ are I/O-connectable (hence, $\mathcal{Q}, \mathcal{R}_1, \dots, \mathcal{R}_k$ are I/O-connectable), and $\mathcal{Q} | \mathcal{P}_1 | \dots | \mathcal{P}_k$ is R -environmentally bounded.*

Then, $\mathcal{E} | \mathcal{Q} | \mathcal{P}_1 | \dots | \mathcal{P}_k \equiv \mathcal{E} | \mathcal{Q} | \mathcal{R}_1 | \dots | \mathcal{R}_k$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{Q} | \mathcal{P}_1 | \dots | \mathcal{P}_k)$, and the system $\mathcal{Q} | \mathcal{R}_1 | \dots | \mathcal{R}_k$ is R -environmentally bounded.

Proof. The proof follows essentially the same argument as the original proof in [99], but uses responsive environments instead of general environments.

Let $\mathcal{Q}, \mathcal{P}_1, \dots, \mathcal{P}_k, \mathcal{R}_1, \dots, \mathcal{R}_k$ be systems of IITMs such that the requirements of the theorem are fulfilled. We will first prove the theorem for $k = 1$:

Let $\mathcal{E} \in \text{Env}_R(\mathcal{Q} | \mathcal{P}_1)$ be a responsive environment. We can assume that **start** is a tape of \mathcal{E} ; otherwise there is no master IITM in $\mathcal{E} | \mathcal{Q} | \mathcal{P}$, thus every run terminates directly with empty output, in which case the theorem holds true. If \mathcal{P}_1 sends a restricting message m on an external network tape t , then t is also an external network tape of $\mathcal{Q} | \mathcal{P}_1$ as \mathcal{Q} and \mathcal{P}_1 only connect via I/O tapes. Because \mathcal{E} is a responsive environment for $\mathcal{Q} | \mathcal{P}_1$, we have that the next message m' that

is sent to $\mathcal{Q} | \mathcal{P}_1$ will be on tape t^{-1} with $(m, m') \in R$ (except for a negligible set of runs). Using Lemma 3.2 and Lemma 3.4, we obtain $\mathcal{E} | \mathcal{Q} | \mathcal{P}_1 \equiv [\mathcal{E} | \mathcal{Q}]_{\mathcal{P}_1} | \mathcal{P}_1$, with $[\mathcal{E} | \mathcal{Q}]_{\mathcal{P}_1} \in \text{Env}_R(\mathcal{P}_1)$. Then $\mathcal{E} | \mathcal{P}_1 \equiv \mathcal{E} | \mathcal{R}_1$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P}_1)$ implies $[\mathcal{E} | \mathcal{Q}]_{\mathcal{P}_1} | \mathcal{P}_1 \equiv [\mathcal{E} | \mathcal{Q}]_{\mathcal{P}_1} | \mathcal{R}_1$. Using Lemma 3.5, we obtain $[\mathcal{E} | \mathcal{Q}]_{\mathcal{P}_1} | \mathcal{R}_1 \equiv \mathcal{E} | \mathcal{Q} | \mathcal{R}_1$. This implies that $\mathcal{E} | \mathcal{Q} | \mathcal{P}_1 \equiv \mathcal{E} | \mathcal{Q} | \mathcal{R}_1$.

We still have to prove that $\mathcal{Q} | \mathcal{R}_1$ is R-environmentally bounded: By Lemma 3.1 we know that $[\mathcal{E} | \mathcal{Q}]_{\mathcal{P}_1} \in \text{Env}_R(\mathcal{R}_1)$, and thus, because \mathcal{R}_1 is R-environmentally bounded, we know that $[\mathcal{E} | \mathcal{Q}]_{\mathcal{P}_1} | \mathcal{R}_1$ is almost bounded. By Lemma 3.5, we conclude that $\mathcal{E} | \mathcal{Q} | \mathcal{R}_1$ is almost bounded. As this holds true for all $\mathcal{E} \in \text{Env}_R(\mathcal{Q} | \mathcal{P}_1)$ and thus, by Lemma 3.1, for all $\mathcal{E} \in \text{Env}_R(\mathcal{Q} | \mathcal{R}_1)$, we have that $\mathcal{Q} | \mathcal{R}_1$ is R-environmentally bounded.

We now prove the theorem for $k > 1$. For every $r \leq k$, we define the r -th hybrid system:

$$\mathcal{H}_r := \mathcal{Q} | \mathcal{R}_1 | \cdots | \mathcal{R}_{r-1} | \mathcal{P}_{r+1} | \cdots | \mathcal{P}_k$$

which can be connected to \mathcal{P}_r or \mathcal{R}_r . Let $\mathcal{E} \in \text{Env}_R(\mathcal{Q} | \mathcal{P}_1 | \cdots | \mathcal{P}_k)$ be a responsive environment. By applying the case “ $k = 1$ ” k times, we obtain that:

$$\begin{aligned} & \mathcal{E} | \mathcal{Q} | \mathcal{P}_1 | \cdots | \mathcal{P}_k \\ &= \mathcal{E} | \mathcal{H}_1 | \mathcal{P}_1 && \text{(syntactic reordering of systems)} \\ &\equiv \mathcal{E} | \mathcal{H}_1 | \mathcal{R}_1 && \text{(case } k = 1 \text{ with } \mathcal{H}_1 \text{ playing the role of } \mathcal{Q}) \\ &= \mathcal{E} | \mathcal{H}_2 | \mathcal{P}_2 && \text{(syntactic reordering of systems)} \\ &\equiv \mathcal{E} | \mathcal{H}_2 | \mathcal{R}_2 && \text{(Lemma 3.1 and case } k = 1 \text{ with } \mathcal{H}_2 \text{ playing the role of } \mathcal{Q}) \\ & \vdots \\ &= \mathcal{E} | \mathcal{H}_k | \mathcal{P}_k && \text{(syntactic reordering of systems)} \\ &\equiv \mathcal{E} | \mathcal{H}_k | \mathcal{R}_k && \text{(Lemma 3.1 and case } k = 1 \text{ with } \mathcal{H}_k \text{ playing the role of } \mathcal{Q}) \\ &= \mathcal{E} | \mathcal{Q} | \mathcal{R}_1 | \cdots | \mathcal{R}_k \end{aligned}$$

In the above argument, we need that $\mathcal{H}_r | \mathcal{P}_r$ is R-environmentally bounded. This is the case: $\mathcal{Q} | \mathcal{P}_1 | \cdots | \mathcal{P}_k = \mathcal{H}_1 | \mathcal{P}_1$ is R-environmentally bounded by assumption. But then $\mathcal{H}_1 | \mathcal{R}_1$ is also R-environmentally bounded because the theorem holds true for “ $k = 1$ ”. Iterating this argument proves the claim; in particular this shows that $\mathcal{Q} | \mathcal{R}_1 | \cdots | \mathcal{R}_k$ is R-environmentally bounded, which is the second part of the theorem. \square

Using the above theorem, we can now prove the composition theorem for a constant number of protocol systems. This corresponds to Theorem 8 in [99] (cf. Theorem 2.1 in Chapter 2).

Theorem 3.3 (Composition Theorem for a Constant Number of Protocol Systems). *Let R be a restriction. Let $k \geq 1$, \mathcal{Q} be a system of IITMs without **start** and **decision** tape, and $\mathcal{P}_1, \dots, \mathcal{P}_k, \mathcal{F}_1, \dots, \mathcal{F}_k$ be protocol systems such that all systems have pairwise disjoint sets of network tapes and the following conditions are satisfied:*

1. For all $j \leq k$: $\mathcal{P}_j \leq_R \mathcal{F}_j$

2. $\mathcal{Q}, \mathcal{P}_1, \dots, \mathcal{P}_k$ are I/O-connectable, and $\mathcal{Q} | \mathcal{P}_1 | \dots | \mathcal{P}_k$ is R-environmentally bounded.

Then, $\mathcal{Q} | \mathcal{P}_1 | \dots | \mathcal{P}_k \leq_R \mathcal{Q} | \mathcal{F}_1 | \dots | \mathcal{F}_k$

Proof. Let $\mathcal{Q}, \mathcal{P}_1, \dots, \mathcal{P}_k, \mathcal{F}_1, \dots, \mathcal{F}_k$ be as assumed in the theorem. For all $j \leq k$, let $\mathcal{S}_j \in \text{Sim}_R^{\mathcal{P}_j}(\mathcal{F}_j)$ be a responsive simulator such that $\mathcal{E} | \mathcal{P}_j \equiv \mathcal{E} | \mathcal{S}_j | \mathcal{F}_j$ for all responsive environments $\mathcal{E} \in \text{Env}_R(\mathcal{P}_j)$. Note that, because $\mathcal{Q}, \mathcal{P}_1, \dots, \mathcal{P}_k, \mathcal{F}_1, \dots, \mathcal{F}_k$ have disjoint sets of network tapes, the systems $\mathcal{Q}, \mathcal{S}_1, \mathcal{F}_1, \dots, \mathcal{S}_k, \mathcal{F}_k$ are connectable.

By applying Theorem 3.2 (with $\mathcal{R}_j := \mathcal{S}_j | \mathcal{F}_j$ for $j \leq k$), we obtain that $\mathcal{E} | \mathcal{Q} | \mathcal{P}_1 | \dots | \mathcal{P}_k \equiv \mathcal{E} | \mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{Q} | \mathcal{P}_1 | \dots | \mathcal{P}_k)$. We also obtain that the system $\mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k$ is R-environmentally bounded.

We still have to show that $\mathcal{S}_1 | \dots | \mathcal{S}_k \in \text{Sim}_R^{\mathcal{Q} | \mathcal{P}_1 | \dots | \mathcal{P}_k}(\mathcal{Q} | \mathcal{F}_1 | \dots | \mathcal{F}_k)$. It is easy to see that all conditions regarding interfaces of responsive simulators are fulfilled. We have also just proven the runtime condition for $\mathcal{S}_1 | \dots | \mathcal{S}_k$. So we only have to show that restricting messages from $\mathcal{Q} | \mathcal{F}_1 | \dots | \mathcal{F}_k$ are answered correctly in an overwhelming set of runs. Let $\mathcal{E} \in \text{Env}_R(\mathcal{Q} | \mathcal{P}_1 | \dots | \mathcal{P}_k)$ be a responsive environment. By Lemma 3.1, we know that $\mathcal{E} \in \text{Env}_R(\mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k)$. We may assume that **start** is a tape of \mathcal{E} ; otherwise there is no master IITM in $\mathcal{E} | \mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k$ and every run directly terminates with empty output. As in such a run no messages are sent, there also is no restricting message that is answered incorrectly and the claim follows.

Let the event E be the set of all runs of $\mathcal{E} | \mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k$ in which a restricting message from $\mathcal{Q} | \mathcal{F}_1 | \dots | \mathcal{F}_k$ is answered incorrectly, let E' be the set of all runs of $\mathcal{E} | \mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k$ in which a restricting message from \mathcal{Q} is answered incorrectly, and let E_j for $j \leq k$ be the set of all runs of $\mathcal{E} | \mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k$ where a restricting message from \mathcal{F}_j is answered incorrectly. Note that $\mathcal{Q}, \mathcal{F}_1, \dots, \mathcal{F}_k$ have disjoint network tapes, so if one of those systems sends a restricting message on a network tape, then this message is sent on an external network tape of $\mathcal{Q} | \mathcal{F}_1 | \dots | \mathcal{F}_k$. In particular, we have that $E = E' \cup \bigcup_{j \leq k} E_j$, and thus $\Pr[E] \leq \Pr[E'] + \sum_{j \leq k} \Pr[E_j]$.

As every incorrectly answered restricting message from \mathcal{Q} is sent on an external network tape of $\mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k$ (recall that network tapes of $\mathcal{Q}, \mathcal{P}_j$ and \mathcal{F}_j are disjoint for all $j \leq k$) and as $\mathcal{E} \in \text{Env}_R(\mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k)$, we directly obtain that E' is negligible.

Now, let us consider the event E_j for some $j \leq k$. For brevity of presentation, we define the system $\mathcal{H} := \mathcal{E} | \mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_{j-1} | \mathcal{F}_{j-1} | \mathcal{S}_{j+1} | \mathcal{F}_{j+1} | \dots | \mathcal{S}_k | \mathcal{F}_k$, i.e., \mathcal{H} does not contain \mathcal{S}_j and \mathcal{F}_j . We will construct a responsive environment for $\mathcal{S}_j | \mathcal{F}_j$: First we observe that if the system $\mathcal{S}_j | \mathcal{F}_j$ sends a restricting message on an external network tape (in a run of $\mathcal{E} | \mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k$), the restricting message is sent on an external network tape of $\mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k$ because the systems $\mathcal{Q}, \mathcal{S}_1 | \mathcal{F}_1, \dots, \mathcal{S}_k | \mathcal{F}_k$ have disjoint network tapes. Because of $\mathcal{E} \in \text{Env}_R(\mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k)$, those messages are answered correctly in an overwhelming set of runs; we will refer to this observation by (*) in what follows. As the system $\mathcal{E} | \mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k$ is almost bounded and \mathcal{E} contains a master

IITM, we can use Lemma 3.2, and because of (*), we can also use Lemma 3.4 to obtain $\mathcal{E} | \mathcal{Q} | \mathcal{S}_1 | \mathcal{F}_1 | \dots | \mathcal{S}_k | \mathcal{F}_k = \mathcal{H} | \mathcal{S}_j | \mathcal{F}_j \equiv [\mathcal{H}]_{\mathcal{S}_j | \mathcal{F}_j} | \mathcal{S}_j | \mathcal{F}_j$, with $[\mathcal{H}]_{\mathcal{S}_j | \mathcal{F}_j} \in \text{Env}_R(\mathcal{S}_i | \mathcal{F}_i)$. Furthermore, by Lemma 3.1, we also have that $[\mathcal{H}]_{\mathcal{S}_j | \mathcal{F}_j} \in \text{Env}_R(\mathcal{P}_j)$.

Because the system \mathcal{S}_j is a responsive simulator, we know that restricting messages from \mathcal{F}_j must be answered correctly in an overwhelming set of runs of $[\mathcal{H}]_{\mathcal{S}_j | \mathcal{F}_j} | \mathcal{S}_j | \mathcal{F}_j$. But then restricting messages from \mathcal{F} in $\mathcal{H} | \mathcal{S}_j | \mathcal{F}_j$ must also be answered correctly in an overwhelming set of runs because $\mathcal{H} | \mathcal{S}_j | \mathcal{F}_j$ and $[\mathcal{H}]_{\mathcal{S}_j | \mathcal{F}_j} | \mathcal{S}_j | \mathcal{F}_j$ are identical in an overwhelming set of runs. This implies that the event E_j is negligible.

Overall, we have that $\Pr[E] \leq \Pr[E'] + \sum_{j \leq k} \Pr[E_j]$, where the right side is a sum of $k + 1$ negligible functions, and hence $\Pr[E]$ is negligible. \square

3.2.8. Unbounded Self-Composition

In this section, we formally define and prove the composition theorem for unbounded self-composition in our model. Recall from Chapter 2 that this composition theorem says that it suffices to prove that a single session of a (real) protocol \mathcal{P} realizes a single session of an (ideal) protocol \mathcal{F} in order to conclude that multiple sessions of the real protocol realize multiple sessions of the ideal protocol. This holds true as long as \mathcal{P} and \mathcal{F} have disjoint sessions, i.e., sessions do not share any state and do not directly interact with each other.

For this purpose, we first recall and adapt the definitions of the *protocol session identifier (PSID) functions* and *σ -session versions* from [99].¹⁵ As explained in Chapter 2, these notions formalize the intuition of a protocol having disjoint sessions. In particular, they allow for assigning every instance of an IITM to a single protocol session (identified by a unique PSID), where instances from different sessions do not interact with each other. We then also recall and adapt the definitions of *σ -single session environments and simulators* from [99], where such environments are allowed to invoke only a single session of a connected protocol. Finally, we state and prove a version of the composition theorem for unbounded self-composition from [99] (see Theorem 10 in that work) that is adapted for responsive environments.

As mentioned, we start with the definition of PSID functions and σ -session versions.

Definition 3.21 (Protocol Session Identifier (PSID) Function). *A function $\sigma : \{0, 1\}^* \times \mathcal{T} \rightarrow \{0, 1\}^* \cup \{\perp\}$ (where \mathcal{T} is a set of tapes names) is called a protocol session identifier (PSID) function if it is computable in polynomial time (in the length of its input).*

Definition 3.22 (σ -Session Versions). *Let σ be a PSID function and let M be an IITM such that σ is defined for all (names of) tapes of M . Then, M is a σ -session machine (also called a σ -session version) if for every system \mathcal{Q} such that \mathcal{Q} and M are connectable the following conditions are satisfied for every η , a , and every run ρ of $(\mathcal{Q} | M)(1^\eta, a)$:*

¹⁵As already mentioned in Chapter 2, these functions were just called *session identifier (SID) functions* in [99]; the same name was also used in the original publication of our IITM model with responsive environments in [29]. We have renamed these functions to PSID functions in this thesis to avoid any confusion with the concept of session identifiers (SIDs) introduced in the iUC framework in Chapter 4.

1. Whenever M is activated in ρ in mode **CheckAddress** with an input message m on tape c , M rejects m if $\sigma(m, c) = \perp$.
2. If the first input message that M accepted in ρ in mode **CheckAddress** is m_0 on tape c_0 and (later) M is activated in mode **CheckAddress** in ρ with an input message m on tape c , then M rejects m if $\sigma(m, c) \neq \sigma(m_0, c_0)$.
3. Whenever M outputs a non-empty message m on tape c in ρ in mode **Compute**, then $\sigma(m, c) = \sigma(m_0, c_0)$ (where the first accepted message was m_0 on tape c_0 , see above).

A system \mathcal{R} is called a σ -session system/version if every IITM occurring in \mathcal{R} is a σ -session version.

We will also use a stronger variant of σ -session versions. An IITM M is called σ -complete if it fulfills the conditions stated in Definition 3.22, but with Condition 2 replaced by the following stronger condition: If the first input message that M accepted in ρ in mode **CheckAddress** is m_0 on tape c_0 and (later) M is activated in mode **CheckAddress** in ρ with an input message m on tape c , then M accepts m iff $\sigma(m, c) = \sigma(m_0, c_0)$. In other words, σ exactly determines those messages accepted by M in mode **CheckAddress**.

Definition 3.23 (σ -Single Session Responsive Environment). *Let R be a restriction. Let σ be a PSID function, let \mathcal{P} be a system of IITMs, and let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ be a responsive environment for \mathcal{P} . The system \mathcal{E} is called a σ -single session responsive environment if for every system \mathcal{Q} such that \mathcal{E} and \mathcal{Q} are connectable the following holds true for every η, a and in every run ρ of $(\mathcal{E} \mid \mathcal{Q})(1^\eta, a)$:*

Let $m_0 \neq \epsilon$ (where ϵ is the empty bit string) be the first message output by \mathcal{E} on some external tape $c_0 \neq \text{decision}$ in ρ . Then $\sigma(m_0, c_0) \neq \perp$, and every message $m \neq \epsilon$ output by \mathcal{E} on an external tape $c \neq \text{decision}$ in ρ satisfies $\sigma(m, c) = \sigma(m_0, c_0)$.

We use $\text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$ to denote the set of all σ -single session responsive environments for a system \mathcal{P} .

Definition 3.24 (σ -Single Session Responsive Simulator). *Let R be a restriction. Let σ be a PSID function, and let \mathcal{P}, \mathcal{F} be protocol systems. Let \mathcal{S} be an adversarial system such that \mathcal{S} can be connected to \mathcal{F} , the set of external tapes of \mathcal{S} is disjoint from the set of I/O-tapes of \mathcal{F} , $\mathcal{S} \mid \mathcal{F}$ and \mathcal{P} have the same external interface, and $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ is almost bounded for all \mathcal{E} in $\text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$.*

Let $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$ be a σ -single session responsive environment. Define the event E to be the set of all runs of $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ in which the following holds true: If \mathcal{F} sends a restricting message $m \in R[0]$ on an external tape $t \in \text{NET}$ and if there exists a message m' such that m' is the first message received on an external tape t' of \mathcal{F} after sending m , then $t' = t^{-1}$ and $(m, m') \in R$.

The simulator \mathcal{S} is called a σ -single session responsive simulator (with respect to σ and R) if and only if $\Pr[E]$ is overwhelming for all σ -single session responsive environments $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$.

$\text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$. We denote the set of all σ -single session responsive simulators for protocol systems \mathcal{P} and \mathcal{F} by $\text{Sim}_{R,\sigma\text{-single}}^{\mathcal{P}}(\mathcal{F})$.

Note that every responsive simulator also is a σ -single session responsive simulator, because the definition of the latter is a relaxed version of the former (i.e. the runtime condition and the event E are defined only for a subset of all responsive environments).

Definition 3.25 (σ -Single Session Realization). *Let R be a restriction. Let σ be a PSID function, and let \mathcal{P} and \mathcal{F} be protocol systems such that \mathcal{P} and \mathcal{F} are σ -session versions. Then, \mathcal{P} single-session realizes \mathcal{F} w.r.t. σ and R ($\mathcal{P} \leq_{R,\sigma\text{-single}} \mathcal{F}$) if and only if there exists $\mathcal{S} \in \text{Sim}_{R,\sigma\text{-single}}^{\mathcal{P}}(\mathcal{F})$ such that $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ for every σ -single session responsive environment $\mathcal{E} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$.*

Now, the following theorem says that if a single session of \mathcal{P} realizes a single session of \mathcal{F} , i.e., if \mathcal{P} realizes \mathcal{F} for all σ -single session (responsive) environments, then multiple session of \mathcal{P} realize multiple sessions of \mathcal{F} , i.e., \mathcal{P} realizes \mathcal{F} for all (responsive) environments. This corresponds to Theorem 10 in [99] (cf. Theorem 2.2 in Chapter 2).

Theorem 3.4 (Composition Theorem for Unbounded Self-Composition of PSID-Dependent Protocols). *Let R be a restriction. Let σ be a PSID function, and let \mathcal{P} and \mathcal{F} be protocol systems such that \mathcal{P} and \mathcal{F} are σ -session versions and $\mathcal{P} \leq_{R,\sigma\text{-single}} \mathcal{F}$. Then, $\mathcal{P} \leq_R \mathcal{F}$.*

Before we can prove this theorem, we have to show some additional lemmas and another theorem.

In what follows, we will often use the (informal) term *copy of a system* to denote the set of all instances of a σ -session version (in some run) that accept messages with the same PSID. Note that this term is justified, because σ -session versions will never accept two messages with different PSIDs, and thus we can group instances according to the PSID of the first message they accepted.

Lemma 3.8 (Variant of Lemma 3.1 for σ -single session environments). *Let R be a restriction. Let \mathcal{P} and \mathcal{P}' be two systems of IITMs such that neither of them has a **start** or **decision** tape, both systems have the same external interface, and $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{P}'$ for all $\mathcal{E} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$. Then, $\text{Env}_{R,\sigma\text{-single}}(\mathcal{P}) = \text{Env}_{R,\sigma\text{-single}}(\mathcal{P}')$.*

Proof. The original proof can easily be adjusted: First, one assumes $\mathcal{E} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$. Then it is obvious that $\mathcal{E}' \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$ by construction. The remainder of the proof stays exactly the same. \square

Lemma 3.9 (Variant of Lemma 3.5 for σ -single session environments). *Let R be a restriction. Let \mathcal{R} and \mathcal{Q} be connectable systems such that $\mathcal{R} \mid \mathcal{Q}$ is almost bounded, **start** is a tape of \mathcal{R} (i.e. \mathcal{R} contains a master IITM), and **decision** is not a tape of \mathcal{Q} . Furthermore, let \mathcal{Q}' be a*

system which is compatible with \mathcal{Q} and satisfies the following condition: $\mathcal{E} \mid \mathcal{Q} \equiv \mathcal{E} \mid \mathcal{Q}'$ for every $\mathcal{E} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{Q})$ such that $\mathcal{E} \mid \mathcal{Q}$ is almost bounded. If $[\mathcal{R}]_{\mathcal{Q}} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{Q})$, then

$$[\mathcal{R}]_{\mathcal{Q}} \mid \mathcal{Q}' \equiv \mathcal{R} \mid \mathcal{Q}' .$$

Moreover, if $[\mathcal{R}]_{\mathcal{Q}} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{Q})$ and $[\mathcal{R}]_{\mathcal{Q}} \mid \mathcal{Q}'$ is almost bounded, then also $\mathcal{R} \mid \mathcal{Q}'$ is almost bounded.

Proof. The proof is analogous to the proof of Lemma 3.5. □

The following theorem says that unbounded self-composition holds true for the computational indistinguishability relation. From this theorem, we will obtain Theorem 3.4, i.e., unbounded self-composition for strong simulatability. The following theorem corresponds to the stronger version of Theorem 6 in [99].

Theorem 3.5. *Let R be a restriction. Let σ be a PSID function, and let \mathcal{P} be a protocol system such that \mathcal{P} is a σ -session version. Let \mathcal{Q} be a system of IITMs compatible with \mathcal{P} such that \mathcal{Q} is a σ -session version, every machine in \mathcal{Q} that is not in the scope of a bang accepts all messages in mode `CheckAddress`, and $\mathcal{E} \mid \mathcal{Q}$ is almost bounded for all $\mathcal{E} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{Q})$.¹⁶*

If $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{Q}$ for all $\mathcal{E} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$, then $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{Q}$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$, and \mathcal{Q} is R -environmentally bounded.

Proof. Let \mathcal{P}, \mathcal{Q} be systems of IITMs as required by the theorem such that $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{Q}$ for all $\mathcal{E} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$. Let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$. In the following, by a hybrid argument in which we replace the copies of \mathcal{P} by copies of \mathcal{Q} with the same PSID, we show that $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{Q}$. Then, we use this result to conclude that \mathcal{Q} is R -environmentally bounded.

By Lemma 6 from [99], we may assume that \mathcal{E} is a single IITM which, in mode `CheckAddress`, accepts all messages. We may also assume that `start` is an external tape of \mathcal{E} (otherwise there would be no master IITM in $\mathcal{E} \mid \mathcal{P}$ and $\mathcal{E} \mid \mathcal{Q}$, i.e., every run immediately ends with empty output. Thus, $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{Q}$ trivially holds true in this case). In addition, we may assume that the only external tapes of $\mathcal{E} \mid \mathcal{P}$ (and hence of $\mathcal{E} \mid \mathcal{Q}$) are `start` and `decision`. Moreover, we may assume, without loss of generality, that \mathcal{E} is such that every message m that \mathcal{E} outputs on tape c (except if m is output on tape `decision`) has some PSID, i.e. $\sigma(m, c) \neq \perp$: Because \mathcal{E} will interact only with σ -session versions, messages without a PSID would be rejected by these σ -session versions anyway. As \mathcal{E} is universally bounded, it follows that there exists a polynomial $p_{\mathcal{E}}$ such that the number of different sessions (i.e., messages with distinct PSIDs output by \mathcal{E}) is bounded from above by $p_{\mathcal{E}}(\eta, |a|)$ (where η is the security parameter and a is the external input given to \mathcal{E}).

In what follows, let \mathcal{Q}' be the variant of \mathcal{Q} obtained from \mathcal{Q} by renaming every tape c occurring in \mathcal{Q} to c' . Analogously, let \mathcal{P}'' be obtained from \mathcal{P} by renaming every tape c occurring in \mathcal{P} to c'' . By this, we have that \mathcal{P} , \mathcal{Q}' , and \mathcal{P}'' have pairwise disjoint sets of external tapes, and hence, these systems are pairwise connectable.

¹⁶Note that \mathcal{Q} is essentially a protocol system with a relaxed runtime bound that is only required to hold for σ -single session responsive environments, instead of arbitrary responsive environments.

We now define an IITM \mathcal{E}_r (for every $r \in \mathbb{N}$), which essentially simulates \mathcal{E} and which will run in the system $\mathcal{E}_r | \mathcal{P}'' | \mathcal{Q}' | \mathcal{P}$ or $\mathcal{E}_r | \mathcal{P}'' | \mathcal{Q}' | \mathcal{Q}$, respectively. The IITM \mathcal{E}_r randomly shuffles the copies of the systems \mathcal{P} and \mathcal{Q} , respectively, invoked by \mathcal{E} by choosing a random permutation on $\{1, \dots, p_{\mathcal{E}}(\eta, |a|)\}$. Intuitively, this permutation is needed because \mathcal{E} must not “know” whether the copy invoked in the system $\mathcal{P}'' | \mathcal{Q}' | \mathcal{P}$ ($\mathcal{P}'' | \mathcal{Q}' | \mathcal{Q}$) is a copy of \mathcal{P}'' , \mathcal{Q}' , or \mathcal{P} (\mathcal{P}'' , \mathcal{Q}' , or \mathcal{Q}). This is needed in the proof of Lemma 3.14. The first $r - 1$ copies (after shuffling) of the protocol invoked by \mathcal{E} will be copies of \mathcal{Q}' , the r -th copy will be the external system \mathcal{P} or \mathcal{Q} , respectively, and the remaining copies will be copies of \mathcal{P}'' .

Formally, \mathcal{E}_r is obtained from \mathcal{E} as follows. The IITM \mathcal{E} has tapes to connect to all of the external tapes of \mathcal{P} and \mathcal{Q} . \mathcal{E}_r has the same tapes. Furthermore, for each such tape c , we add two more tapes c' and c'' to \mathcal{E}_r to connect to the external tapes of \mathcal{Q}' and \mathcal{P}'' , respectively. Just as \mathcal{E} , the IITM \mathcal{E}_r will also always accept every message in mode `CheckAddress`. The behavior of \mathcal{E}_r in mode `Compute` is specified next.

We need to specify how \mathcal{E}_r redirects protocol invocations of \mathcal{E} in the way sketched above: \mathcal{E}_r keeps a list L of PSIDs, which initially is empty, and the length l of the list, which initially is 0. By definition of $p_{\mathcal{E}}$, it will always hold that $l \leq p_{\mathcal{E}}(\eta, |a|)$. Furthermore, in the first activation with security parameter $\eta \in \mathbb{N}$ and external input $a \in \{0, 1\}^*$, \mathcal{E}_r chooses a permutation π of $\{1, \dots, p_{\mathcal{E}}(\eta, |a|)\}$ uniformly at random. From now on, \mathcal{E}_r simulates \mathcal{E} with security parameter η and external input a . In particular, if \mathcal{E} produces output, then so does \mathcal{E}_r , and if \mathcal{E}_r receives input, then \mathcal{E} is simulated with this input. However, as explained next, the behavior of \mathcal{E}_r deviates from that of \mathcal{E} in terms of sending and receiving messages to the different copies of protocols.

1. If \mathcal{E} produces output m on some external tape c of \mathcal{P} (and hence, \mathcal{Q}) with $s := \sigma(m, c)$, then \mathcal{E}_r checks whether s occurs in L . If s does not occur in L , s is first appended at the end of L and l is increased by 1. Let $j \in \{1, \dots, l\}$ be the position at which s occurs in L .
 - a) If $\pi(j) < r$, then \mathcal{E}_r writes m on tape c' .
 - b) If $\pi(j) = r$, then \mathcal{E}_r writes m on c .
 - c) If $\pi(j) > r$, then \mathcal{E}_r writes m on tape c'' .
2. If \mathcal{E}_r receives input on tape c'' (where c'' is an external tape of \mathcal{P}'' corresponding to an external tape c of \mathcal{P}), then \mathcal{E}_r behaves as \mathcal{E} would if input was received on tape c .
3. If \mathcal{E}_r receives input on tape c' (where c' is an external tape of \mathcal{Q}' corresponding to an external tape c of \mathcal{Q}), then \mathcal{E}_r behaves as \mathcal{E} would if input was received on tape c .
4. If \mathcal{E}_r receives input on tape c (where c is an external tape of \mathcal{P} or \mathcal{Q} , respectively), then \mathcal{E}_r behaves as \mathcal{E} would if input was received on tape c .

It is easy to see that \mathcal{E}_r is universally bounded for every $r \in \mathbb{N}$ because \mathcal{E} is universally bounded.

We define the following hybrid systems, for every $r \in \mathbb{N}$:

$$\mathcal{H}_r := \mathcal{E}_r | \mathcal{P}'' | \mathcal{Q}' ,$$

which can be connected to \mathcal{P} (and hence \mathcal{Q}).

By $\mathcal{E}_{p_{\mathcal{E}}}$ and $\mathcal{H}_{p_{\mathcal{E}}}$ we denote the systems which first set $r = p_{\mathcal{E}}(\eta, |a|)$ and then behave exactly as \mathcal{E}_r and \mathcal{H}_r , respectively. By construction, for every $r \in \mathbb{N}$, the systems $\mathcal{E} | \mathcal{P}$ and $\mathcal{H}_1 | \mathcal{P}$, the systems $\mathcal{H}_r | \mathcal{Q}$ and $\mathcal{H}_{r+1} | \mathcal{P}$, and the systems $\mathcal{E} | \mathcal{Q}$ and $\mathcal{H}_{p_{\mathcal{E}}} | \mathcal{Q}$, respectively, behave exactly the same (see below). In particular, for all $r \in \mathbb{N}$, we have that

$$\mathcal{E} | \mathcal{P} \equiv \mathcal{H}_1 | \mathcal{P} , \tag{3.3}$$

$$\mathcal{H}_r | \mathcal{Q} \equiv \mathcal{H}_{r+1} | \mathcal{P} , \text{ and} \tag{3.4}$$

$$\mathcal{E} | \mathcal{Q} \equiv \mathcal{H}_{p_{\mathcal{E}}} | \mathcal{Q} . \tag{3.5}$$

For (3.3), we need that \mathcal{P} is a protocol system and a σ -session version:

The second property is necessary because it guarantees that two instances of machines in \mathcal{P} with different PSIDs will never send messages to each other (as they only send messages with their own PSID or produce empty output, which activates the environment), i.e., it is no problem that the systems \mathcal{P}'' and \mathcal{P} in $\mathcal{H}_1 | \mathcal{P}$ do not have a connection to each other.

Furthermore, for machines M of \mathcal{P} that are in the scope of a bang, it is easy to see that every instance of such a machine in a run of $\mathcal{E} | \mathcal{P}$ corresponds to one instance of this machine (either in \mathcal{P}'' or \mathcal{P} , depending on the PSID of the instance) in a run of $\mathcal{H}_1 | \mathcal{P}$.

If there is a machine M in \mathcal{P} that is not in the scope of a bang, the following problem can occur: In a run of $\mathcal{E} | \mathcal{P}$, there can only be at most one instance of such a machine, whereas in a run of $\mathcal{H}_1 | \mathcal{P}$ there may be two instances (one in \mathcal{P}'' and one in \mathcal{P}), i.e., one of those instances would not have a corresponding instance of M in $\mathcal{E} | \mathcal{P}$. This is where we need that \mathcal{P} is a protocol system, i.e., that such a machine M must accept all messages m on all tapes c in mode `CheckAddress`. As M is a σ -session version, it accepts only messages with the same PSID, and thus we obtain $\sigma(m, c) = s$ for some fixed value $s \neq \perp$, $m \in \{0, 1\}^*$ and all input tapes c of M . (If $\sigma(m, c)$ were \perp for some $m \in \{0, 1\}^*$ and some input tape c of M , then M would accept m in contradiction to the assumption that M is a σ -session version. Similarly, if there existed (m, c) and (m', c') with $\sigma(m, c) \neq \sigma(m', c')$, then M would accept both (m, c) and (m', c') , again in contradiction to the assumption that M is a σ -session version.) As \mathcal{E}_1 never sends messages with the same PSID to both \mathcal{P} and \mathcal{P}'' , there will be at most one copy of either \mathcal{P} or \mathcal{P}'' with PSID s . But then there is also only one instance of M in a run of $\mathcal{H}_1 | \mathcal{P}$, which corresponds to the single instance of M in a run of $\mathcal{E} | \mathcal{P}$.

A similar argument is used for (3.4) and (3.5).

As $\mathcal{E} | \mathcal{P}$ is almost bounded by assumption (i.e., \mathcal{P} is a protocol system and thus R-environmentally bounded), it is easy to see that also $\mathcal{H}_1 | \mathcal{P}$ is almost bounded. Moreover, it is easy to see that $\mathcal{E} | \mathcal{Q}$ is almost bounded if and only if $\mathcal{H}_{p_{\mathcal{E}}} | \mathcal{Q}$ is almost bounded.

We proceed by showing the following claim.

Claim 3.1. $\mathcal{H}_{p_{\mathcal{E}}} | \mathcal{Q}$ is almost bounded.

This is a crucial and in fact the most difficult step of the proof. For this, we first have to define some events that describe the runtime of systems and the probability of answering a restricting message incorrectly.

For different systems, we define the event (i.e. a set of runs or equivalently a set of random coins) that the j -th copy of the system takes more than $q(\eta, |a|)$ steps. More specifically, for every polynomial q , natural numbers $r, i, j, \eta \in \mathbb{N}$, and $a \in \{0, 1\}^*$, we define:

1. $B_{\mathcal{H}_r | \mathcal{P}}^{q,j} = B_{\mathcal{H}_r | \mathcal{P}}^{q,j}(1^\eta, a)$ is the following set of runs of $(\mathcal{H}_r | \mathcal{P})(1^\eta, a)$. A run ρ belongs to $B_{\mathcal{H}_r | \mathcal{P}}^{q,j}$ iff one of the following conditions is satisfied:
 - a) $\pi(j) < r$ and the instances of machines in \mathcal{Q}' with PSID $L[j]$ took more than $q(\eta, |a|)$ steps in ρ , where only the steps in mode `Compute` are counted; in other words, only the computation steps carried out by \mathcal{Q}' (and hence, \mathcal{Q}) are counted.
 - b) $\pi(j) = r$ and the machines in \mathcal{P} (with PSID $L[j]$) took more than $q(\eta, |a|)$ steps in ρ , with the steps counted as above.
 - c) $\pi(j) > r$ and the instances of machines in \mathcal{P}'' with PSID $L[j]$ took more than $q(\eta, |a|)$ steps in ρ , with the steps counted as above.
2. $B_{\mathcal{H}_r | \mathcal{P}}^q := \bigcup_{i \in \mathbb{N}} B_{\mathcal{H}_r | \mathcal{P}}^{q,i}$.
3. $B_{\mathcal{H}_r | \mathcal{P}}^{q, \neq j} := \bigcup_{i \in \mathbb{N} \setminus \{j\}} B_{\mathcal{H}_r | \mathcal{P}}^{q,i}$.
4. A run ρ belongs to $B_{\mathcal{H}_r | \mathcal{P}}^{q, \pi^{-1}(i)}$ iff for the permutation, say π' , chosen in ρ , it holds that $\rho \in B_{\mathcal{H}_r | \mathcal{P}}^{q,j}$ with $j = \pi'^{-1}(i)$. The event $B_{\mathcal{H}_r | \mathcal{P}}^{q, \neq \pi^{-1}(i)}$ is defined analogously.
5. Analogously to the above events, we define the following events in which the external system \mathcal{P} is replaced by \mathcal{Q} : $B_{\mathcal{H}_r | \mathcal{Q}}^{q,j}$, $B_{\mathcal{H}_r | \mathcal{Q}}^q$, $B_{\mathcal{H}_r | \mathcal{Q}}^{q, \neq j}$, $B_{\mathcal{H}_r | \mathcal{Q}}^{q, \pi^{-1}(i)}$, $B_{\mathcal{H}_r | \mathcal{Q}}^{q, \neq \pi^{-1}(i)}$.

We note that for all r , the system $\mathcal{H}_r | \mathcal{P}$ (resp., $\mathcal{H}_r | \mathcal{Q}$) is almost bounded if and only if there exists a negligible function f and a polynomial q such that $\Pr \left[B_{\mathcal{H}_r | \mathcal{P}}^q(1^\eta, a) \right] \leq f(\eta, a)$ (resp., $\Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^q(1^\eta, a) \right] \leq f(\eta, a)$) for all $\eta \in \mathbb{N}$ and $a \in \{0, 1\}^*$. The direction from left to right is trivial. For the other direction, first recall that (the simulated) \mathcal{E} is universally bounded, and hence, only a polynomial number of copies of the protocol is created (the length l of the list L in \mathcal{H}_r is bounded by $p_{\mathcal{E}}(\eta, |a|)$). Now, because by assumption the number of steps taken by every copy of the protocol is polynomially bounded (except with negligible probability) and the number of steps taken by (the simulated) \mathcal{E} is polynomially bounded, the number of steps taken by $\mathcal{H}_r | \mathcal{P}$ (resp., $\mathcal{H}_r | \mathcal{Q}$) is polynomially bounded.

Furthermore, for a system \mathcal{S} with tape `start` and a polynomial q we define the single IITM $[\mathcal{S}]_q$ to be the IITM that simulates \mathcal{S} but simulates at most $q(\eta, |a|)$ many steps of \mathcal{S} in mode

Compute. That is, $[\mathcal{S}]_q$ works just like the definition of $[\mathcal{S}]_{\mathcal{R}}$ (for some system \mathcal{R} such that $\mathcal{S} | \mathcal{R}$ is almost bounded) from Lemma 3.2, but uses a fixed polynomial q instead of the runtime bound of $\mathcal{S} | \mathcal{R}$.

Finally, we also need to define some events that include runs in which a restricting message from some part of a system is answered incorrectly. More specifically:

1. For all systems \mathcal{S}, \mathcal{R} the event $C_{\mathcal{S}}^{\mathcal{R}} = C_{\mathcal{S}}^{\mathcal{R}}(1^\eta, a)$ is defined to be the set of all runs of $\mathcal{S} | \mathcal{R}$ in which a restricting message of \mathcal{R} is answered incorrectly.
2. For all $r \in \mathbb{N}$ and polynomials q , the event $\hat{C}_{r,q}^{\mathcal{P}} = \hat{C}_{r,q}^{\mathcal{P}}(1^\eta, a)$ is defined to be the set of all runs of $[\mathcal{H}_r]_q | \mathcal{P} = [\mathcal{E}_r | \mathcal{P}'' | \mathcal{Q}']_q | \mathcal{P}$ in which a restricting message of \mathcal{P} , \mathcal{P}'' , or \mathcal{Q}' is answered incorrectly (by \mathcal{E}_r).
3. Analogous to $\hat{C}_{r,q}^{\mathcal{P}}$, we define $\hat{C}_{r,q}^{\mathcal{Q}}$ by replacing all occurrences of \mathcal{P} with \mathcal{Q} .

After having fixed some definitions, we can now start proving Claim 3.1. First, we present a lemma which shows that, for all $r \in \mathbb{N}$, there exists a polynomial q_r such that the event $B_{\mathcal{H}_r}^{q_r} | \mathcal{Q}$ occurs with negligible probability. As argued above, this is equivalent to $\mathcal{H}_r | \mathcal{Q}$ being almost bounded. This, however, is not sufficient to conclude that $\mathcal{H}_{p_\varepsilon} | \mathcal{Q}$ is almost bounded as we need a uniform runtime bound that is independent of r for this result (such a bound will be established in Lemma 3.14). Obtaining such a uniform runtime bound is in fact one of the main difficulties in proving Claim 3.1 and the only reason the permutation π is required. Additionally, the next lemma also shows some properties of the system \mathcal{E}_r which are required by the following lemmas.

Lemma 3.10. *For all $r \in \mathbb{N}$ the following holds true:*

1. *There exists a polynomial q_r and a negligible function f_r such that for all $\eta \in \mathbb{N}$ and $a \in \{0, 1\}^*$:*

$$\Pr \left[B_{\mathcal{H}_r}^{q_r} | \mathcal{Q}(1^\eta, a) \right] \leq f_r(\eta, a) .$$
2. $\mathcal{E}_r \in \text{Env}_R(\mathcal{P}'' | \mathcal{Q}' | \mathcal{P})$ and $\mathcal{E}_r \in \text{Env}_R(\mathcal{P}'' | \mathcal{Q}' | \mathcal{Q})$.

Proof. The case $r = 0$ is simple: By construction the system \mathcal{E}_0 in a run of $\mathcal{E}_0 | \mathcal{P}'' | \mathcal{Q}' | \mathcal{P}$ sends all messages to \mathcal{P}'' , i.e., \mathcal{Q}' and \mathcal{P} will never be activated by \mathcal{E}_0 and, as they do not contain a master IITM, they are also not activated via **start**. Thus, we can drop \mathcal{Q}' and \mathcal{P} and only look at $\mathcal{E}_0 | \mathcal{P}''$, which behaves just as $\mathcal{E} | \mathcal{P}$. As $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ and $\mathcal{E} | \mathcal{P}$ is almost bounded (because \mathcal{P} is R-environmentally bounded), we have that $\mathcal{E}_0 \in \text{Env}_R(\mathcal{P}'' | \mathcal{Q}' | \mathcal{P})$ and $\mathcal{E}_0 | \mathcal{P}'' | \mathcal{Q}' | \mathcal{P}$ is almost bounded. We can also easily define a bijection between runs of $\mathcal{E}_0 | \mathcal{P}'' | \mathcal{Q}' | \mathcal{P}$ and $\mathcal{E}_0 | \mathcal{P}'' | \mathcal{Q}' | \mathcal{Q}$, as \mathcal{P} and \mathcal{Q} , respectively, are never activated in any run. By this, we obtain that $\mathcal{E}_0 \in \text{Env}_R(\mathcal{P}'' | \mathcal{Q}' | \mathcal{Q})$ and $\mathcal{E}_0 | \mathcal{P}'' | \mathcal{Q}' | \mathcal{Q}$ is almost bounded (the latter implies that q_r and f_r for $r = 0$ exist, as argued above).

Now, let $r > 0$ and suppose that the lemma holds for $r - 1$, i.e., $\mathcal{H}_{r-1} | \mathcal{Q}$ is almost bounded, $\mathcal{E}_{r-1} \in \text{Env}_R(\mathcal{P}'' | \mathcal{Q}' | \mathcal{P})$, and $\mathcal{E}_{r-1} \in \text{Env}_R(\mathcal{P}'' | \mathcal{Q}' | \mathcal{Q})$. Because $\mathcal{H}_{r-1} | \mathcal{Q}$ and $\mathcal{H}_r | \mathcal{P}$ behave

exactly the same (see Equation (3.4)), $\mathcal{E}_{r-1} \in \text{Env}_R(\mathcal{P}'' \mid \mathcal{Q}' \mid \mathcal{Q})$ implies $\mathcal{E}_r \in \text{Env}_R(\mathcal{P}'' \mid \mathcal{Q}' \mid \mathcal{P})$, and $\mathcal{H}_{r-1} \mid \mathcal{Q}$ being almost bounded implies that $\mathcal{H}_r \mid \mathcal{P}$ is almost bounded. We also know that restricting messages from \mathcal{P} are answered correctly in an overwhelming set of runs of $\mathcal{H}_r \mid \mathcal{P}$, because all external tapes of \mathcal{P} are external tapes of $\mathcal{P}'' \mid \mathcal{Q}' \mid \mathcal{P}$ and \mathcal{E}_r is a responsive environment for $\mathcal{P}'' \mid \mathcal{Q}' \mid \mathcal{P}$. Now Lemma 3.2 and Lemma 3.4 imply $\mathcal{H}_r \mid \mathcal{P} \equiv [\mathcal{H}_r]_{\mathcal{P}} \mid \mathcal{P}$ with $[\mathcal{H}_r]_{\mathcal{P}} \in \text{Env}_R(\mathcal{P})$. Note that $[\mathcal{H}_r]_{\mathcal{P}}$ is a σ -single session responsive environment, i.e., $[\mathcal{H}_r]_{\mathcal{P}} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$, as the only external tapes (except for **start** and **decision**) of $[\mathcal{H}_r]_{\mathcal{P}}$ are those that connect \mathcal{E}_r and \mathcal{P}/\mathcal{Q} and by construction, \mathcal{E}_r sends only messages with the same PSID on those tapes. By Lemma 3.8, we obtain that $[\mathcal{H}_r]_{\mathcal{P}} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{Q})$, and in particular, that $[\mathcal{H}_r]_{\mathcal{P}} \mid \mathcal{Q}$ is almost bounded. Finally, by Lemma 3.9, we also obtain $[\mathcal{H}_r]_{\mathcal{P}} \mid \mathcal{Q} \equiv \mathcal{H}_r \mid \mathcal{Q}$ and that $\mathcal{H}_r \mid \mathcal{Q}$ is almost bounded, which implies the existence of q_r and f_r .

We still have to show that $\mathcal{E}_r \in \text{Env}_R(\mathcal{P}'' \mid \mathcal{Q}' \mid \mathcal{Q})$. Suppose this was not the case, i.e., a restricting message from $\mathcal{P}'' \mid \mathcal{Q}' \mid \mathcal{Q}$ is answered incorrectly in a non-negligible set of runs. Then there must be a non-negligible set of runs of $[\mathcal{H}_r]_{\mathcal{P}} \mid \mathcal{Q}$ in which a restricting message from \mathcal{Q} or $\mathcal{P}'' \mid \mathcal{Q}'$ (which are simulated by $[\mathcal{H}_r]_{\mathcal{P}}$) is answered incorrectly. This is because $[\mathcal{H}_r]_{\mathcal{P}} \mid \mathcal{Q}$ behaves just like $\mathcal{H}_r \mid \mathcal{Q}$ except for a negligible set of runs in which a runtime bound is reached. Furthermore, because $\mathcal{E}_r \in \text{Env}_R(\mathcal{P}'' \mid \mathcal{Q}' \mid \mathcal{P})$ and every restricting message of \mathcal{P} or $\mathcal{P}'' \mid \mathcal{Q}'$ is a restricting message of $\mathcal{P}'' \mid \mathcal{Q}' \mid \mathcal{P}$, we obtain with a similar argument that there must be a negligible set of runs of $[\mathcal{H}_r]_{\mathcal{P}} \mid \mathcal{P}$ in which a restricting message from \mathcal{P} or $\mathcal{P}'' \mid \mathcal{Q}'$ is answered incorrectly. Now one can construct a new environment $\mathcal{E}' \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$ that distinguishes \mathcal{P} and \mathcal{Q} . The system \mathcal{E}' connects to \mathcal{P}/\mathcal{Q} and simulates $[\mathcal{H}_r]_{\mathcal{P}}$ (the system $[\mathcal{H}_r]_{\mathcal{P}}$ is universally bounded and thus can be simulated by an environment), but produces different output on tape **decision**: It outputs 1 if and only if a restricting message from $\mathcal{P}'' \mid \mathcal{Q}'$ or \mathcal{P} and \mathcal{Q} , respectively, was answered incorrectly during the run. Note that \mathcal{E}' can easily check this as it sees all messages on all tapes except for internal tapes of \mathcal{P} and \mathcal{Q} , respectively, and R is decidable in polynomial time in the second component (see also the proof of Lemma 3.1, which includes a more detailed discussion on why this can be done by a universally bounded system). It is also obvious that $\mathcal{E}' \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$ as $[\mathcal{H}_r]_{\mathcal{P}} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$. Because this contradicts the assumption that \mathcal{P} and \mathcal{Q} are indistinguishable for all σ -single session responsive environments, it must hold true that $\mathcal{E}_r \in \text{Env}_R(\mathcal{P}'' \mid \mathcal{Q}' \mid \mathcal{Q})$.

This concludes the proof of Lemma 3.10. □

In the next steps of the proof of Claim 3.1, we will define several systems that get r as external input and then simulate $[\mathcal{H}_r]_q$ for some polynomial q . We need that such a system is a responsive environment, i.e., correctly answers restricting messages from \mathcal{P} in an overwhelming set of runs. For this it is not sufficient to show that $[\mathcal{H}_r]_q \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$, i.e., that for every r there exists a negligible function that bounds $\Pr \left[C_{[\mathcal{H}_r]_q}^{\mathcal{P}} \right]$. Instead we need a universal function that bounds $\Pr \left[C_{[\mathcal{H}_r]_q}^{\mathcal{P}} \right]$ for all values of r (in some range). The next two lemmas show that such a function exists.

Lemma 3.11. *Let q be a polynomial. Then there exists a negligible function $g_{1,q}$ such that for all $\eta \in \mathbb{N}$, $a \in \{0,1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$:*

$$|\Pr [\hat{C}_{r,q}^{\mathcal{P}}(1^\eta, a)] - \Pr [\hat{C}_{r,q}^{\mathcal{Q}}(1^\eta, a)]| \leq g_{1,q}(\eta, a) .$$

Proof. We prove this lemma for the case of non-uniform environments, that is, environments that obtain external input. Using the same technique as in [99] (see Appendix B in that publication), namely sampling of runs, one easily obtains a proof for uniform environments, i.e., those that do not obtain external input, as well.

We define an IITM $D \in \text{Env}_R(\mathcal{P})$ which expects to receive (r, a) as external input and then simulates $[\mathcal{H}_r]_q$ with external input a and outputs 1 if and only if $[\mathcal{H}_r]_q$ is about to send an incorrect answer to a restricting message of \mathcal{P}'' , \mathcal{Q}' , or \mathcal{P} and \mathcal{Q} , respectively.

More formally, D is defined as follows. The IITM D has the same (external) tapes as \mathcal{H}_r . In mode **CheckAddress**, D accepts every message. In mode **Compute**, D behaves as follows: First, D parses the external input on **start** as (r, a) , with $r \in \{1, \dots, p_{\mathcal{E}}(\eta, |a|)\}$ and $a \in \{0, 1\}^*$. If the external input is not of this form, D outputs 0 on **decision**. Otherwise, D simulates the system $[\mathcal{H}_r]_q = [\mathcal{E}_r | \mathcal{P}'' | \mathcal{Q}']_q$ with external input a , i.e., it first activates \mathcal{E}_r with input a on **start** and then simulates all the machines in $[\mathcal{H}_r]_q$. If D receives input on a tape, D forwards this input to the simulated $[\mathcal{H}_r]_q$. If $[\mathcal{H}_r]_q$ produces output, then D first checks whether the output is an incorrect response to a restricting message of \mathcal{P}'' , \mathcal{Q}' , and \mathcal{P} and \mathcal{Q} , respectively. Note that this is easy to check because D sees all restricting messages from those systems (recall that we assume that \mathcal{E} and hence \mathcal{E}_r , \mathcal{H}_r , and D connect to all tapes of \mathcal{P} and \mathcal{Q} , respectively) and R is decidable in polynomial time in the second component (again, see the proof of Lemma 3.1, which includes a more detailed discussion on why this can be done by a universally bounded system). If the answer is an incorrect response, D outputs 1 on **decision** instead of sending the message. If the run terminates (i.e., $[\mathcal{H}_r]_q$ outputs something on **decision** or $[\mathcal{H}_r]_q$ produces empty output, which terminates the run because $[\mathcal{H}_r]_q$ is a master IITM), then D halts with output 0 on **decision**.

It is easy to see that D is universally bounded (i.e. $D \in \text{Env}(\mathcal{P})$) and that it is a responsive environment for \mathcal{P} and \mathcal{Q} because it will never incorrectly answer a restricting message from \mathcal{P} and \mathcal{Q} , respectively. It is also obvious that D is a σ -single session environment, as it simulates $[\mathcal{H}_r]_q$ (for some r) and \mathcal{H}_r only sends messages with a single PSID to \mathcal{P} and \mathcal{Q} , respectively. We have for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta)$:

$$\begin{aligned} \Pr [(D | \mathcal{P})(1^\eta, (r, a)) = 1] &= \Pr [\hat{C}_{r,q}^{\mathcal{P}}(1^\eta, a)] \quad \text{and} \\ \Pr [(D | \mathcal{Q})(1^\eta, (r, a)) = 1] &= \Pr [\hat{C}_{r,q}^{\mathcal{Q}}(1^\eta, a)] . \end{aligned} \tag{3.6}$$

As $\mathcal{E}' | \mathcal{P} \equiv \mathcal{E}' | \mathcal{Q}$ for all $\mathcal{E}' \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$ and $D \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$, we have that $D | \mathcal{P} \equiv D | \mathcal{Q}$. In particular, there exists a negligible function $g'_{1,q}$ such that for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$,

and $1 \leq r \leq p_{\mathcal{E}}(\eta)$:

$$\begin{aligned} g'_{1,q}(\eta, (r, a)) &\geq |\Pr[(D | \mathcal{P})(1^\eta, (r, a)) = 1] - \Pr[(D | \mathcal{Q})(1^\eta, (r, a)) = 1]| \\ &\stackrel{(3.6)}{=} |\Pr[\hat{C}_{r,q}^{\mathcal{P}}(1^\eta, a)] - \Pr[\hat{C}_{r,q}^{\mathcal{Q}}(1^\eta, a)]|. \end{aligned}$$

Let $g_{1,q}(\eta, a) := \max_{r \leq p_{\mathcal{E}}(\eta, |a|)} g'_{1,q}(\eta, (r, a))$. It is easy to see that $g_{1,q}$ is a negligible function. Now, clearly we have that

$$|\Pr[\hat{C}_{r,q}^{\mathcal{P}}(1^\eta, a)] - \Pr[\hat{C}_{r,q}^{\mathcal{Q}}(1^\eta, a)]| \leq g_{1,q}(\eta, a)$$

for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $r \leq p_{\mathcal{E}}(\eta, |a|)$. Note that $g_{1,q}$ does not depend on r (only on q). This concludes the proof of Lemma 3.11. \square

We can now use Lemma 3.11 to prove that, for every polynomial q , the probability of $C_{[\mathcal{H}_r]_q}^{\mathcal{P}}$ is bounded by a negligible function $g_{2,q}$ that is independent of r (for $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$). Note that this implies in particular that $[\mathcal{H}_r]_q \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$ because $[\mathcal{H}_r]_q$ is universally bounded and the only external tapes (except for **start** and **decision**) of $[\mathcal{H}_r]_q$ are those that connect \mathcal{E}_r and \mathcal{P}/\mathcal{Q} and, by construction, \mathcal{E}_r sends only messages with the same PSID on those tapes.

Lemma 3.12. *Let q be a polynomial. There exists a negligible function $g_{2,q}$ such that for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$:*

$$\Pr[C_{[\mathcal{H}_r]_q}^{\mathcal{P}}(1^\eta, a)] \leq g_{2,q}(\eta, a) .$$

Proof. We will show the following stronger statement instead: Let q be a polynomial. There exists a negligible function $g_{2,q}$ such that for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ it holds true that $\Pr[\hat{C}_{r,q}^{\mathcal{P}}(1^\eta, a)] \leq g_{2,q}(\eta, a)$. As $C_{[\mathcal{H}_r]_q}^{\mathcal{P}} \subseteq \hat{C}_{r,q}^{\mathcal{P}}$, the lemma trivially follows by using the same $g_{2,q}$.

First, let $r = 1$. Observe that $\Pr[\hat{C}_{1,q}^{\mathcal{P}}(1^\eta, a)]$ is bounded by a negligible function $g(\eta, a)$ because $\mathcal{E}_1 \in \text{Env}_R(\mathcal{P}'' | \mathcal{Q}' | \mathcal{P})$ by Lemma 3.10.

Now let $r > 1$. We have that

$$\begin{aligned} \Pr[\hat{C}_{r,q}^{\mathcal{P}}(1^\eta, a)] &\stackrel{(3.4)}{=} \Pr[\hat{C}_{r-1,q}^{\mathcal{Q}}(1^\eta, a)] \\ &\stackrel{\text{Lemma 3.11}}{\leq} \Pr[\hat{C}_{r-1,q}^{\mathcal{P}}(1^\eta, a)] + g_{1,q}(\eta, a). \end{aligned}$$

By induction on r , we obtain that

$$\begin{aligned} \Pr[\hat{C}_{r,q}^{\mathcal{P}}(1^\eta, a)] &\leq (r-1) \cdot g_{1,q}(\eta, a) + \Pr[\hat{C}_{1,q}^{\mathcal{P}}(1^\eta, a)] \\ &\leq (r-1) \cdot g_{1,q}(\eta, a) + g(\eta, a). \end{aligned}$$

We define

$$g_{2,q}(\eta, a) := p_{\mathcal{E}}(\eta, a) \cdot g_{1,q}(\eta, a) + g(\eta, a)$$

and obtain

$$\Pr \left[\hat{C}_{r,q}^{\mathcal{P}}(1^\eta, a) \right] \leq g_{2,q}(\eta, a)$$

for all $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$. Note that $g_{2,q}$ is negligible and does not depend on r (only on q). This concludes the proof of Lemma 3.12. \square \square

We can now continue to prove Claim 3.1, i.e., that $\mathcal{H}_{p_{\mathcal{E}}} | \mathcal{Q}$ is almost bounded. First, we will show in the next lemma that for all $r \leq p_{\mathcal{E}}(\eta, |a|)$ the difference of the probabilities that the events $B_{\mathcal{H}_r | \mathcal{P}}^{q_1, \neq \pi^{-1}(r)}$ and $B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}$ occur is negligible (where q_1 is the polynomial from Lemma 3.10 for $r = 1$). Note that this statement does not consider the number of steps taken by the external protocol copy \mathcal{P} or \mathcal{Q} , respectively, because this copy, which corresponds to $\pi^{-1}(r)$, is excluded in these events.

Lemma 3.13. *There exists a negligible function f' such that for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$:*

$$\left| \Pr \left[B_{\mathcal{H}_r | \mathcal{P}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a) \right] - \Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a) \right] \right| \leq f'(\eta, a) .$$

Proof. Just as in Lemma 3.11, we prove this lemma for the case of non-uniform environments. Again, using the same technique as in [99] (see Appendix B), one easily obtains a proof for uniform environments.

This proof follows essentially the same idea as the proof of Lemma 3.11. We define an IITM $D \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$ which expects to receive (r, a) as external input and then simulates \mathcal{H}_r with external input a until the runtime bound $q_1(\eta, |a|)$ is exceeded by any of the internally simulated sessions. If the runtime bound is exceeded, D outputs 1 on **decision**. If the run stops and the runtime bound has not been exceeded, D outputs 0 on **decision**.

More formally, D is defined as follows. In mode **CheckAddress**, D accepts every message. In mode **Compute**, D behaves as follows: First, D parses the external input on **start** as (r, a) , with $r \in \{1, \dots, p_{\mathcal{E}}(\eta, |a|)\}$ and $a \in \{0, 1\}^*$. If the external input is not of this form, D outputs 0 on **decision**. Otherwise, D simulates the system $\mathcal{H}_r = \mathcal{E}_r | \mathcal{P}'' | \mathcal{Q}'$ with external input a , i.e., it first activates \mathcal{E}_r with input a on **start** and then simulates all the machines in \mathcal{H}_r . If \mathcal{H}_r produces output, then so does D and if D receives input, then D forwards this input to the simulated \mathcal{H}_r . Additionally, D counts the number of transitions taken by all the simulated machines and checks if the conditions of the event $B_{\mathcal{H}_r | \mathcal{P}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a)$ and $B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a)$, respectively, are satisfied (note that D can do this even though it cannot inspect the number of transitions in the external systems, \mathcal{P} and \mathcal{Q} , respectively). If such a condition is satisfied (i.e., some internal session takes more than $q_1(\eta, |a|)$ steps), then D halts with output 1 on **decision**.

If the run terminates (i.e., \mathcal{E}_r outputs something on **decision** or \mathcal{E}_r produces empty output, which terminates the run because \mathcal{E}_r is a master IITM) but the conditions are not satisfied, then D halts with output 0 on **decision**.

It is easy to see that D is universally bounded (i.e., $D \in \text{Env}(\mathcal{P})$) and that it is σ -single session. We now show that $D \in \text{Env}_R(\mathcal{P})$: If the input of the system $D | \mathcal{P}$ is not of the expected form, D will never send a message to \mathcal{P} and thus never violate the condition of responsive environments. More formally, $\Pr [C_D^{\mathcal{P}}(1^\eta, a')] = 0$ for all a' such that a' cannot be parsed as (r, a) for some r with $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$. So we only have to find a bound for those cases where the input is of the correct form, i.e., the input a' is of the form (r, a) , with $r \in \{1, \dots, p_{\mathcal{E}}(\eta, |a|)\}$; this bound is then also an upper bound for all other inputs.

As D is universally bounded, there exists a polynomial q such that the runtime of the simulated \mathcal{H}_r in mode **Compute** is bounded by q in every run. Now, observe that $(D | \mathcal{P})(1^\eta, (r, a))$ and $([\mathcal{H}_r]_q | \mathcal{P})(1^\eta, a)$ behave in the same way, except that $(D | \mathcal{P})(1^\eta, (r, a))$ might terminate earlier than $([\mathcal{H}_r]_q | \mathcal{P})(1^\eta, a)$, namely, if in a session simulated by D the runtime bound q_1 is reached. Hence, we have that $\Pr [C_D^{\mathcal{P}}(1^\eta, (r, a))] \leq \Pr [C_{[\mathcal{H}_r]_q}^{\mathcal{P}}(1^\eta, a)]$. This is because $\Pr [C_D^{\mathcal{P}}(1^\eta, a)] = \Pr [C_{[\mathcal{H}_r]_q}^{\mathcal{P}}(1^\eta, a) \cap \text{Runtime}_{q_1}]$, where Runtime_{q_1} is the event that a restricting message of \mathcal{P} is answered incorrectly before any session of \mathcal{P}'' and \mathcal{Q}' in $[\mathcal{H}_r]_q$ exceeds the runtime bound q_1 . Now, by Lemma 3.12, this implies $\Pr [C_D^{\mathcal{P}}(1^\eta, (r, a))] \leq g_{2,q}(\eta, a)$ (note that this bound is the same for every valid r , i.e., it is independent of r). Combined with the previous observation, this shows that $D \in \text{Env}_R(\mathcal{P})$, and thus $D \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$.

By the definition of D , we have for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$:

$$\begin{aligned} \Pr [(D | \mathcal{P})(1^\eta, (r, a)) = 1] &= \Pr [B_{\mathcal{H}_r | \mathcal{P}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a)] \quad \text{and} \\ \Pr [(D | \mathcal{Q})(1^\eta, (r, a)) = 1] &= \Pr [B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a)] \quad . \end{aligned} \tag{3.7}$$

Because $\mathcal{E}' | \mathcal{P} \equiv \mathcal{E}' | \mathcal{Q}$ for all $\mathcal{E}' \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$ and $D \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$, we have that $D | \mathcal{P} \equiv D | \mathcal{Q}$. In particular, there exists a negligible function f such that for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$:

$$\begin{aligned} f(\eta, (r, a)) &\geq |\Pr [(D | \mathcal{P})(1^\eta, (r, a)) = 1] - \Pr [(D | \mathcal{Q})(1^\eta, (r, a)) = 1]| \\ &\stackrel{(3.7)}{=} |\Pr [B_{\mathcal{H}_r | \mathcal{P}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a)] - \Pr [B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a)]| \quad . \end{aligned}$$

Let $f'(\eta, a) := \max_{r \leq p_{\mathcal{E}}(\eta, |a|)} f(\eta, (r, a))$. It is easy to see that f' is a negligible function. Now, obviously we have that

$$|\Pr [B_{\mathcal{H}_r | \mathcal{P}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a)] - \Pr [B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a)]| \leq f'(\eta, a)$$

for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$. This concludes the proof of Lemma 3.13. \square

In the next step, we can finally prove that there is a polynomial q' such that there is a universal bound, i.e., one that is independent of r , for the probability of $B_{\mathcal{H}_r|Q}^{q'}$. This directly implies that $\mathcal{H}_{p\mathcal{E}}|Q$ is almost bounded, and hence, Claim 3.1.

Lemma 3.14. *There exists a polynomial q' and a negligible function f'' such that for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $0 \leq r \leq p\mathcal{E}(\eta, |a|)$:*

$$\Pr \left[B_{\mathcal{H}_r|Q}^{q'}(1^\eta, a) \right] \leq f''(\eta, a) .$$

Proof. We will first prove a (slightly) weaker version of the lemma which holds true only for $1 \leq r \leq p\mathcal{E}(\eta, |a|)$. For this we use the polynomial q_1 from Lemma 3.10 (for $r = 1$) and construct a negligible function g . Afterwards we use this result to show the lemma for $0 \leq r \leq p\mathcal{E}(\eta, |a|)$.

Let $\eta \in \mathbb{N}$, and $a \in \{0, 1\}^*$. For all $1 \leq r \leq p\mathcal{E}(\eta, |a|)$, we define

$$t_r := t_r(1^\eta, a) := \Pr \left[B_{\mathcal{H}_r|Q}^{q_1}(1^\eta, a) \right] .$$

We need to show that there exists a negligible function that bounds t_r from above for every $1 \leq r \leq p\mathcal{E}(\eta, |a|)$.

Let $1 \leq r \leq p\mathcal{E}(\eta, |a|)$. If $r = 1$, then $t_r = t_1 \leq f_1(\eta, a)$ by Lemma 3.10. Next, we consider the case $r > 1$. Because the permutation π is chosen uniformly at random, for all $j \leq r$, we obtain the following equality:

$$\Pr \left[B_{\mathcal{H}_r|Q}^{q_1, \pi^{-1}(r)}(1^\eta, a) \setminus B_{\mathcal{H}_r|Q}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a) \right] = \Pr \left[B_{\mathcal{H}_r|Q}^{q_1, \pi^{-1}(j)}(1^\eta, a) \setminus B_{\mathcal{H}_r|Q}^{q_1, \neq \pi^{-1}(j)}(1^\eta, a) \right] . \quad (3.8)$$

Intuitively, the event $B_1 := B_{\mathcal{H}_r|Q}^{q_1, \pi^{-1}(r)}(1^\eta, a) \setminus B_{\mathcal{H}_r|Q}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a)$ says that the number of steps taken in the external protocol system Q exceeded q_1 , but not the number of steps in the internal systems. The event $B_2 := B_{\mathcal{H}_r|Q}^{q_1, \pi^{-1}(j)}(1^\eta, a) \setminus B_{\mathcal{H}_r|Q}^{q_1, \neq \pi^{-1}(j)}(1^\eta, a)$ says that the number of steps taken in the j -th copy of the internally simulated protocol system Q exceeded q_1 , but not the number of steps taken in the other protocol systems, i.e., in the other internal copies and the external system. As π is chosen uniformly at random, it does not make a difference whether q_1 is exceeded by the external system or one of the internal copies of Q .

Formally, equality (3.8) can easily be established by defining a bijection between the runs in B_1 and those in B_2 : a run $\rho \in B_1$ in which the permutation π was chosen is mapped to the corresponding run in B_2 where a permutation π' is chosen which coincides with π except that $\pi^{-1}(r)$ and $\pi^{-1}(j)$ are swapped, i.e., $\pi'^{-1}(r) = \pi^{-1}(j)$ and $\pi'^{-1}(j) = \pi^{-1}(r)$.

Now, by (3.8), we have that

$$\begin{aligned}
\Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a) \right] &\geq \Pr \left[\bigcup_{j=1}^{r-1} B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \pi^{-1}(j)}(1^\eta, a) \right] \\
&\geq \Pr \left[\bigcup_{j=1}^{r-1} B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \pi^{-1}(j)}(1^\eta, a) \setminus B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(j)}(1^\eta, a) \right] \\
&= \sum_{j=1}^{r-1} \Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \pi^{-1}(j)}(1^\eta, a) \setminus B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(j)}(1^\eta, a) \right] \\
&\stackrel{(3.8)}{=} (r-1) \cdot \Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \pi^{-1}(r)}(1^\eta, a) \setminus B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a) \right] .
\end{aligned} \tag{3.9}$$

We conclude that

$$\begin{aligned}
t_r &= \Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^{q_1}(1^\eta, a) \right] \\
&= \Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a) \right] + \Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \pi^{-1}(r)}(1^\eta, a) \setminus B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a) \right] \\
&\leq \frac{r}{r-1} \cdot \Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a) \right] && \text{because of (3.9)} \\
&\leq \frac{r}{r-1} \cdot \left(\Pr \left[B_{\mathcal{H}_r | \mathcal{P}}^{q_1, \neq \pi^{-1}(r)}(1^\eta, a) \right] + f'(\eta, a) \right) && \text{because of Lemma 3.13} \\
&\leq \frac{r}{r-1} \cdot \left(\Pr \left[B_{\mathcal{H}_r | \mathcal{P}}^{q_1}(1^\eta, a) \right] + f'(\eta, a) \right) .
\end{aligned}$$

As the systems $\mathcal{H}_r | \mathcal{P}$ and $\mathcal{H}_{r-1} | \mathcal{Q}$ behave exactly the same, we have that

$$\Pr \left[B_{\mathcal{H}_r | \mathcal{P}}^{q_1}(1^\eta, a) \right] = \Pr \left[B_{\mathcal{H}_{r-1} | \mathcal{Q}}^{q_1}(1^\eta, a) \right] = t_{r-1}$$

and obtain the following simple recurrence relation:

$$t_r \leq \frac{r}{r-1} \cdot (t_{r-1} + f'(\eta, a)) .$$

By induction on r , it can be shown that

$$t_r \leq \left(\prod_{j=1}^{r-1} \frac{j+1}{j} \right) \cdot t_1 + \left(\sum_{j=1}^{r-1} \prod_{i=j}^{r-1} \frac{i+1}{i} \right) \cdot f'(\eta, a) . \tag{3.10}$$

From this, we obtain for $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$:

$$\begin{aligned}
t_r &\leq \left(\prod_{j=1}^{r-1} \frac{j+1}{j} \right) \cdot t_1 + \left(\sum_{j=1}^{r-1} \prod_{i=j}^{r-1} \frac{i+1}{i} \right) \cdot f'(\eta, a) && \text{because of (3.10)} \\
&= r \cdot t_1 + \left(\sum_{j=1}^{r-1} \frac{r}{j} \right) \cdot f'(\eta, a) \\
&\leq r \cdot t_1 + (r-1) \cdot r \cdot f'(\eta, a) \\
&\leq p_{\mathcal{E}}(\eta, |a|) \cdot f_1(\eta, a) + p_{\mathcal{E}}^2(\eta, |a|) \cdot f'(\eta, a) && \text{because of Lemma 3.10.}
\end{aligned}$$

Hence, with

$$g(\eta, a) := p_{\mathcal{E}}(\eta, |a|) \cdot f_1(\eta, a) + p_{\mathcal{E}}^2(\eta, |a|) \cdot f'(\eta, a) ,$$

we obtain that

$$\Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^{q_1}(1^\eta, a) \right] = t_r \leq g(\eta, a) ,$$

for all $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$. Note that g is negligible and does not depend on r .

Now we still have to look at the case $r = 0$. By Lemma 3.10, we already know that $\Pr \left[B_{\mathcal{H}_0 | \mathcal{Q}}^{q_0}(1^\eta, a) \right] \leq f_0(\eta, a)$. We define the polynomial $q'(\eta, |a|) := q_0(\eta, |a|) + q_1(\eta, |a|)$. Observe that both the results for $r = 0$ and $1 \leq r \leq p_{\mathcal{E}}(\eta)$ also hold for the polynomial q' as every run of $\mathcal{H}_r | \mathcal{Q}(1^\eta, a)$ that exceeds the runtime bound $q'(\eta, |a|)$ also exceeds both runtime bounds $q_0(\eta, |a|)$ and $q_1(\eta, |a|)$ (note that q_0 and q_1 cannot be negative). Furthermore we define the negligible function $f''(\eta, a) = f_0(\eta, a) + g(\eta, a)$. Now the following holds true for all $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ by construction:

$$\Pr \left[B_{\mathcal{H}_r | \mathcal{Q}}^{q'}(1^\eta, a) \right] \leq f''(\eta, a) .$$

Note that f'' does not depend on the value of r . This concludes the proof of Lemma 3.14. \square

Lemma 3.14 immediately implies that $\Pr \left[B_{\mathcal{H}_{p_{\mathcal{E}}} | \mathcal{Q}}^{q'}(1^\eta, a) \right] \leq f''(\eta, a)$ for all $\eta \in \mathbb{N}$ and $a \in \{0, 1\}^*$. In particular, it follows that $\mathcal{H}_{p_{\mathcal{E}}} | \mathcal{Q}$ is almost bounded, and hence, Claim 3.1. By (3.5), this immediately implies that $\mathcal{E} | \mathcal{Q}$ is almost bounded (for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$).

Furthermore, Lemma 3.14 also implies that $\Pr \left[B_{\mathcal{H}_r | \mathcal{P}}^{q'}(1^\eta, a) \right] \leq f''(\eta, a)$ for all $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ as $\mathcal{H}_r | \mathcal{Q}$ and $\mathcal{H}_{r+1} | \mathcal{P}$ behave exactly the same (see Equation (3.4)). In fact, the only reason to include the case $r = 0$ in Lemma 3.14 is to obtain this result.

Having bounded the probability for exceeding a certain runtime, we will now proceed in the proof of Theorem 3.5 by showing that \mathcal{P} and \mathcal{Q} are indistinguishable by any $\mathcal{E} \in \text{Env}_R(\mathcal{P})$, i.e., $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{Q}$. For this we need just one more result: It is necessary to show that, for every polynomial q , all σ -single session responsive environments $[\mathcal{H}_r]_q$ (for $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$) can distinguish \mathcal{P} and \mathcal{Q} with only roughly the same probability. Note that this is stronger than

what we have already shown in the proof of Lemma 3.10 (i.e., $[\mathcal{H}_r]_q \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$ which implies that $[\mathcal{H}_r]_q | \mathcal{P} \equiv [\mathcal{H}_r]_q | \mathcal{Q}$ by assumption) because we have to find a single negligible function that holds for all possible values of r .

Lemma 3.15. *Let q be a polynomial. There exists a negligible function $g_{3,q}$ such that the following holds true for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$:*

$$|\Pr [[\mathcal{H}_r]_q | \mathcal{P}(1^\eta, a) = 1] - \Pr [[\mathcal{H}_r]_q | \mathcal{Q}(1^\eta, a) = 1]| \leq g_{3,q}(\eta, a).$$

Proof. Just as in Lemmas 3.11 and 3.13, we prove this lemma for the case of non-uniform environments. Again, using the same technique as in [99] (see Appendix B), one easily obtains a proof for uniform environments.

This proof follows essentially the same idea as the proofs of Lemma 3.11 and Lemma 3.13. We define an IITM $D \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$ which expects to receive (r, a) as external input and then simulates $[\mathcal{H}_r]_q$ with external input a . The IITM D outputs 1 on tape **decision** if and only if the simulated $[\mathcal{H}_r]_q$ wants to output 1 on tape **decision**.

More formally, D is defined as follows. In mode **CheckAddress**, D accepts every message. In mode **Compute**, D behaves as follows: First, D parses the external input on **start** as (r, a) , with $r \in \{1, \dots, p_{\mathcal{E}}(\eta, |a|)\}$ and $a \in \{0, 1\}^*$. If the external input is not of this form, D outputs 0 on **decision**. Otherwise, D simulates the system $[\mathcal{H}_r]_q$ with external input a . If $[\mathcal{H}_r]_q$ produces output, then so does D and if D receives input, then D forwards this input to the simulated $[\mathcal{H}_r]_q$. If the run terminates (i.e., $[\mathcal{H}_r]_q$ outputs something on **decision** or $[\mathcal{H}_r]_q$ produces empty output, which terminates the run because $[\mathcal{H}_r]_q$ is a master IITM), then D terminates its run in the same way (i.e., it either outputs the same message on tape **decision** or produces empty output).

It is easy to see that D is universally bounded (i.e. $D \in \text{Env}(\mathcal{P})$) and that it is σ -single session. By Lemma 3.12, we also know that $\Pr [C_{[\mathcal{H}_r]_q}^{\mathcal{P}}(1^\eta, a)] \leq g_{2,q}(\eta, a)$ for all $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$. By this, we have that $\Pr [C_D^{\mathcal{P}}(1^\eta, a')] \leq \max_{a \in \{0,1\}^{\leq |a'|}} g_{2,q}(\eta, a)$, which is negligible, and thus $D \in \text{Env}_R(\mathcal{P})$. Overall, this implies that $D \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$.

We have for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$:

$$\begin{aligned} \Pr [(D | \mathcal{P})(1^\eta, (r, a)) = 1] &= \Pr [[\mathcal{H}_r]_q | \mathcal{P}(1^\eta, a) = 1] \quad \text{and} \\ \Pr [(D | \mathcal{Q})(1^\eta, (r, a)) = 1] &= \Pr [[\mathcal{H}_r]_q | \mathcal{Q}(1^\eta, a) = 1] \quad . \end{aligned} \tag{3.11}$$

As $\mathcal{E}' | \mathcal{P} \equiv \mathcal{E}' | \mathcal{Q}$ for all $\mathcal{E}' \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$ and $D \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$, we have that $D | \mathcal{P} \equiv D | \mathcal{Q}$. In particular, there exists a negligible function $g'_{3,q}$ such that for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$,

and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$:

$$\begin{aligned} g'_{3,q}(\eta, (r, a)) &\geq |\Pr[(D | \mathcal{P})(1^\eta, (r, a)) = 1] - \Pr[(D | \mathcal{Q})(1^\eta, (r, a)) = 1]| \\ &\stackrel{(3.11)}{=} |\Pr[[\mathcal{H}_r]_q | \mathcal{P}(1^\eta, a) = 1] - \Pr[[\mathcal{H}_r]_q | \mathcal{Q}(1^\eta, a) = 1]| . \end{aligned}$$

Let $g_{3,q}(\eta, a) := \max_{r \leq p_{\mathcal{E}}(\eta, |a|)} g'_{3,q}(\eta, (r, a))$. It is easy to see that $g_{3,q}$ is a negligible function. Now, obviously we have that

$$|\Pr[[\mathcal{H}_r]_q | \mathcal{P}(1^\eta, a) = 1] - \Pr[[\mathcal{H}_r]_q | \mathcal{Q}(1^\eta, a) = 1]| \leq g_{3,q}(\eta, a)$$

for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$. This concludes the proof of Lemma 3.15. \square

We can now conclude the proof of Theorem 3.5. First we observe that there exists a single polynomial q that bounds the runtime of \mathcal{E}_r (in every run) for all $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$. This is because every \mathcal{E}_r just simulates \mathcal{E} , which is universally bounded, and because $r \leq p_{\mathcal{E}}(\eta, |a|)$.

We define the polynomial $q''(\eta, |a|) := q(\eta, |a|) + p_{\mathcal{E}}(\eta, |a|) \cdot q'(\eta, |a|)$, where q is the polynomial from the last paragraph and q' is the polynomial from Lemma 3.14. Now we have that, for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$, the system \mathcal{H}_r in a run of $\mathcal{H}_r | \mathcal{P}(1^\eta, a)$ will exceed the runtime $q''(\eta, |a|)$ in mode **Compute** only if at least one copy of \mathcal{P}'' or \mathcal{Q}' exceeds the runtime $q'(\eta, |a|)$. That is, every run of $\mathcal{H}_r | \mathcal{P}(1^\eta, a)$ in which \mathcal{H}_r needs more than $q''(\eta, |a|)$ steps is in $B_{\mathcal{H}_r | \mathcal{P}}^{q'}(1^\eta, a)$. Analogously, we have that runs of $\mathcal{H}_r | \mathcal{Q}(1^\eta, a)$ in which \mathcal{H}_r exceeds the runtime $q''(\eta, |a|)$ must be in $B_{\mathcal{H}_r | \mathcal{Q}}^{q'}(1^\eta, a)$.

As $\Pr[B_{\mathcal{H}_r | \mathcal{P}}^{q'}(1^\eta, a)] \leq f''(\eta, a)$ and $\Pr[B_{\mathcal{H}_r | \mathcal{Q}}^{q'}(1^\eta, a)] \leq f''(\eta, a)$ for all $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ (by Lemma 3.14 and the remark following this lemma), we can conclude that the systems $\mathcal{H}_r | \mathcal{P}$ and $[\mathcal{H}_r]_{q''} | \mathcal{P}$ as well as $\mathcal{H}_r | \mathcal{Q}$ and $[\mathcal{H}_r]_{q''} | \mathcal{Q}$ only differ in a negligible set of runs. More formally, for all $\eta \in \mathbb{N}$, $a \in \{0, 1\}^*$, and $1 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$, we have

$$\begin{aligned} |\Pr[\mathcal{H}_r | \mathcal{P}(1^\eta, a) = 1] - \Pr[[\mathcal{H}_r]_{q''} | \mathcal{P}(1^\eta, a) = 1]| &\leq f''(\eta, a) \text{ and} \\ |\Pr[\mathcal{H}_r | \mathcal{Q}(1^\eta, a) = 1] - \Pr[[\mathcal{H}_r]_{q''} | \mathcal{Q}(1^\eta, a) = 1]| &\leq f''(\eta, a) . \end{aligned} \tag{3.12}$$

By this, we obtain the following inequalities, which hold true for all $\eta \in \mathbb{N}$ and $a \in \{0, 1\}^*$:

$$\begin{aligned} &|\Pr[\mathcal{E} | \mathcal{P}(1^\eta, a) = 1] - \Pr[\mathcal{E} | \mathcal{Q}(1^\eta, a) = 1]| \\ &\stackrel{(3.3), (3.5)}{=} |\Pr[\mathcal{H}_1 | \mathcal{P}(1^\eta, a) = 1] - \Pr[\mathcal{H}_{p_{\mathcal{E}}} | \mathcal{Q}(1^\eta, a) = 1]| \\ &\leq \sum_{r=1}^{p_{\mathcal{E}}(\eta, |a|)} |\Pr[\mathcal{H}_r | \mathcal{P}(1^\eta, a) = 1] - \Pr[\mathcal{H}_r | \mathcal{Q}(1^\eta, a) = 1]| \end{aligned}$$

$$\begin{aligned}
&\leq \sum_{r=1}^{p_{\mathcal{E}}(\eta, |a|)} |\Pr[\mathcal{H}_r \mid \mathcal{P}(1^\eta, a) = 1] - \Pr[[\mathcal{H}_r]_{q''} \mid \mathcal{P}(1^\eta, a) = 1]| \\
&+ \sum_{r=1}^{p_{\mathcal{E}}(\eta, |a|)} |\Pr[[\mathcal{H}_r]_{q''} \mid \mathcal{P}(1^\eta, a) = 1] - \Pr[[\mathcal{H}_r]_{q''} \mid \mathcal{Q}(1^\eta, a) = 1]| \\
&+ \sum_{r=1}^{p_{\mathcal{E}}(\eta, |a|)} |\Pr[[\mathcal{H}_r]_{q''} \mid \mathcal{Q}(1^\eta, a) = 1] - \Pr[\mathcal{H}_r \mid \mathcal{Q}(1^\eta, a) = 1]| \\
(3.12), \text{ Lemma 3.15} &\leq \sum_{r=1}^{p_{\mathcal{E}}(\eta, |a|)} (f''(\eta, a) + g_{3, q''}(\eta, a) + f''(\eta, a)) \\
&= p_{\mathcal{E}}(\eta, |a|) \cdot (f''(\eta, a) + g_{3, q''}(\eta, a) + f''(\eta, a))
\end{aligned}$$

We define $f'''(\eta, a) := p_{\mathcal{E}}(\eta, |a|) \cdot (f''(\eta, a) + g_{3, q''}(\eta, a) + f''(\eta, a))$. It is easy to see that f''' is negligible and that the following holds true:

$$|\Pr[\mathcal{E} \mid \mathcal{P}(1^\eta, a) = 1] - \Pr[\mathcal{E} \mid \mathcal{Q}(1^\eta, a) = 1]| \leq f'''(\eta, a) .$$

This implies $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{Q}$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$.

To conclude the proof of Theorem 3.5, it remains to show that \mathcal{Q} is R-environmentally bounded. As \mathcal{P} and \mathcal{Q} are indistinguishable for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$, we can apply Lemma 3.1 to obtain $\text{Env}_R(\mathcal{P}) = \text{Env}_R(\mathcal{Q})$. We have already shown that $\mathcal{E} \mid \mathcal{Q}$ is almost bounded for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ by Lemma 3.14 (see the comment following the lemma) and thus $\mathcal{E} \mid \mathcal{Q}$ is almost bounded for all $\mathcal{E} \in \text{Env}_R(\mathcal{Q})$, i.e., \mathcal{Q} is R-environmentally bounded.

This concludes the proof of Theorem 3.5. \square

The following lemma corresponds to Lemma 14 in [99], but has been adapted to the case of responsive environments. With this lemma, we can then finally prove Theorem 3.4.

Lemma 3.16. *Let σ be a PSID function and let \mathcal{P} and \mathcal{F} be protocol systems such that \mathcal{P} and \mathcal{F} are σ -session versions and $\mathcal{P} \leq_{R, \sigma\text{-single}} \mathcal{F}$. Then, there exists $\mathcal{S} \in \text{Sim}_{R, \sigma\text{-single}}^{\mathcal{P}}(\mathcal{F})$ such that $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$ and \mathcal{S} is a single IITM which is σ -complete.*

Proof. The original proof uses that a simulator $\mathcal{S} \in \text{Sim}_{\sigma\text{-single}}^{\mathcal{P}}(\mathcal{F})$ exists such that $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_{\sigma\text{-single}}(\mathcal{P})$. Then one can construct a new simulator \mathcal{S}' that is defined to be σ -complete and essentially simulates \mathcal{S} and some copies of \mathcal{F} which only interact with \mathcal{S} (except for a negligible set of runs). It is then shown that, by definition, \mathcal{S}' is a σ -single session simulator and $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{S}' \mid \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_{\sigma\text{-single}}(\mathcal{P})$.

Now, if one uses a simulator $\mathcal{S} \in \text{Sim}_{R, \sigma\text{-single}}^{\mathcal{P}}(\mathcal{F})$ with $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$, then it follows with the same construction that $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{S}' \mid \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ and that \mathcal{S}' fulfills the conditions of σ -single session responsive simulators regarding interfaces and runtime. Furthermore, as $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ and $\mathcal{E} \mid \mathcal{S}' \mid \mathcal{F}$ essentially behave the same for all $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$ (except for a negligible set of runs), there must only be a negligible set of runs of $\mathcal{E} \mid \mathcal{S}' \mid \mathcal{F}$ in which a restricting message of \mathcal{F} is answered incorrectly (if this were not the case, then there

would also be a non-negligible set of runs of $\mathcal{E} | \mathcal{S} | \mathcal{F}$ in which a restricting message is answered incorrectly, which contradicts $\mathcal{S} \in \text{Sim}_{R, \sigma\text{-single}}^{\mathcal{P}}(\mathcal{F})$. This implies $\mathcal{S}' \in \text{Sim}_{R, \sigma\text{-single}}^{\mathcal{P}}(\mathcal{F})$. \square

We can now prove Theorem 3.4:

Proof of Theorem 3.4. By Lemma 3.16, there exists $\mathcal{S} \in \text{Sim}_{R, \sigma\text{-single}}^{\mathcal{P}}(\mathcal{F})$ such that \mathcal{S} is a σ -complete IITM and $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{S} | \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$. As \mathcal{S} is σ -complete, we can conclude that $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | !\mathcal{S} | \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$:¹⁷ Because the environment \mathcal{E} invokes only a single session, i.e. sends only messages to \mathcal{S} and \mathcal{F} with the same PSID (w.r.t. σ), and \mathcal{F} is a σ -session version, \mathcal{S} receives only messages with the same PSID (w.r.t. σ) from \mathcal{F} and \mathcal{E} . So, even with $!\mathcal{S}$ only one instance of \mathcal{S} will be invoked in every run of $\mathcal{E} | !\mathcal{S} | \mathcal{F}$, which implies $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{S} | \mathcal{F} \equiv \mathcal{E} | !\mathcal{S} | \mathcal{F}$.

Next, we observe that $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ is almost bounded for all $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(!\mathcal{S} | \mathcal{F})$: As $\mathcal{E} | \mathcal{S} | \mathcal{F} \equiv \mathcal{E} | \mathcal{P} \equiv \mathcal{E} | !\mathcal{S} | \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{P})$, we can use Lemma 3.8 twice to obtain $\text{Env}_{R, \sigma\text{-single}}(\mathcal{S} | \mathcal{F}) = \text{Env}_{R, \sigma\text{-single}}(\mathcal{P}) = \text{Env}_{R, \sigma\text{-single}}(!\mathcal{S} | \mathcal{F})$. Because \mathcal{S} is a σ -single session responsive simulator, the system $\mathcal{E} | \mathcal{S} | \mathcal{F}$ is almost bounded for all $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(\mathcal{S} | \mathcal{F}) = \text{Env}_{R, \sigma\text{-single}}(!\mathcal{S} | \mathcal{F})$. As argued above we can easily define a bijection between runs of $\mathcal{E} | \mathcal{S} | \mathcal{F}$ and runs of $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ for all σ -single session environments, and thus $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ must also be almost bounded for all $\mathcal{E} \in \text{Env}_{R, \sigma\text{-single}}(!\mathcal{S} | \mathcal{F})$.

Finally, we note that, by construction, $!\mathcal{S} | \mathcal{F}$ is a σ -session version in which every machine that does not occur in the scope of a bang accepts all messages in mode `CheckAddress`. Thus, we can use Theorem 3.5 (with $\mathcal{Q} := !\mathcal{S} | \mathcal{F}$) to obtain $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | !\mathcal{S} | \mathcal{F}$ for all $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ and that $!\mathcal{S} | \mathcal{F}$ is R-environmentally bounded.

Let $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ (and thus, by Lemma 3.1, $\mathcal{E} \in \text{Env}_R(!\mathcal{S} | \mathcal{F})$). The remainder of the proof shows that the set of all runs of $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ in which a restricting message from \mathcal{F} is answered incorrectly is negligible; this implies $!\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$ and thus $\mathcal{P} \leq_R \mathcal{F}$. We will first show that there exists a single negligible function which bounds the probability of an incorrectly answered restricting message of a copy of \mathcal{F} for all possible copies. As there are only polynomially many copies in any run, we can then construct another negligible function which bounds the probability that any copy of \mathcal{F} receives an incorrect answer to a restricting message.

By Lemma 6 from [99], we may assume that \mathcal{E} is a single IITM which, in mode `CheckAddress`, accepts all messages. We may also assume that `start` is a tape of \mathcal{E} (otherwise there would be no master IITM in $\mathcal{E} | !\mathcal{S} | \mathcal{F}$, i.e., every run immediately ends with empty output. Thus, the claim holds true trivially). In addition, we may assume that the only external tapes of $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ are `start` and `decision`. Moreover, we may assume, again without loss of generality, that \mathcal{E} is such that every message m that \mathcal{E} outputs on tape c (except if m is output on tape `decision`) has some PSID, i.e. $\sigma(m, c) \neq \perp$: As \mathcal{E} will only interact with σ -session versions, messages without a PSID would be rejected by these σ -session versions anyway. As \mathcal{E} is universally bounded, it follows that there exists a polynomial $p_{\mathcal{E}}$ such that the number of different sessions

¹⁷Recall that ‘!’ binds more strongly than ‘|’, and hence, the system $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ is the same as $\mathcal{E} | (!\mathcal{S}) | \mathcal{F}$.

(i.e., messages with distinct PSIDs output by \mathcal{E}) is bounded from above by $p_{\mathcal{E}}(\eta, |a|)$ (where η is the security parameter and a is the external input).

In what follows, let \mathcal{S}' be the variant of \mathcal{S} obtained from \mathcal{S} by renaming every tape c occurring in \mathcal{S} to c' . Analogously, let \mathcal{F}' be obtained from \mathcal{F} by renaming every tape c occurring in \mathcal{F} to c' . By this, we have that $!\mathcal{S}|\mathcal{F}$ and $!\mathcal{S}'|\mathcal{F}'$ have pairwise disjoint sets of external tapes, and hence, these systems are pairwise connectable.

We now define an IITM \mathcal{E}_r (for every $r \in \mathbb{N}$) which basically simulates \mathcal{E} and which will run in the system $\mathcal{E}_r|!\mathcal{S}'|\mathcal{F}'|!\mathcal{S}|\mathcal{F}$. The r -th copy of the protocol invoked by (the simulated) \mathcal{E} will be a copy of $!\mathcal{S}|\mathcal{F}$, and the remaining copies will be copies of $!\mathcal{S}'|\mathcal{F}'$.

Formally, \mathcal{E}_r is obtained from \mathcal{E} as follows. The IITM \mathcal{E} has tapes to connect to all external tapes of $!\mathcal{S}|\mathcal{F}$. \mathcal{E}_r has the same tapes. Furthermore, for each such tape c , we add another tapes c' to \mathcal{E}_r to connect to the external tapes of $!\mathcal{S}'|\mathcal{F}'$. Just as \mathcal{E} , the IITM \mathcal{E}_r will also always accept every message in mode `CheckAddress`. The behavior of \mathcal{E}_r in mode `Compute` is specified next.

We need to specify how \mathcal{E}_r redirects protocol invocations of \mathcal{E} in the way sketched above: \mathcal{E}_r keeps a list L of PSIDs, which initially is empty, and the length l of the list, which initially is 0. By definition of $p_{\mathcal{E}}$, it will always hold that $l \leq p_{\mathcal{E}}(\eta, |a|)$. After the first activation, \mathcal{E}_r simulates \mathcal{E} with security parameter η and external input a . In particular, if \mathcal{E} produces output, then so does \mathcal{E}_r , and if \mathcal{E}_r receives input, then \mathcal{E} is simulated with this input. However, as explained next, the behavior of \mathcal{E}_r deviates from that of \mathcal{E} when sending and receiving messages to the different copies of protocols.

1. If \mathcal{E} produces output m on some external tape c of $!\mathcal{S}|\mathcal{F}$ with $s := \sigma(m, c)$, then \mathcal{E}_r checks whether s occurs in L . If s does not occur in L , s is first appended at the end of L and l is increased by 1. Let $j \in \{1, \dots, l\}$ be the position at which s occurs in L .
 - a) If $j = r$, then \mathcal{E}_r writes m on tape c .
 - b) If $j \neq r$, then \mathcal{E}_r writes m on tape c' .
2. If \mathcal{E}_r receives input on tape c' (where c' is an external tape of $!\mathcal{S}'|\mathcal{F}'$ corresponding to an external tape c of $!\mathcal{S}|\mathcal{F}$), then \mathcal{E}_r behaves as \mathcal{E} does when input is received on tape c .
3. If \mathcal{E}_r receives input on tape c (where c is an external tape of $!\mathcal{S}|\mathcal{F}$), then \mathcal{E}_r behaves as \mathcal{E} does when input is received on tape c .

It is easy to see that \mathcal{E}_r is universally bounded for every $r \in \mathbb{N}$ because \mathcal{E} is universally bounded.

We now define a system \mathcal{E}_{\S} which first chooses some $r \in \{1, \dots, p_{\mathcal{E}}(\eta, |a|)\}$ uniformly at random and then behaves exactly as \mathcal{E}_r . It is easy to see that \mathcal{E}_{\S} is universally bounded as \mathcal{E}_{\S} simulates the universally bounded system \mathcal{E}_r (whose runtime bound is the same for all valid r) and r is bounded by a polynomial. Furthermore, we define the system $\mathcal{H}_{\S} = \mathcal{E}_{\S}|!\mathcal{S}'|\mathcal{F}'$ for brevity of presentation.

By construction, the systems $\mathcal{E}|!\mathcal{S}|\mathcal{F}$ and $\mathcal{H}_{\S}|!\mathcal{S}|\mathcal{F}$ behave exactly the same (see below). That is, there exists a mapping which maps a run ρ of $\mathcal{E}|!\mathcal{S}|\mathcal{F}$ to a set of corresponding runs

of $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$ such that the probability of ρ equals the probability of the corresponding set of runs of $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$, where the runs in that set differ only in the choice of r . In particular, we have that

$$\mathcal{E} | !\mathcal{S} | \mathcal{F} \equiv \mathcal{H}_\S | !\mathcal{S} | \mathcal{F} , \quad (3.13)$$

For (3.13), we need that $!\mathcal{S} | \mathcal{F}$ is a protocol system and a σ -session version. The second property is necessary as it guarantees that two instances of machines in $!\mathcal{S} | \mathcal{F}$ with different PSIDs will never send messages to each other (as they only send messages with their own PSID), i.e., it is no problem that the systems $!\mathcal{S}' | \mathcal{F}'$ and $!\mathcal{S} | \mathcal{F}$ in $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$ do not have a connection to each other. Furthermore, for machines M of $!\mathcal{S} | \mathcal{F}$ that are in the scope of a bang, it is easy to see that every instance of such a machine in a run of $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ corresponds to one instance of this machine (either in $!\mathcal{S}' | \mathcal{F}'$ or $!\mathcal{S} | \mathcal{F}$, depending on the PSID of the instance) in a run of $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$.

If there is a machine M in $!\mathcal{S} | \mathcal{F}$ that is not in the scope of a bang, the following problem can occur: In a run of $\mathcal{E} | !\mathcal{S} | \mathcal{F}$, there can only be at most one instance of such a machine, whereas there may be two instances (one in $!\mathcal{S}' | \mathcal{F}'$ and one in $!\mathcal{S} | \mathcal{F}$) in a run of $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$, i.e., one of those instances would not have a corresponding instance of M in $\mathcal{E} | !\mathcal{S} | \mathcal{F}$. This is where we need that $!\mathcal{S} | \mathcal{F}$ is a protocol system, i.e., such a machine M must accept all messages m on all tapes c in mode `CheckAddress`. As M is a σ -session version, i.e., accepts only messages with the same PSID, we obtain $\sigma(m, c) = s$ for some fixed value $s \neq \perp$, $m \in \{0, 1\}^*$, and all (input-)tapes c of M . Because \mathcal{E}_r and thus \mathcal{E}_\S never send messages with the same PSID to both $!\mathcal{S}' | \mathcal{F}'$ and $!\mathcal{S} | \mathcal{F}$, there will be at most one copy of either $!\mathcal{S}' | \mathcal{F}'$ or $!\mathcal{S} | \mathcal{F}$ with PSID s . But then there is also at most one instance of M in a run of $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$, which corresponds to the single instance of M in a run of $\mathcal{E} | !\mathcal{S} | \mathcal{F}$.

Recall that we have to show that restricting messages of \mathcal{F} are answered correctly in an overwhelming set of runs of $\mathcal{E} | !\mathcal{S} | \mathcal{F}$. For this, we need the following events: Recall from the proof of Lemma 3.5 that, for all systems \mathcal{Q} and \mathcal{R} , we define the event $C_{\mathcal{Q}}^{\mathcal{R}} = C_{\mathcal{Q}}^{\mathcal{R}}(1^\eta, a)$ to be the set of all runs of $\mathcal{Q} | \mathcal{R}$ in which a restricting message of \mathcal{R} is answered incorrectly. With $\overline{C_{\mathcal{Q}}^{\mathcal{R}}}$ we denote the complementary event.

In the following, we will first prove that $\Pr \left[C_{\mathcal{H}_\S | !\mathcal{S}}^{\mathcal{F}} \right]$ is negligible, which can then be used to conclude that $\Pr \left[C_{\mathcal{E} | !\mathcal{S}}^{\mathcal{F}} \right]$ is negligible. For this result, we need that $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$ is almost bounded: Observe that $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ is almost bounded, because $\mathcal{E} \in \text{Env}_R(!\mathcal{S} | \mathcal{F})$ and $!\mathcal{S} | \mathcal{F}$ is R-environmentally bounded. Because $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$ and $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ behave exactly the same (see (3.13) and the explanation above that equation), it is easy to see that also $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$ is almost bounded.

It also holds true that $\mathcal{E}_\S \in \text{Env}_R(!\mathcal{S}' | \mathcal{F}' | !\mathcal{S} | \mathcal{F})$: As the systems $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$ and $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ have exactly the same, we have that in a run ρ of $\mathcal{E} | !\mathcal{S} | \mathcal{F}$ a restricting message of $!\mathcal{S} | \mathcal{F}$ is not answered correctly iff in every run in the set of runs of $\mathcal{H}_\S | !\mathcal{S} | \mathcal{F}$ corresponding to ρ this restricting message, which comes from $!\mathcal{S} | \mathcal{F}$ or $!\mathcal{S}' | \mathcal{F}'$, is also not answered correctly. As $\Pr \left[\overline{C_{\mathcal{E}}^{!\mathcal{S} | \mathcal{F}}} \right]$

is overwhelming (by $\mathcal{E} \in \text{Env}_R(!\mathcal{S}|\mathcal{F})$), this directly implies that $\Pr \left[\overline{C_{\mathcal{E}_\S}^{!\mathcal{S}'|\mathcal{F}'|!\mathcal{S}|\mathcal{F}}} \right]$ is also overwhelming and thus $\mathcal{E}_\S \in \text{Env}_R(!\mathcal{S}'|\mathcal{F}'|!\mathcal{S}|\mathcal{F})$.

Now observe that every restricting message (on an external network tape) of $!\mathcal{S}|\mathcal{F}$ in the system $\mathcal{H}_\S|!\mathcal{S}|\mathcal{F}$ is also a restricting message of $!\mathcal{S}'|\mathcal{F}'|!\mathcal{S}|\mathcal{F}$ (note that $!\mathcal{S}|\mathcal{F}$ and $!\mathcal{S}'|\mathcal{F}'$ have disjoint tapes). Thus we have $\Pr \left[C_{\mathcal{H}_\S}^{!\mathcal{S}|\mathcal{F}} \right] = \Pr \left[C_{\mathcal{E}_\S|!\mathcal{S}'|\mathcal{F}'}^{!\mathcal{S}|\mathcal{F}} \right] \leq \Pr \left[C_{\mathcal{E}_\S}^{!\mathcal{S}'|\mathcal{F}'|!\mathcal{S}|\mathcal{F}} \right]$, i.e., $\Pr \left[C_{\mathcal{H}_\S}^{!\mathcal{S}|\mathcal{F}} \right]$ must also be negligible.

By the previous observations, we can use Lemma 3.2 and Lemma 3.4 to obtain that $[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}} \in \text{Env}_R(!\mathcal{S}|\mathcal{F})$. Furthermore, $[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}}$ is also a σ -single session environment, as the only external tapes (except for **start** and **decision**) of $[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}}$ are those between (the simulated) \mathcal{E}_r and $!\mathcal{S}|\mathcal{F}$, and \mathcal{E}_r only sends messages with the same PSID on those tapes. Overall, we have that $[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}} \in \text{Env}_{R,\sigma\text{-single}}(!\mathcal{S}|\mathcal{F})$, and thus, by Lemma 3.8, $[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}} \in \text{Env}_{R,\sigma\text{-single}}(\mathcal{P})$. As $\mathcal{S} \in \text{Sim}_{R,\sigma\text{-single}}^{\mathcal{P}}(\mathcal{F})$, this implies that $\Pr \left[C_{[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}}|\mathcal{S}}^{\mathcal{F}} \right] = \Pr \left[C_{[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}}|!\mathcal{S}}^{\mathcal{F}} \right]$ is negligible. (Recall that $[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}}$ is a σ -single session environment and that \mathcal{S} is σ -complete. Hence, the systems $[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}}|\mathcal{S}$ and $[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}}|!\mathcal{S}$ behave in exactly the same way.) Finally, because $[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}}|!\mathcal{S}|\mathcal{F}$ and $\mathcal{H}_\S|!\mathcal{S}|\mathcal{F}$ behave exactly the same except for a negligible set of runs (i.e., when the runtime bound of $[\mathcal{H}_\S]_{!\mathcal{S}|\mathcal{F}}$ is reached), we can conclude that $\Pr \left[C_{\mathcal{H}_\S|!\mathcal{S}}^{\mathcal{F}} \right]$ is negligible.

We can now show that $\Pr \left[C_{\mathcal{E}|!\mathcal{S}}^{\mathcal{F}} \right]$ is negligible. First we observe that, for every run ρ in $C_{\mathcal{E}|!\mathcal{S}}^{\mathcal{F}}(1^\eta, a)$, there exists a unique run ρ' in $C_{\mathcal{H}_\S|!\mathcal{S}}^{\mathcal{F}}(1^\eta, a)$ with $\Pr[\rho'] = \frac{1}{p_{\mathcal{E}}(\eta, |a|)} \cdot \Pr[\rho]$. Note the difference between both events: $C_{\mathcal{E}|!\mathcal{S}}^{\mathcal{F}}$ contains runs where multiple sessions of \mathcal{F} exist, i.e., any one of those sessions might have sent a restricting messages that was not answered correctly. In $C_{\mathcal{H}_\S|!\mathcal{S}}^{\mathcal{F}}$, there is only one session of \mathcal{F} , whereas all other sessions are part of \mathcal{H}_\S . So that one specific session of \mathcal{F} has sent a restricting message that was answered incorrectly.

More formally, we can define an injective function f that maps runs ρ from $C_{\mathcal{E}|!\mathcal{S}}^{\mathcal{F}}$ to runs ρ' of $C_{\mathcal{H}_\S|!\mathcal{S}}^{\mathcal{F}}$ in the following way: Let ρ be such a run. Then there exists at least one restricting message of \mathcal{F} that is answered incorrectly. Let s be the PSID of the instance of \mathcal{F} (in the run ρ) which is the first instance that receives an incorrect answer. Let r' be such that s is the r' -th (unique) PSID that is used by \mathcal{E} . We define ρ' to be the run of $\mathcal{H}_\S|!\mathcal{S}|\mathcal{F}$, where \mathcal{H}_\S chooses $r = r'$ (which happens with probability $p_{\mathcal{E}}(\eta, |a|)^{-1}$) and the run then continues just as ρ , i.e., all instances in ρ' , after r has been chosen, use the same random coins as those in ρ . Then ρ' is in $C_{\mathcal{H}_\S|!\mathcal{S}}^{\mathcal{F}}$ by construction. Furthermore, it is easy to see that f is injective and that $\Pr[\rho'] = \frac{1}{p_{\mathcal{E}}(\eta, |a|)} \cdot \Pr[\rho]$. Now, we obtain:

$$\begin{aligned} \Pr \left[C_{\mathcal{E}|!\mathcal{S}}^{\mathcal{F}} \right] &= \sum_{\rho \in C_{\mathcal{E}|!\mathcal{S}}^{\mathcal{F}}} \Pr[\rho] \\ &= \sum_{\rho \in C_{\mathcal{E}|!\mathcal{S}}^{\mathcal{F}}} p_{\mathcal{E}}(\eta, |a|) \cdot \Pr[f(\rho)] \\ &\stackrel{f \text{ injective}}{\leq} \sum_{\rho' \in C_{\mathcal{H}_\S|!\mathcal{S}}^{\mathcal{F}}} p_{\mathcal{E}}(\eta, |a|) \cdot \Pr[\rho'] \end{aligned}$$

$$= p_{\mathcal{E}}(\eta, |a|) \cdot \Pr \left[C_{\mathcal{H}_{\mathfrak{s}}^{\mathcal{F}}} | !\mathcal{S} \right] .$$

As $\Pr \left[C_{\mathcal{H}_{\mathfrak{s}}^{\mathcal{F}}} | !\mathcal{S} \right]$ is negligible, it follows that also $\Pr \left[C_{\mathcal{E}' | !\mathcal{S}}^{\mathcal{F}} \right]$ is bounded by a negligible function. This implies $!\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$, and thus, concludes the proof of Theorem 3.4. \square

3.2.9. Combining Security Results for Different Restrictions

All of our lemmas and theorems, including transitivity of the \leq_R relation and the composition theorems, have been proven using only a single restriction R . In other words, to be able to actually use these lemmas and theorems, a protocol designer has to reuse the same restriction in every analysis. However, sometimes it is desirable to extend a restriction, for example, because one wants to introduce a new type of restricting message that is necessary to model a specific protocol, or combine the results of two different security analyses that used different restrictions. This section first shows that extending restrictions is indeed possible, and then shows how this can be used to combine the results of two security analyses that used different restrictions.

We first show a lemma that allows us to extend restrictions while still retaining security results. The general idea is as follows: Observe that there are two major constraints in the security notion (cf. Definition 3.18), namely, indistinguishability for all responsive environments and responsiveness of the simulator. To be able to use a restriction R' instead of R while still retaining security results, we have to preserve both constraints. The first can be preserved if R' only restricts responsive environments further, hence yielding a subset of responsive environments and adversaries for R . The second can be preserved if the ideal protocol simply never sends one of the new restricting messages that were added by R' , as then there are no runs where the simulator answers one of these new messages incorrectly.

Lemma 3.17. *Let R be a restriction and let \mathcal{P} and \mathcal{F} be protocols such that $\mathcal{P} \leq_R \mathcal{F}$. Let R' be a restriction that extends R , i.e., $R \subseteq R'$. Then we have that $\mathcal{P} \leq_{R'} \mathcal{F}$ if the following holds true:*

- $\{(m, m') \mid m \in R[0], (m, m') \in R' \setminus R\} = \emptyset$, that is, R' does not add new allowed answers to any restricting message of $R[0]$, and
- for all systems \mathcal{Q} , the protocol \mathcal{F} in runs of $\mathcal{Q} | \mathcal{F}$ never outputs a message $m \in R'[0] \setminus R[0]$ on the network interface.

Proof. Assume that the requirements of the lemma are met. Let \mathcal{S} be the simulator that yields $\mathcal{P} \leq_R \mathcal{F}$. We have to show that (1) $\mathcal{E} | \mathcal{P} \equiv \mathcal{E} | \mathcal{S} | \mathcal{F}$ for every environment $\mathcal{E} \in \text{Env}_{R'}(\mathcal{P})$ and that (2) $\mathcal{S} \in \text{Sim}_{R'}^{\mathcal{P}}(\mathcal{F})$.

For (1), observe that we have $R[0] \subseteq R'[0]$ due to $R \subseteq R'$, i.e., R' only extends the set of restricting messages. Furthermore, by assumption R' does not add new possible answers to any restricting message already defined in $R[0]$. Therefore the restriction R' actually restricts responsive environments and adversaries more than R , i.e., any environment in $\text{Env}_{R'}(\mathcal{P})$ is

also in $\text{Env}_R(\mathcal{P})$. Because we have $\mathcal{E} \mid \mathcal{P} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ for every environment $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ by the definition of $\mathcal{P} \leq_R \mathcal{F}$, (1) directly follows.

For (2), let $\mathcal{E} \in \text{Env}_{R'}(\mathcal{P})$. Observe that $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ by the same reasoning as in the above paragraph. Now first suppose there was a non-negligible set of runs of $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ where \mathcal{S} answers a restricting message $m \in R[0]$ incorrectly according to the restriction R' (note that this does not consider restricting messages $m \in R'[0] \setminus R[0]$). Because $R \subseteq R'$, these answers would also be incorrect according to the restriction R . This contradicts the responsiveness of \mathcal{S} (for R) because $\mathcal{E} \in \text{Env}_R(\mathcal{P})$.

Finally observe that \mathcal{F} will never send a restricting message $m \in R'[0] \setminus R[0]$ in runs of $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}$ by assumption. Therefore \mathcal{S} will also never answer such a message incorrectly. Combined with the results from the previous paragraph, this implies $\mathcal{S} \in \text{Sim}_{R'}^{\mathcal{P}}(\mathcal{F})$. \square

Using Lemma 3.17, we directly obtain that security results carry over if one extends a restriction R by adding some *new* restricting messages (with arbitrary sets of allowed responses) that are not used by \mathcal{F} . This allows a protocol designer to simply add new messages each time he analyzes a different protocol, without having to re-prove all previous results. Should \mathcal{F} use some restricting messages one wants to add, then one could slightly redefine \mathcal{F} (e.g., by prefixing some messages) to avoid overlap. Such a change is purely syntactical and does not affect the behavior of and security results for \mathcal{F} .

We now explain how Lemma 3.17 even allows us to combine security results for two different restrictions R_1 and R_2 , where one is not necessarily an extension of the other as required by the lemma.

To combine two (or more) security results for different restrictions, one has to define a new restriction R that combines all restrictions. As a running example illustrating the exact procedure, assume two protocols $\mathcal{P}_1, \mathcal{P}_2$ that realize ideal functionalities $\mathcal{F}_1, \mathcal{F}_2$ for restrictions R_1, R_2 , i.e., $\mathcal{P}_1 \leq_{R_1} \mathcal{F}_1$ and $\mathcal{P}_2 \leq_{R_2} \mathcal{F}_2$. To combine both results, e.g., via the parallel composition theorem (cf. Theorem 3.3), we need a restriction R such that $\mathcal{P}_1 \leq_R \mathcal{F}_1$ and $\mathcal{P}_2 \leq_R \mathcal{F}_2$. Generally speaking, to construct such a restriction, one syntactically changes both the protocols and the restrictions such that protocols using different restrictions send the same restricting message only if both restrictions agree on the allowed answers. It is then simple to combine the two restrictions.

To be more precise, first define $R_{1 \cap 2}$ to be the binary set that contains all pairs $(m, m') \in R_1$ such that $m \in R_1[0] \cap R_2[0]$ and both R_1 and R_2 define the same set of allowed answers for m . In other words, $R_{1 \cap 2}$ contains those message pairs on which both restrictions agree. Now, syntactically change R_1 and R_2 such that there is no restricting message with $m \in R_1[0] \cap R_2[0]$ but $m \notin R_{1 \cap 2}[0]$. In other words, for every restricting message from the resulting restrictions, say R'_1 and R'_2 , they either agree on the allowed answers or the restricting message is defined in only one of these restrictions. This can easily be established, e.g., by prefixing every restricting message in which the restrictions do not agree with different strings.

Now observe that we can modify the protocols (and simulators) to adjust them to the new re-

restrictions by simply renaming messages in the same way. This yields new protocols $\mathcal{P}'_1, \mathcal{P}'_2, \mathcal{F}'_1, \mathcal{F}'_2$ such that $\mathcal{P}'_1 \leq_{R'_1} \mathcal{F}'_1$ and $\mathcal{P}'_2 \leq_{R'_2} \mathcal{F}'_2$. Note that these protocols are “as good as” the original protocols for a security analysis as their behavior is still exactly the same.

Now define $R := R'_1 \cup R'_2$.¹⁸ By construction, we can use Lemma 3.17 to obtain $\mathcal{P}'_1 \leq_R \mathcal{F}'_1$ and $\mathcal{P}'_2 \leq_R \mathcal{F}'_2$. This yields realizations for the same restriction, and thus it is now possible to apply all lemmas and theorems of our framework.

3.2.10. Discussion

In this section, we discuss central design choices and resulting properties of our IITM model with responsive environments.

Defining responsiveness only for a specific protocol. Our definitions of responsive environments and responsive simulators depend on a specific protocol. That is, an environment $\mathcal{E} \in \text{Env}_R(\mathcal{P})$ is required to be responsive only for a specific protocol \mathcal{P} ; similarly for simulators $\mathcal{S} \in \text{Sim}_R^{\mathcal{P}}(\mathcal{F})$. This is an intuitive definition for simulators as they are used only in the context of a single protocol \mathcal{F} . In contrast, for environments one might ask the question whether one could alternatively base the model on a set of “universally responsive environments” that are responsive for all protocols. For such universally responsive environments, we would not have to prove Lemmas 3.1 and 3.8, and the proofs of some other lemmas and theorems would become slightly simpler.

However, universally responsive environments come with the risk of overly restricting the set of environments. In particular, for a restriction R and protocols \mathcal{P}, \mathcal{R} , there might be environments \mathcal{E} such that \mathcal{E} is responsive for \mathcal{P} but not for \mathcal{R} . Such an environment \mathcal{E} would not be universally responsive and would thus not be considered for the security definition (cf. Definition 3.18) when the protocol \mathcal{P} is analyzed, even though \mathcal{E} should intuitively be a valid environment for \mathcal{P} to run in. This in turn means that one might potentially miss valid attacks during a security analysis as the environment that executes such an attack might not be responsive for some other (potentially very artificial) protocol \mathcal{R} . Furthermore, it is also generally hard to check whether a given restriction overly restricts universally responsive environments: one has to argue over the set of all protocols \mathcal{R} , including artificial ones, and check that none of them causes non-responsiveness for any “interesting” environment.

To avoid the risk of overly restricting the set of environments that is considered in a security analysis, which would in turn weaken security results to an unacceptable degree, we require responsiveness only for the single protocol \mathcal{P} that we want to analyze. This also makes it very easy to check whether a restriction R excludes only artificial attacks with no counterpart in reality: this can be determined by looking only at the single protocol \mathcal{P} at hand.

We note that from the point of view of a protocol designer that performs a security analysis, a

¹⁸Formally, one still has to ensure that R actually is a restriction, i.e., fulfills Definition 3.13. This, however, should be the case for natural definitions of R_1 and R_2 .

model based on universally responsive environments does not provide any advantages compared to our model. This is because a protocol designer can already use our model to exclude all unwanted behavior of environments for the specific protocol \mathcal{P} at hand, i.e., there is no need for universally responsive environments.

On the necessity of restrictions. An alternative way to try and solve the non-responsiveness problem would have been to introduce a “flag” that protocols can set when they send a network message. Such a flagged message would then be considered as restricting and force the environment to respond immediately, i.e., the next message to the protocol must be on the corresponding tape. The allowed responses could either be arbitrary messages, or one could fix a certain mechanism for deriving possible responses from restricting messages. Note that a key difference between this approach and our model is that the protocol designer would not be able to further customize the sets of allowed responses; responses are rather hard coded into the model itself. Intuitively, however, this approach might still seem sufficient to solve the non-responsiveness problem.

At least for the IITM model this is not a suitable solution. This is due to the very flexible and general addressing mechanism for instances of IITMs: Protocol designers can freely specify the `CheckAddress` mode of a machine M to determine, upon receiving an incoming message on some tape, which instance of M gets to process the message. To prevent the non-responsiveness problem, we have to make sure that a response to a restricting message is sent not only to the correct machine M , but rather to the specific instance of M that sent the restricting message in the first place. Thus, one must be able to restrict the environment to responses of a certain format such that, in `CheckAddress` mode, the response is accepted only by the correct instance. As this format strongly depends on how the `CheckAddress` mode is specified, which in turn is specified by the protocol designer and can differ from machine to machine, one needs a flexible solution that can be tailored towards the individual protocols at hand.

This is the main reason for introducing the highly customizable concept of restrictions. This allows protocol designers to tailor sets of allowed responses for each restricting message to the (addressing mechanisms of the) individual protocols at hand. For example, the restriction given in Section 3.2.3 is built for protocols where every instance of a machine has a unique ID id . Outgoing/incoming messages on the network are then prefixed with id to uniquely determine the sending/receiving instance of a machine. For such an addressing mechanism, the environment must be restricted to responses that start with the same id as the restricting message, thus ensuring that the correct instance receives the response.

On the generality of our model. Our IITM model with responsive environments is strictly more general than the original IITM model. This is because our model contains the original IITM model as a special case for the restriction $R = \emptyset$, i.e., the restriction that does not restrict environments in any way. This in particular implies that all security results obtained in the original IITM model carry over to our model and, by Section 3.2.9, can be combined with results

obtained for arbitrary other restrictions.

In comparison to the original IITM model, our model not only simplifies protocol definitions and security proofs but even allows for expressing properties and proving results that the original IITM model was not able to capture (cf. Sections [3.1](#) and [3.3](#)). This makes our extension also more expressive.

3.3. Applying Responsive Environments

Our new concepts of restricting messages and responsive environments and adversaries allow protocol designers to avoid the non-responsiveness problem elegantly and completely. As mentioned, urgent requests can simply be declared to be restricting messages, causing the adversary/environment to reply with a valid response before sending any other message to the protocol. This seems to be the most reasonable and natural solution to the non-responsiveness problem. We now show that our approach indeed easily solves all the problems mentioned in Section 3.1, often even retroactively for existing literature.

The frequently encountered formulations of the form (3.1) mentioned in Section 3.1.1 can now actually be used without causing confusion and flawed specifications, if the message sent to the adversary is declared to be restricting: there will now in fact be an immediate answer to this message. Similarly, ideal functionalities which are intended to be non-interactive can now actually be made non-interactive (at least if uncorrupted; but, if desired and realistic, also in the corrupted case) just like their realizations, which solves the problem discussed in Section 3.1.3 (lack of expressivity), and also makes it possible to use the, again, often encountered specifications of the form (3.2): if such ideal functionalities have to send urgent requests to the adversary, such requests can be made restricting, and hence, prompt replies are guaranteed, i.e., if the (responsive) adversary/environment contacts the protocol at all again, it first has to answer the request. Clearly, the other problems caused by urgent requests not being answered immediately discussed in Section 3.1, namely, unintended state changes and race conditions, the reentrance problem, and unnatural specifications of higher-level protocols, vanish also; again, because urgent request now *are answered immediately*.

We note that there are essentially two main approaches to define restrictions R that solve the above problems:

Tailored restrictions. One approach is to define restrictions tailored to specific protocols and functionalities. For example, for \mathcal{F}_{D-Cert} the restriction could be defined as follows:

$$\{((\text{Verify}, sid, m, \sigma), (\text{Verified}, sid, m, \phi)) : sid, m, \sigma \in \{0, 1\}^*, \phi \in \{0, 1\}\}$$

Now, whenever the adversary is asked to verify some σ , the next message sent to the ideal functionality is guaranteed to be the expected response. This directly resolves the issues discussed in Section 3.1.2. Similarly, one could, for example, also define suitable restrictions for \mathcal{F}_{NIKE} and \mathcal{F}_{SoK} .¹⁹

We note that the above approach of defining a separate restriction for each protocol allows one to fix many ideal functionalities and their realizations found in the literature without any

¹⁹Note that to show that the respective real protocols realize their ideal functionalities, according to Definition 3.18, one needs to prove that there exists a *responsive* simulator. However, it is easy to verify that the simulators constructed in [57, 72, 116] for the functionalities mentioned already are responsive, and thus these realizations can be used unalteredly also in a responsive setting.

modifications to the specifications, including all examples mentioned in this chapter. However, since the composition theorems and the transitivity property assume one restriction, different restrictions have to be combined into a single one to compose different protocols. This is always possible as shown in Section 3.2.9. Nevertheless, the following solution seems preferable.

Generic Restriction. Alternatively to employing tailored restrictions, one can use a generic restriction of the following form:

$$R_G := \{(m, m') \mid m = (\mathbf{Respond}, m''), m', m'' \in \{0, 1\}^*\}.$$

This means that messages prefixed with **Respond** are considered to be restricting, and hence protocol designers can declare a message to be restricting by simply prefixing it by **Respond**. According to the definition of R_G , the adversary/environment can respond with any message to these messages, but protocols or ideal functionalities can be defined in such a way that they repeat their requests until they receive the expected answer: for instance, in the case of \mathcal{F}_{SoK} , it can repeatedly send $m'' = (\mathbf{Setup}, \text{sid})$ to the adversary until it receives the expected algorithms. In this way, the adversary is forced to eventually provide an expected answer (if he wants the protocol to proceed).

We note that, as discussed in Section 3.2.10, one still has to tailor the above generic restriction towards the instantiation of the **CheckAddress** mode used in a protocol. For example, if machine instances are assumed to have a unique ID and messages are prefixed with this ID, then the generic restriction should be modified such that the ID from the restricting message is repeated in the response (cf. the example from Section 3.2.3). This is to ensure that the correct instance of a machine receives the response.

A generic restriction can be reused for all protocols with the same “type” of addressing mechanism (e.g., all protocols that prefix messages with an instance ID). Using such a single fixed multi-purpose restriction has the advantage that, in contrast to the former approach, there is no need to manually combine different restrictions as multiple protocols can be based on the same restriction. Also, in protocol specifications, the prefixing immediately makes clear which messages are considered to be restricting.

3.4. Responsive Environments Beyond the IITM Model

So far, we have focused on extending the IITM model to support responsive environments. However, our concepts are of interest even beyond the IITM model as they are sufficiently general to be applicable to any universal composability model. We exemplify this by outlining how the prominent UC and GNUC models can be extended to support responsive environments. While the UC, GNUC, and IITM models, as explained in the introduction, are all closely related in that they follow the same general ideas for their computational frameworks, they differ in several details which affect the concrete implementation of our concepts in these models (see, e.g., [75, 99] for a detailed discussion of these differences). The main differences and details to be considered concern runtime definitions and the mechanism for addressing (instances of) machines. Those differences makes it necessary to slightly adjust the definitions of protocol runtime, restrictions, responsive environments, and responsive adversaries/simulators to each model (cf. Sections 3.2.3 and 3.2.4). Before we describe each change in detail, let us briefly summarize the central aspects that we have to take care of, why they have to be changed, and recap how these aspects have been addressed in our extension of the IITM model:

Runtime. Some models require the runtime of systems/protocols to be polynomially bounded only for a certain class of environments. For those cases, the class of environments should be restricted to those that are responsive, just as we did for the IITM model.²⁰

Definition of restrictions. For restrictions (cf. Definition 3.13), we need that an environment \mathcal{E}' that simulates another environment \mathcal{E} is able to decide whether the simulated \mathcal{E} is about to send an incorrect response to a restricting message. This in turn depends on the runtime notions used for environments (and protocols). For the IITM model, we thus required “efficient decidability in the second component” for restrictions.

Definition of responsive environments. For responsive environments (cf. Definition 3.14), we need that an answer to a restricting message is sent back to the correct (instance of a) machine. In the IITM model, this is guaranteed by requiring responses on the correct tape; as discussed in Section 3.2.10, appropriate definitions of the restriction are then used to further ensure that the correct instance of that machine receives the message.

Definition of responsive adversaries. Depending on the restriction R considered, in some models, including UC and GNUC, the definition of responsive adversaries (cf. Definition 3.19) can be too restrictive as, e.g., the dummy adversary in these models might not satisfy the definition. The dummy adversary in these models is required to perform multiplexing: When it receives a message from an instance of the protocol and forwards this message to the environment, it has to prefix the message with the ID of that instance to tell the environment where the message came from. This alters the message, and the resulting message might not

²⁰This not only makes intuitive sense, but, depending on the model, can even be necessary on a technical level: One might have to argue based on indistinguishability that some newly constructed system meets the models runtime bound. Since indistinguishability holds true only for responsive environments, we cannot make any statement about the behavior (and thus runtime) in the context of arbitrary environments. For example, this is the case for Lemma 3.5 and Theorem 3.3.

longer be restricting, depending on the definition of the restriction R . Hence, the environment would no longer be obliged to answer directly, and thus the (dummy) adversary would not be responsive. One way to fix this is to require a certain closure property of restrictions, namely that adding IDs at the beginning of restricting messages still yields restricting messages and that these messages permit the same answers. But this is quite cumbersome. For example, by recursively applying this constraint one would have to require that R be closed under arbitrarily long prefixes of sequences of IDs. A more elegant solution that would still allow simple and natural restrictions would be to redefine what it means for a message from an adversary to the environment to be restricting. This is what we suggest for the UC and GNUC models (see below). For the IITM model, this is not an issue as the dummy adversary does not have to modify messages.

In what follows, we sketch how to adjust the runtime notions and the definitions for the UC and GNUC models.

3.4.1. UC Model

For the UC model, we do not have to change the runtime definition because the runtime of a protocol is not defined w.r.t. a class of environments, but simply bounded by a fixed polynomial (see also below).

Definition of restrictions. For the UC model we require both R and $R[0]$ to be decidable in polynomial time in the length of the input. Due to the stricter runtime notion of the UC model, this relatively straightforward definition is sufficient to satisfy the requirement mentioned above, namely, that an environment \mathcal{E}' simulating another environment \mathcal{E} can check whether a restricting message received by \mathcal{E} is answered correctly by \mathcal{E} . To see this, recall that every machine in UC is required to be parameterized with a polynomial. At every point in the run, the runtime of every instance of a machine is bounded by this polynomial, where the polynomial is in $n := n_I - n_O$, with n_I being the number of bits received so far on the I/O interface from higher-level machines and n_O being the number of bits sent on the I/O interface to lower-level machines. Environment machines have to satisfy this condition as well, where n_I is the number of bits of the external input (which contains the security parameter η). Hence, as protocols will receive only a polynomial number of input bits from the environment, they can send messages of polynomial length in the length of the external input plus η only. Therefore, given some message m that was received by an environment and a response m' to this message, the message pair (m, m') has at most polynomial length in the external input plus η , and an environment is able to decide within its runtime bound whether m is restricting and, if so, whether m' is a correct answer to m by the above definition of efficiently decidable restrictions.

Definition of responsive environments. We require that a response to a restricting message be sent back to the *instance* of the machine that sent the restricting message. This is possible

because every instance in UC is assigned a globally unique ID, which is then used to specify the sender and the recipient of a message.

Definition of responsive adversaries. As explained above, messages from the adversary to the environment and vice versa may contain a prefix (typically an ID). For reasons explained above, we say in UC that such a prefix is ignored for the sake of checking whether a message is restricting and whether the answer is correct. To be more specific, a message $m = (pre, \bar{m})$ from the adversary to the environment is restricting iff $\bar{m} \in R[0]$. Also, if m is restricting (in this sense), an answer $m' = (pre', \bar{m}')$ from the environment is allowed if $(\bar{m}, \bar{m}') \in R$ and $pre' = pre$. Using this definition, it is easy to see that the dummy adversary in UC, which adds some prefix to messages from a protocol to the environment and strips off a prefix from messages from the environment to a protocol, is responsive.

Remark 3.1. We note that one should not use the generic restriction R_G in the UC model for protocols that expect a response of a specific format (instead of just an arbitrary bit string of arbitrary length). This is due to following: Recall from Section 3.3 that, if the generic restriction R_G is used, a protocol can simply re-send the original restricting message/request if a response from the adversary is not of the expected format. This mechanism effectively stops the run for the protocol until the attacker responds with an expected answer, just as for tailored restrictions. We can use this mechanism in the IITM model due to its general runtime notion: Even if we modify a protocol to re-send requests, it still meets the runtime notion as long as the original protocol met it. The same is true for the GNUC model.

This mechanism of repeating requests, however, cannot be used by protocols in the UC model. Here, a protocol has a fixed runtime bound that depends on the bits received on the I/O interface. Hence, if the protocol receives sufficiently many wrong responses on the network, it runs out of runtime and has to stop without repeating the initial restricting message. From that point onwards the adversary/environment is no longer forced to respond immediately and can instead behave freely, causing all the issues from Section 3.1. Hence, the generic restriction cannot be used in the UC model to obtain immediate responses of a specific format; it just allows for obtaining *some* responses, which might even be empty. (This is still useful for cases where the purpose of the restricting message was just to signal some event or leak some information to the adversary.)

To enforce immediate responses of a specific format in the UC model, one can instead use tailored restrictions (cf. Section 3.3). This is one advantage of tailored restrictions over the generic one: They can be used also in models where protocols are not able to repeat requests arbitrarily often.

3.4.2. GNUC Model

The changes necessary for the GNUC model are similar to those for the UC model. However, the runtime notion has now also to be modified:

Runtime. Let us first recall the relevant parts of the runtime definition of GNUC.²¹ In this model, the runtime definition depends on the entity considered. For an environment \mathcal{E} , there has to exist a polynomial p that bounds the runtime of the environment in runs with *every* system where p gets as input the number of bits of all messages that have been received by the environment during the run, including the external input, plus the security parameter η . For a protocol \mathcal{P} , there has to exist a polynomial q such that the runtime of \mathcal{P} is bounded by q in runs with any environment and the dummy adversary where q gets as input the number of bits that are output by the environment (to both the adversary and the protocol). This definition has to be changed such that the runtime of a protocol needs to be bounded only for all environments (in the sense of GNUC) that in addition are responsive.

Definition of restrictions. Analogously to UC, we require R and $R[0]$ to be decidable in polynomial time in the length of the input. This is sufficient to satisfy the described requirement (\mathcal{E}' simulating \mathcal{E}) as the runtime of environments in GNUC depends on the number of bits received from a protocol. Hence, an environment is always able to read a potentially restricting message m entirely and then run an arbitrary ppt algorithm on top of it (note that the length of an answer m' is already bounded by the runtime bound of the simulated environment).

Definition of responsive environments. Just as for UC, we require that responses to restricting messages be sent to the same *instance* of a machine. This is possible in GNUC because, just as in the UC model, all machines have globally unique IDs to address instances.

Definition of responsive adversaries. Just as for UC, the adversary in GNUC might (have to) add IDs as prefixes or remove such prefixes. Therefore these prefixes are ignored in the definition of responsive adversaries.

²¹Note that there are several additional requirements, such as bounds on the number of bits that are sent by the environment as well as so-called invited messages. These details, however, are not relevant here.

3.5. Related Work

The concept of responsive adversaries and environments is new and has not been considered before. While there are some works that also restrict environments or adversaries, these works address different issues that are not related to the non-responsiveness problem. We discuss these works in the following.

In [9], composition for restricted classes of environments is studied, motivated by impossibility results in universal composability frameworks and to weaken the requirements on realizations of ideal functionalities. In this setting, environments are restricted in that they may send only certain sequences of messages to the I/O interfaces of protocols and functionalities. These restrictions cannot express that urgent requests are answered immediately and also do not restrict adversaries in any way. Hence, this approach cannot be used to solve the non-responsiveness problem, which anyway was not the intention of the work in [9].

In the first version of his UC model [35, 36], Canetti introduced *immediate functionalities*. According to the definition (cf. page 35 of the 2001 version), an immediate functionality uses an *immediate adversary* to guarantee that messages are delivered immediately between the functionality and its dummy. To be more precise, an immediate functionality may force an immediate adversary to deliver a message to a specific dummy party within a single activation. This construct was necessary as in the initial version of Canetti’s model, the ideal functionality could not directly pass an output to its dummy parties but had to rely on the adversary instead. In current versions of UC, the problem addressed by immediate adversaries has vanished completely because ideal functionalities can directly communicate with their dummies. Clearly, immediate adversaries do not address, let alone solve, the non-responsiveness problem, which is about immediate answers for certain request to the adversary on the network interface rather than between a functionality and its dummies.

In a recent update of the UC model from 2018 [35]²² Canetti added predicates to the UC model that allow for restricting the environment. More specifically, when an environment sends an input on the I/O interface of a protocol in the name of some higher-level protocol instance with identifier id , then this input is delivered if and only if id meets a specific predicate ξ (that might depend on the whole run so far). This mechanism is similar and closely related to [9] discussed above, but focuses on preventing the environment from claiming certain higher-level protocol identities instead of prohibiting certain message contents and message sequences. In particular, this approach is also not aimed at and does not solve the non-responsiveness problem.

²²This update occurred after our IITM model with responsive environments had already been published and was therefore not discussed in our original publication in [29].

4. The iUC Framework

In this chapter, we present our iUC framework, which was originally published in [31]. As explained in the introduction (cf. Chapter 1), the iUC framework is an instantiation of the IITM model with responsive environments that preserves its soundness and flexibility while drastically improving its usability. Therefore the iUC framework provides an excellent combination of soundness, flexibility, and usability that is so far unmatched in the universal composability literature.

This chapter is structured as follows. We start by fixing some notation and terminology in Section 4.1 before presenting the full iUC framework in Section 4.2. Section 4.3 provides a simple example of a protocol specified in the iUC framework, with more examples available in our case study (cf. Chapter 5). We discuss various core concepts and important features of the iUC framework in Section 4.4, with a direct comparison to other important universal composability models given in Section 4.5. Further details are available in Appendices B to D.

4.1. Preliminaries

In this section we fix some notation and terminology to simplify the following presentation of the iUC framework. This mainly concerns changes to notation and terminology from Chapter 3. More specifically:

- **(Bidirectional) tapes:** For simplicity of presentation, we will abstract away from input and output tapes and simply say that an IITM M has a (bidirectional) tape t with some name n . Another machine M' is connected to M if it has a tape t' also named n ; we also say that the tapes t and t' are connected. Hence, instances of M can send and receive messages to and from instances of M' . As before, we require that tapes are uniquely connected within a system, i.e., there are at most two tapes with the same name.

On a technical level, such a (bidirectional) tape t is actually a pair of input and output tapes (t, t^{-1}) with suitable names, i.e., exactly what was considered throughout Chapter 3. This pair can then be connected to another (bidirectional) tape t' , i.e., the pair of input and output tapes (t', t'^{-1}) , by naming the input and output tapes appropriately. Hence, the above simplification is justified and indeed only a change in presentation.

- **System notation:** We will use the notation $\{M_1, \dots, M_n\}$ to denote the system of IITMs $!M_1 \mid \dots \mid !M_n$, i.e., we consider machines that can spawn arbitrarily many instances during a run to be the default. Note that this does not limit the expressiveness of the IITM model:

protocol designers can still model machines that spawn at most one instance by defining the `CheckAddress` mode appropriately. The iUC framework provides a similar mechanism that can be used in the same way to model a single instance per machine.

To keep notation consistent, we will also write $\{\mathcal{R}, \mathcal{Q}\}$ to denote the combination of two systems $\mathcal{R} | \mathcal{Q}$. This will mostly be used in the context of the realization relation where we consider an environment \mathcal{E} running with a real protocol \mathcal{P} or a simulator \mathcal{S} and an ideal protocol \mathcal{F} , i.e., the combined systems $\{\mathcal{E}, \mathcal{P}\}$ and $\{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$.

- **Notation for the IITM model with responsive environments:** For the rest of this work, we consider the IITM model with responsive environments to be the default. Hence, we can simplify some of the notation from Chapter 3 which was mainly used to distinguish new notions of our extension from notions of the original IITM model. In particular, we can make the restriction R implicit (there is of course still a restriction, which, however, is fixed and thus identical for all protocols in the iUC framework; cf. Appendix D.1 for the formal definition of the restriction that we use). The changes are as follows:
 - **Model:** In what follows, when we say *IITM model*, we now refer to our extension of the IITM model with responsive environments. We will say *original IITM model* to refer to the basic IITM model.
 - **Runtime:** We now simply say *environmentally bounded* instead of *R-environmentally bounded* (cf. Definition 3.15). That is, we always consider the runtime notion that is defined with respect to responsive environments.
 - **Sets:** We write $\text{Env}(\mathcal{Q})$ and $\text{Adv}(\mathcal{Q})$ (instead of $\text{Env}_R(\mathcal{Q})$ and $\text{Adv}_R(\mathcal{Q})$) to denote the sets of responsive environments and adversaries for a system \mathcal{Q} . In slight abuse of notation we also use $\text{Adv}(\mathcal{Q})$ to denote the set of responsive simulators. This simplification is justified as the definitions of responsive simulators and responsive adversaries is essentially the same, except that simulators also have to meet some conditions concerning connectivity. Note that these conditions are only of technical nature anyway and can always be trivially met for any two protocols with identical I/O interfaces, potentially by renaming some network tapes (which does not affect the system behavior and hence security statement). We will therefore keep these conditions implicit in what follows.
 - **Realization relation:** We use the symbol \leq to denote the realization relation \leq_R that is defined for responsive environments and adversaries. Analogously, we just write $\leq_{\sigma\text{-single}}$ instead of $\leq_{R, \sigma\text{-single}}$.

4.2. Protocols in the iUC Framework

In this section, we present the iUC framework which is built on top of the IITM model with responsive environments. As explained in the introduction (cf. Chapter 1), the main shortcoming of the IITM model is a lack of usability due to missing conventions for protocol specifications. Thus, protocol designers have to manually define many repetitive modeling related aspects such as a corruption model, connections between machines, specifying the desired interpretation of machine instances (e.g., does an instance model a single party, a protocol session consisting of multiple parties, a globally available resource), the application specific addressing of individual instances, etc. The iUC framework solves this shortcoming by adding convenient and powerful conventions for protocol specifications to the IITM model. A key difficulty in crafting these conventions is preserving the flexibility of the original IITM model in terms of expressing a multitude of various protocols in natural ways, while at the same time not overburdening a protocol designer with too many details. We solve this tension by providing *a single template* for specifying arbitrary types of protocols, including real, ideal, joint state, global state protocols, which needed several sets of conventions and syntax in other frameworks, and sometimes even new theorems. Our template includes many optional parts with sensible defaults such that a protocol designer has to define only those parts relevant to his specific protocol. As the iUC framework is an instantiation of the IITM model, all composition theorems and properties of the IITM model carry over.

We start by explaining the general structure of protocols in iUC in Section 4.2.1, with corruption explained in Section 4.2.2. We then present our protocol template in Section 4.2.3. In Section 4.2.4, we explain how protocol specifications can be composed in iUC to create new, more complex protocol specification. Finally, in Section 4.2.5, we present the realization relation and the composition theorem of iUC. As already mentioned, a concrete example protocol in iUC is given in Section 4.3 with more examples available in our case study (cf. Chapter 5). We provide a precise mapping from iUC protocols to the underlying IITM model in Appendix D, which is crucial to verify that our framework indeed is an instantiation of the IITM model, and hence inherits soundness and all theorems of the IITM model. We note, however, that it is not necessary to read this technical mapping to be able to use our framework. The abstraction level provided by iUC is entirely sufficient to understand and use this framework.

4.2.1. Structure of Protocols

A protocol \mathcal{P} in our framework is specified via a system of machines $\{M_1, \dots, M_n\}$. Each machine M_i implements one or more roles of the protocol, where a role describes a piece of code that performs a specific task. For example, a (real) protocol \mathcal{P}_{sig} for digital signatures might contain a **signer** role for signing messages and a **verifier** role for verifying signatures. In a run of a protocol, there can be several instances of every machine, interacting with each other (and the environment) via I/O interfaces and interacting with the adversary (and possibly

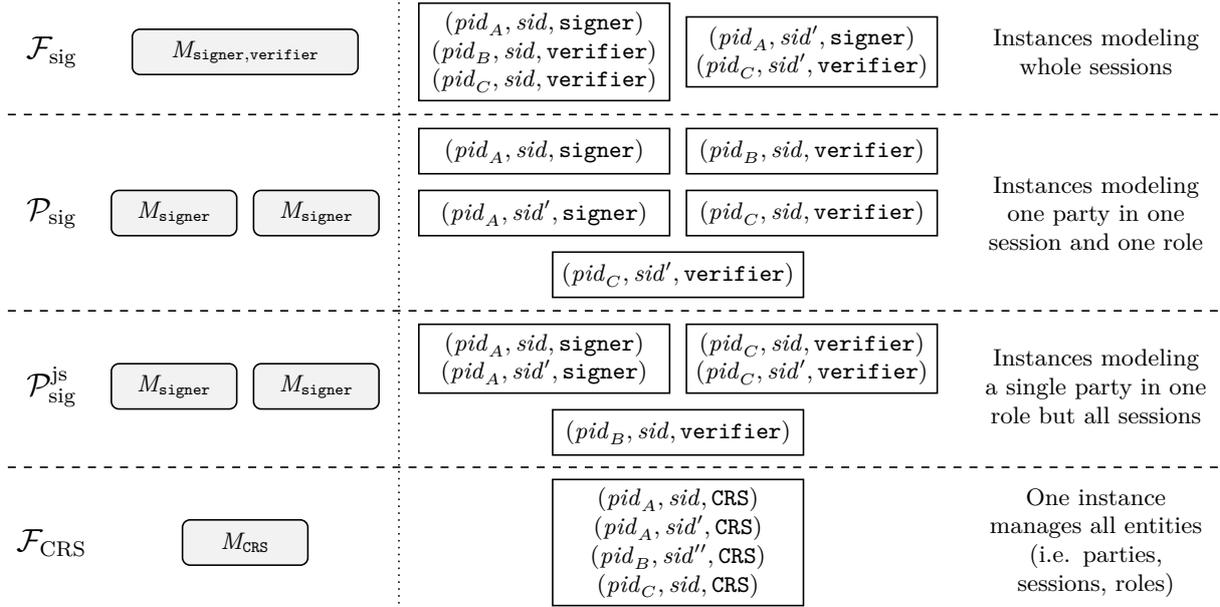


Figure 4.1.: Examples of static and dynamic structures of various protocol types. \mathcal{F}_{sig} is an ideal protocol, \mathcal{P}_{sig} a real protocol, $\mathcal{P}_{\text{sig}}^{\text{js}}$ a so-called joint state realization, and \mathcal{F}_{CRS} a global state protocol. On the left-hand side: static structures, i.e., (specifications of) machines/protocols. On the right-hand side: possible dynamic structures (i.e., several machine instances managing various entities).

the environment) via network interfaces. An instance of a machine M_i manages one or more so-called *entities*. An entity is identified by a tuple $(pid, sid, role)$ and describes a specific party with party ID (PID) pid running in a session with session ID (SID) sid and executing some code defined by the role $role$ where this role has to be (one of) the role(s) of M_i according to the specification of M_i . Entities can send messages to and receive messages from other entities and the adversary using the I/O and network interfaces of their respective machine instances. In the following, we explain each of these parts in more detail, including roles and entities; we also provide examples of the static and dynamic structure of various protocols in Figure 4.1.

Roles: As already mentioned, a role is a piece of code that performs a specific task in a protocol \mathcal{P} . Every role in \mathcal{P} is implemented by a single unique machine M_i , but one machine can implement more than one role. This is useful for sharing state between several roles: for example, consider an ideal functionality \mathcal{F}_{sig} for digital signatures consisting of a **signer** and a **verifier** role. Such an ideal protocol usually stores all messages signed by the **signer** role in some global set that the **verifier** role can then use to prevent signature forgery. To share such a set between roles, both roles must run on the same (instance of a) machine, i.e., \mathcal{F}_{sig} generally consists of a single machine $M_{\text{signer,verifier}}$ implementing both roles. In contrast, the real protocol \mathcal{P}_{sig} uses two machines M_{signer} and M_{verifier} as those roles do not and cannot directly share state in a real implementation (cf. left-hand side of Figure 4.1). Machines provide an I/O interface and a network interface for every role that they implement. The I/O

interfaces of two roles of two different machines can be connected. This means that, in a run of a system, two entities (managed by two instances of machines) with connected roles can then directly send and receive messages to/from each other; in contrast, entities of unconnected roles cannot directly send and receive messages to/from each other.²³ Jumping ahead, in a protocol specification (see Section 4.2.3) it is specified for each machine in that protocol to which other roles (subroutines) a machine connects to (see, e.g., also Figure 4.2 where the arrows denote connected roles/machines). The network interface of every role is connected to the adversary (or simulator), allowing for sending and receiving messages to and from the adversary. For addressing purposes, we assume that each role in \mathcal{P} has a unique name. Thus, role names can be used for communicating with a specific piece of code, i.e., sending and receiving a message to/from the correct machine.

Public and private roles: We, in addition, introduce the concept of public and private roles, which, as we will explain, is a very powerful tool. Every role of a protocol \mathcal{P} is either *private* or *public*. Intuitively, a private role can be called/used only internally by other roles of \mathcal{P} whereas a public role can be called/used by any protocol and the environment. Thus, private roles provide their functionality only internally within \mathcal{P} , whereas public roles provide their functionality also to other protocols and the environment. More precisely, a private role connects via its I/O interface only to (some of the) other roles in \mathcal{P} such that only those roles can send messages to and receive messages from a private role; a public role additionally provides its I/O interface for arbitrary other protocols and the environment such that they can also send messages to and receive messages from a public role. We illustrate the concept of public and private roles by an example below.

Using other protocols as subroutines: Protocols can be combined to construct new, more complex protocols. Intuitively, two protocols \mathcal{P} and \mathcal{R} can be combined if they connect to each other only via (the I/O interfaces of) their public roles. (We give a formal definition of connectable protocols in Section 4.2.4.) The new combined protocol \mathcal{Q} consists of all roles of \mathcal{P} and \mathcal{R} , where private roles remain private while public roles can be either public or private in \mathcal{Q} ; this is up to the protocol designer to decide. To keep role names unique within \mathcal{Q} , even if the same role name was used in both \mathcal{P} and \mathcal{R} , we (implicitly) assume that role names are prefixed with the name of their original protocol. We will often also explicitly write down this prefix in the protocol specification for better readability (cf. Section 4.2.3).

Examples illustrating the above concepts: Figure 4.2, which is further explained in Section 4.3, illustrates the structure of two protocols modeling a real and ideal digital signature protocol, respectively, with a globally available certificate authority (modeling a public key

²³This bidirectional connection of interfaces is an abstraction from named (pairs of input and output) tapes in the IITM model. Protocol designers need not care about those in iUC. More information about how connections are mapped to named tapes in the sense of the IITM model is available in Appendix D.

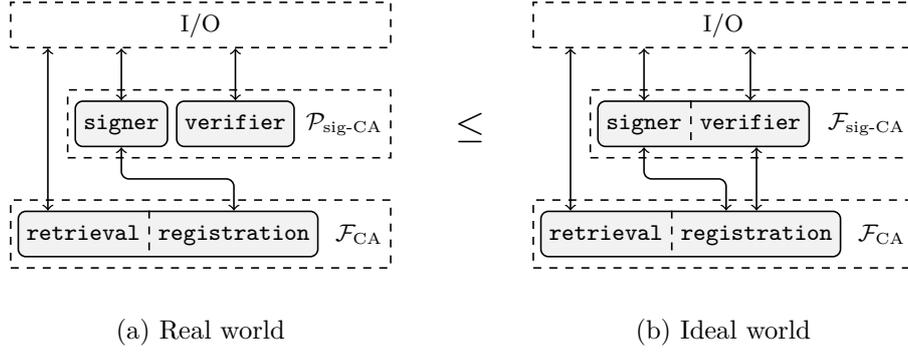


Figure 4.2.: The static structures of the ideal signature functionality with public key infrastructure $\mathcal{F}_{\text{sig-CA}}$ (right side) and its realization $\mathcal{P}_{\text{sig-CA}}$ (left side), including their subroutines (formal specifications of all machines are provided in Section 4.3). Arrows denote direct connections of I/O interfaces; network connections are omitted for simplicity. Solid boxes (labeled with one or two role names) denote individual machines, dotted boxes denote (sub-)protocols that are specified by one instance of our template each (cf. Section 4.2.3).

infrastructure). The ideal signature functionality $\mathcal{F}_{\text{sig-CA}}$, shown in Figure 4.2b, consists of a single machine with two public roles **signer** and **verifier** for signing and verifying messages. Messages that have been signed are added to a separate set of signed messages, which is then later on accessed during verification to check for and prevent signature forgery. That is, **signer** and **verifier** roles have to access the same shared state and hence need to be implemented by the same machine. The ideal signature functionality $\mathcal{F}_{\text{sig-CA}}$ uses a subroutine \mathcal{F}_{CA} , which is an ideal functionality for certificate authorities consisting of a single machine with two roles **registration** and **retrieval**. The **registration** role is private within the context of the whole protocol as only public keys generated by $\mathcal{F}_{\text{sig-CA}}$ should be registered, modeling that an honest user only ever registers his own public key that has been generated alongside his private key. If one were to make this role public and hence allow everyone to register keys, then it would not be possible to give any security guarantees in higher-level protocols that rely on $\mathcal{F}_{\text{sig-CA}}$. In contrast to the **registration** role, the **retrieval** role is public, allowing arbitrary other protocols and the environment to access all public keys stored in \mathcal{F}_{CA} . This models so-called global state, where multiple independent protocols can rely on the same shared information. The real digital signature protocol $\mathcal{P}_{\text{sig-CA}}$, shown in Figure 4.2a, is structurally very similar to $\mathcal{F}_{\text{sig-CA}}$. The main difference is that the **signer** and **verifier** roles are implemented by separate machines that in particular cannot directly share any state. This directly models reality, where the signing algorithm is generally executed on a different computer than the verification algorithm.

Now, a higher-level protocol that uses $\mathcal{F}_{\text{sig-CA}}$ (or its realization $\mathcal{P}_{\text{sig-CA}}$) may connect to all public roles, i.e., it can connect to all parts that are connected to the box labeled “I/O” in Figure 4.2b. Within this higher-level protocol, one can choose whether public roles of $\mathcal{F}_{\text{sig-CA}}$ and its subroutine should remain public, giving arbitrary other protocols access to signing, verification, and key retrieval operations, or whether some of them should be changed to be

private such that, e.g., signatures can only be created by the higher-level protocol. Altogether, with the concept of public and private roles, we can easily decide whether we want to model global state or make parts of a machine globally available while others remain local subroutines. We can even change globally available roles to be only locally available in the context of a new combined protocol.

As it is important to specify which roles of a (potentially combined) protocol are considered public and which ones are considered private, we introduce a simple notation for this. We write $(role_1, \dots, role_n \mid role_{n+1}, \dots, role_m)$ to denote a protocol \mathcal{P} with public roles $role_1, \dots, role_n$ and private roles $role_{n+1}, \dots, role_m$. If there are no private roles, we just write $(role_1, \dots, role_n)$, i.e., we omit “|”. Using this notation, the example (real) digital signature protocol from Figure 4.2a can be written as $(\text{signer}, \text{verifier}, \text{retrieval} \mid \text{registration})$.

Entities and Instances: As mentioned before, in a run of a protocol there can be several instances of every protocol machine, and every instance of a protocol machine can manage one or more, what we call, *entities*. Recall that an entity is identified by a tuple $(pid, sid, role)$, which represents party pid running in a session with SID sid and executing some code defined by the role $role$. As also mentioned, such an entity can be managed by an instance of a machine only if this machine implements $role$. We note that sid does not necessarily identify a protocol session in a classical sense. The general purpose is to identify multiple instantiations of the role $role$ executed by party pid . In particular, entities with different SIDs may very well interact with each other, if so desired, unlike in many other frameworks.

The novel concept of entities allows for easily customizing the interpretation of a machine instance by managing appropriate sets of entities. An important property of entities managed by the same instance is that they have access to the same internal state, i.e., they can share state; entities managed by different instances cannot access each others internal state directly. This property is usually the main factor for deciding which entities should be managed in the same instance. With this concept of entities, we obtain a *single* definitional framework for modeling various types of protocols and protocol components in a uniform way, as explained next and further illustrated by the examples in Figure 4.1.

One instance of an ideal protocol in the literature, such as a signature functionality \mathcal{F}_{sig} , often models a single session of a protocol. In particular, such an instance contains all entities for all parties and all roles of one session. Figure 4.1 shows two instances of the machine $M_{\text{signer, verifier}}$, managing sessions sid and sid' , respectively. In contrast, instances of real protocols in the literature, such as the realization \mathcal{P}_{sig} of \mathcal{F}_{sig} , often model a single party in a single session of a single role, i.e., every instance manages just a single unique entity, as also illustrated in Figure 4.1. If, instead, we want to model one global common reference string (CRS), for example, we have one instance of a machine M_{CRS} which manages all entities, for all sessions, parties, and roles. To give another example, the literature also considers so-called joint state realizations [51, 94] where a party reuses some state, such as a cryptographic key, in multiple sessions. An instance of such a joint state realization thus contains entities for a single

party in one role and in all sessions. Figure 4.1 shows an example joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ of \mathcal{F}_{sig} where a party uses the same signing key in all sessions. As illustrated by these examples, instances model different things depending on the entities they manage.

Exchanging messages: Entities can send and receive messages using the I/O and network interfaces belonging to their respective roles. When an entity sends a message it has to specify the receiver, which is either the adversary in the case of the network interface or some other entity (with a role that has a connected I/O interface) in the case of the I/O interface. If a message is sent to another entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$, then the message is sent to the machine M implementing $role_{rcv}$; a special user-defined **CheckID** algorithm (see Section 4.2.3) is then used to determine the instance of M that manages $(pid_{rcv}, sid_{rcv}, role_{rcv})$ and should hence receive the message. When an entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$ receives a message on the I/O interface, i.e., from another entity $(pid_{snd}, sid_{snd}, role_{snd})$, then the receiver learns pid_{snd} , sid_{snd} ²⁴ and either the actual role name $role_{snd}$ (if the sender is a known subroutine of the receiver, cf. Section 4.2.3) or an arbitrary but fixed number i (from an arbitrary but fixed range of natural numbers) denoting a specific I/O connection to some (unknown) sender role (if the sender is an unknown higher-level protocol or the environment²⁵). The latter models that a receiver/subroutine does not necessarily know the exact machine code of a caller in some arbitrary higher-level protocol, but the receiver can at least address the caller in a consistent way for sending a response. If a message is received from the network interface, then the receiving entity learns only that it was sent from the adversary.

We note that we do not restrict which entities can communicate with each other as long as their roles are connected via their I/O interfaces, i.e., entities need not share the same SID or PID to communicate via an I/O connection. This, for example, facilitates modeling entities in different sessions using the same resource, as illustrated in our case study in Chapter 5. It, for example, also allows us to model the global functionality \mathcal{F}_{CRS} from Figure 4.1 in the following natural way: \mathcal{F}_{CRS} could manage only a single (dummy) entity $(\epsilon, \epsilon, \text{CRS})$ in one machine instance, which can be accessed by all entities of higher-level protocols.

4.2.2. Modeling Corruption

We now explain on an abstract level how our framework models corruption of entities. In Section 4.2.3, we then explain in detail how particular aspects of the corruption model are specified and implemented. Our framework supports five different basic corruption models, which can be further customized by a protocol designer: *incorruptible*, *static corruption*, *dynamic corruption with/without secure erasures*, and *custom corruption*. Incorruptible protocols do not allow the adversary to corrupt any entities; this can, e.g., be used to model setup assumptions

²⁴The environment can claim arbitrary PIDs and SIDs as sender.

²⁵The environment can choose the number that it claims as a sender as long as it does not collide with a number used by another (higher-level) role in the protocol. The numbers assigned to and used by other (higher-level) roles in the protocol are arbitrary but fixed.

such as common reference strings which should not be controllable by an adversary. Static corruption allows adversaries to corrupt entities when they are first created, but not later on, whereas dynamic corruption allows for corruption at arbitrary points in time. In the case of dynamic corruption, one can additionally choose whether by default only the current internal state (known as dynamic corruption *with secure erasures*) or also a history of the entire state, including all messages and internal random coins (known as dynamic corruption *without secure erasures*) is given to the adversary upon corruption. Finally, custom corruption is a special case that allows a protocol designer to disable corruption handling of our framework and instead define his own corruption model while still taking advantage of our template and the defaults that we provide; we will ignore this custom case in the following description.

To corrupt an entity ($pid, sid, role$) in a run, the adversary can send the special message **Corrupt** on the network interface to that entity. Note that, depending on the corruption model, such a request might automatically be rejected (e.g., because the entity is part of an incorruptible protocol). In addition to this automatic check, protocol designers are also able to specify an algorithm **AllowCorruption**, which can be used to specify arbitrary other conditions that must be met for a **Corrupt** request to be accepted. For example, one could require that all subroutines must be corrupted before a corruption request is accepted (whether or not subroutines are corrupted can be determined using **CorruptionStatus?** requests, see later), modeling that an adversary must corrupt the entire protocol stack running on some computer instead of just individual programs, which is often easier to analyze (but yields a less fine grained security result). One could also prevent corruption during a protected/trusted “setup” phase of the protocol, and allow corruption only afterwards.

If a **Corrupt** request for some entity ($pid, sid, role$) passes all checks and is accepted, then the state of the entity is leaked to the adversary (which can be customized by specifying an algorithm **LeakedData**) and the entity is considered *explicitly corrupted* for the rest of the protocol run. The adversary gains full control over explicitly corrupted entities: messages arriving on the I/O interface of ($pid, sid, role$) are forwarded on the network interface to the adversary, while the adversary can tell ($pid, sid, role$) (via its network interface) to send messages to arbitrary other entities on behalf of the corrupted entity (as long as both entities have connected I/O interfaces). The protocol designer can control which messages the adversary can send in the name of a corrupted instance by specifying an algorithm **AllowAdvMessage**. This can be used, e.g., to prevent the adversary from accessing uncorrupted instances or from communicating with other (disjoint) sessions, as detailed in Section 4.2.3.

In addition to the corruption mechanism described above, entities that are activated for the first time also determine their initial corruption status by actively asking the adversary whether he wants to corrupt them. More precisely, once an entity ($pid, sid, role$) has finished its initialization (see Section 4.2.3), it asks the adversary via a *restricting message*²⁶ whether he

²⁶Recall from Section 3.2 that by sending a restricting message the adversary is forced to answer, and hence, decide upon corruption right away, before he can interact in any other way with the protocol, preventing artificial interference with the protocol run. This is a very typical use of restricting messages, which very much

wants to corrupt $(pid, sid, role)$ before performing any other computations. The answer of the adversary is processed as discussed before, i.e., the entity decides whether to accept or reject a corruption request. This gives the adversary the power to corrupt new entities right from the start, if he desires; note that in the case of static corruption, this is also the last point in time where an adversary can explicitly corrupt $(pid, sid, role)$.

For modeling purposes, we allow other entities and the environment to obtain the current corruption status of an entity $(pid, sid, role)$.²⁷ This is done by sending a special `CorruptionStatus?` request on the I/O interface of $(pid, sid, role)$. If $(pid, sid, role)$ has been explicitly corrupted by the adversary, the entity returns `true` immediately. Otherwise, the entity is free to decide whether `true` or `false` is returned, i.e., whether it considers itself corrupted nevertheless (this is specified by the protocol designer via an algorithm `DetermineCorrStatus`). For example, a higher-level protocol might consider itself corrupted if at least one of its subroutines is (explicitly or implicitly) corrupted, which models that no security guarantees can be given if certain subroutines are controlled by the adversary. A protocol might also consider itself to be corrupted if necessary security assumptions, such as an honest majority of participants, are no longer met; in such a situation the protocol cannot provide any security guarantees anymore, which is just as for a protocol that is explicitly corrupted. To figure out whether subroutines are corrupted, a higher-level protocol can send `CorruptionStatus?` requests to subroutines itself. We call an entity that was not explicitly corrupted but still returns `true` *implicitly corrupted*. We note that the responses to `CorruptionStatus?` request are guaranteed to be consistent in the sense that if an entity returns `true` once, it will always return `true`. Also, according to the defaults of our framework, `CorruptionStatus?` request are answered immediately (without intervention of the adversary) and processing these requests does not change state. These are important features which allow for a smooth handling of corruption.

4.2.3. Specifying Protocols

We now present our template for fully specifying a protocol \mathcal{Q} , including its uncorrupted behavior, its corruption model, and its connections to other protocols. As mentioned previously, the template is sufficiently general to capture many different types of protocols (real, ideal, hybrid, joint state, global, ...) and includes several optional parts with reasonable defaults. Thus, our template combines freedom with ease of specification.

The template is given in Figure 4.3. Some parts are self-explanatory; the other parts are described in more detail in the following. The first section of the template specifies properties of the whole protocol that apply to all machines.

simplifies corruption modeling.

²⁷This operation is purely for modeling purposes and does of course not exist in reality. It is crucial for obtaining a reasonable realization relation: The environment needs a way to check that the simulator in the ideal world corrupts exactly those entities that are corrupted in the real world, i.e., the simulation should be perfect also with respect to the corruption states. If we did not provide such a mechanism, the simulator could simply corrupt all entities in the ideal world which generally allows for a trivial simulation of arbitrary protocols.

Setup for the protocol $\mathcal{Q} = \{M_1, \dots, M_n\}$:

Participating roles: list of all n sets of roles participating in this protocol. Each set corresponds to one machine M_i .
Corruption model: incorruptible, static, dynamic with/without erasures, custom.
Protocol parameters*: e.g., externally provided algorithms parametrizing a machine.

Implementation of M_i for each set of roles:

Implemented role(s): the set of roles that is implemented by this machine.
Subroutines*: a list of all (other) roles that this machine uses as subroutines.
Internal state*: state variables used to store data across different invocations.
CheckID*: algorithm for deciding whether this machine is responsible for an entity $(pid, sid, role)$.
Corruption behavior*: description of **DetermineCorrStatus**, **AllowCorruption**, **LeakedData**, and/or **AllowAdvMessage** algorithms.
Initialization*: this block is executed only the first time an instance of the machine accepts a message; useful to, e.g., assign initial values that are globally used for all entities managed by this instance.
EntityInitialization*: this block is executed only the first time that some message for a (new) entity is received; useful to, e.g., assign initial values that are specific for single entities.
MessagePreprocessing*: this algorithm is executed every time a message for an uncorrupted entity is received.
Main: specification of the actual behavior of an uncorrupted entity.

Figure 4.3.: Template for specifying protocols. Blocks labeled with an asterisk (*) are optional and replaced by a default if not specified. **CheckID** is part of the **CheckAddress** mode, whereas **Corruption behavior**, \dots , **Main** are all executed within the **Compute** mode of the machine. Note that the template does not specify public and private roles as those change depending on how several protocols (each defined via a copy of this template) are connected.

Participating roles: This list of sets of roles specifies which roles are (jointly) implemented by a machine. To give an example, the list “ $\{role_1, role_2\}, role_3, \{role_4, role_5, role_6\}$ ” specifies a protocol \mathcal{Q} consisting of three machines $M_{role_1, role_2}$, M_{role_3} , and $M_{role_4, role_5, role_6}$, where $M_{role_1, role_2}$ implements $role_1$ and $role_2$, and so on.

Corruption model: This fixes one of the basic corruption models supported by iUC, as explained in Section 4.2.2: *incorruptible*, *static*, *dynamic with erasures*, and *dynamic without erasures*. Moreover, if the corruption model is set to *custom*, the protocol designer has to manually define his own corruption model and process corruption related messages, such as **CorruptionStatus?**, using the algorithms **MessagePreprocessing** and/or **Main** (see below), providing full flexibility.

Apart from the protocol setup, one has to specify each protocol machine M_i , and hence, the behavior of each set of roles listed in the protocol setup.

Subroutines: Here the protocol designer lists all roles that M_i uses as subroutines. These roles may be part of this or potentially other protocols, but may not include roles that are implemented by M_i . The I/O interface of (all roles of) the machine M_i will then be connected to the I/O interfaces of those roles, allowing M_i to access and send messages to those subroutines.²⁸

²⁸We emphasize that we do not put any restrictions on the graph that the subroutine relationships of machines of several protocols form. For example, it is entirely possible to have machines in two different protocols that

We note that (subroutine) roles are uniquely specified by their name since we assume globally unique names for each role. We also note that subroutines are specified on the level of roles, instead of the level of whole protocols, as this yields more flexibility and a more fine grained subroutine relationship, and hence, access structure.

If roles of some other protocol \mathcal{R} are used, then protocol authors should prefix the roles with the protocol name to improve readability, e.g., “ $\mathcal{R} : \text{roleInR}$ ” to denote a connection to the role `roleInR` in the protocol \mathcal{R} . This is mandatory if the same role name is used in several protocols to avoid ambiguity. If a machine is supposed to connect to all roles of some protocol \mathcal{R} , then, as a short-hand notation, one can list the name \mathcal{R} of the protocol instead.

Internal state: State variables declared here (henceforth denoted by sans-serif fonts, e.g., `a`, `b`) preserve their values across different activations of an instance of M_i .

In addition to these user-specified state variables, every machine has some additional framework-specific state variables that are set and changed automatically according to our conventions. Most of these variables are for internal bookkeeping and need not be accessed by protocol designers. Those that might be useful in certain algorithms are mentioned and explained further below (we provide a complete list of all framework specific variables in Appendix D.2).

CheckID: As mentioned before, instances of machines in our framework manage (potentially several) entities $(pid_i, sid_i, role_i)$. The algorithm **CheckID** allows an instance of a machine to decide which of those entities are accepted and thus managed by that instance, and which are not. Furthermore, it allows for imposing a certain structure on pid_i and sid_i ; for example, SIDs might only be accepted if they encode certain session parameters, e.g., $sid_i = (parameter_1, parameter_2, sid'_i)$.

More precisely, the algorithm **CheckID** $(pid, sid, role)$ is a *deterministic algorithm* that computes on the input $(pid, sid, role)$, the internal state of the machine instance, and the security parameter. It runs in *polynomial time* in the length of the current input, the internal state, and the security parameter and outputs **accept** or **reject**. Every time a machine instance is invoked with a message m for some entity $(pid, sid, role)$, it runs **CheckID** $(pid, sid, role)$ in **CheckAddress** mode to determine whether it manages $(pid, sid, role)$, i.e., whether the message m should be accepted.

We require that **CheckID** behaves consistently, i.e., it never accepts an entity that has previously been rejected, and it never rejects an entity that has previously been accepted; this ensures that there are no two instances that manage the same entity. For this purpose, we provide access to a convenient framework specific list **acceptedEntities** that contains all entities that have been accepted so far (in the order in which they were first accepted). We note that **CheckID** cannot change the (internal) state of an instance; all changes caused by running **CheckID** are dropped after outputting a decision, i.e., the state of an instance is set back to

specify each other as subroutines.

the state before running **CheckID**.²⁹ In Section 4.2.6 we provide a simple syntax for easily specifying the most common cases in the **CheckID** algorithm.

If **CheckID** is not specified, its default behavior is as follows: Given input $(pid, sid, role)$, if the machine instance in which **CheckID** is running has not accepted an entity yet, it outputs **accept**. If it has already accepted an entity $(pid', sid', role')$, then it outputs **accept** iff $pid = pid'$ and $sid = sid'$. Otherwise, it outputs **reject**. Thus, by default, a machine instance accepts, and hence, manages, not more than one entity per role for the roles the machine implements, and all of these entities belong to the same party in the same session. In other words, such a machine instance models one party in one session.

Corruption behavior: This element of the template allows for customization of corruption related behavior of machines by specifying a custom version of one or more of the optional algorithms **DetermineCorrStatus**, **AllowCorruption**, **LeakedData**, and **AllowAdvMessage**. These algorithms are used to customize our corruption model, as explained in Section 4.2.2 (cf. that section for possible use cases). As these algorithms are part of our corruption conventions, they are used only if **Corruption model** is not set to *custom*. A detailed description of the purpose of every algorithm, including its default behavior if not specified, is given next.

The **DetermineCorrStatus** $(pid, sid, role)$ algorithm is used to customize the response upon receiving **CorruptionStatus?** requests (recall that other roles/the environment can send this message on the I/O interface to obtain the current corruption status of $(pid, sid, role)$); the algorithm must output either **true** or **false**. More precisely, upon receiving a **CorruptionStatus?** request from a sender for some receiving entity $(pid, sid, role)$, an instance does the following right at the start of mode **Compute**: if $(pid, sid, role)$ has not yet received a message \neq **CorruptionStatus?** (and thus has not been initialized and in particular has not yet determined whether it is explicitly corrupted), then $(\text{CorruptionStatus}, \text{false})$ is sent back to the sender immediately. Otherwise, the instance checks whether $(pid, sid, role)$ has been explicitly corrupted by the adversary. If so, the instance sends $(\text{CorruptionStatus}, \text{true})$ back to the sender. The same also happens if, at some point in the past, the instance has already responded with **true** for $(pid, sid, role)$ at least once. Finally, in all other cases the **DetermineCorrStatus** algorithm is called to decide whether the instance responds with the message $(\text{CorruptionStatus}, \text{true})$ or $(\text{CorruptionStatus}, \text{false})$. As mentioned previously, this decision might depend on, e.g., the corruption status of subroutines. In particular, **DetermineCorrStatus** might ask for the corruption status of (entities in) subroutines by sending **CorruptionStatus?** to (entities in) these subroutines. If not specified, the **DetermineCorrStatus** algorithm always returns **false**. We note that *no other actions are performed* during or after responding to a **CorruptionStatus?** request. In particular, neither **Initialization**, **Main**, or such is performed nor is the adversary asked whether he wants to

²⁹This is because **CheckID** is part of mode **CheckAddress** which resets all state changes after it has finished its computation.

corrupt an entity.³⁰

The **AllowCorruption**($pid, sid, role$) algorithm is used to decide whether an adversary may explicitly corrupt an entity ($pid, sid, role$); it must output **true** or **false**. More precisely, when an the adversary asks to corrupt some uncorrupted entity ($pid, sid, role$) (either by sending a **Corrupt** request or when ($pid, sid, role$) determines its initial corruption status), the entity first checks whether, for the specified corruption mode of the protocol, corruption of this entity is possible at the current point in time, and rejects the request if not. Otherwise, **AllowCorruption** is called to decide whether corruption of that entity is accepted. The decision can, for example, depend on the corruption states of (entities in) subroutines, as explained in Section 4.2.2. If the algorithm outputs **true**, then the entity is henceforth considered explicitly corrupted. The entity is then also added to a framework-specific set **CorruptionSet** that keeps track of all explicitly corrupted entities managed by the current machine instance; this set can be used by a protocol designer for example to let the behavior of algorithms depend on which entities are explicitly corrupted. If the **AllowCorruption** algorithm is not specified, then the default is to always return **true**, i.e., the adversary is not further restricted in which entities he may corrupt.

The **LeakedData**($pid, sid, role$) algorithm is used to determine the data that is leaked to the adversary upon successful (explicit) corruption of an entity ($pid, sid, role$); it outputs an arbitrary bit string. Formally, this algorithm is called directly after **AllowCorruption** has allowed corruption by outputting **true**; the output of **LeakedData** is then sent as part of a confirmation of successful corruption back to the network. To make the specification of **LeakedData** easier, authors can use the framework specific variable **transcript** that contains a list of all messages that have been received and sent by the **MessagePreprocessing** and **Main** algorithms (see below) of this instance; in other words, this variable generally contains all messages except for (meta) messages related to initialization and corruption. If **LeakedData** is not specified, the following default behavior is used: If an entity is currently determining its initial corruption status after receiving some (first) message m from some sender $sender$ and gets corrupted during this, then $(m, sender)$ is output by **LeakedData**. Thus, the adversary learns all information that this entity ever obtained, modeling that the entity was corrupted before the protocol started. If corruption occurs later on, which is possible only for dynamic corruption modes, then the leakage contains either the **Internal state** (in the case of dynamic corruption with secure erasures) or the **Internal state**, the transcript of all (non-framework specific) messages **transcript**, and all random coins that have been used so far (in the case of dynamic corruption without secure erasures). We note that, generally, this default for dynamic corruption is suitable only for instances that manage exactly one entity because the full **Internal state** and potentially also the **transcript** and random coins are leaked.

The **AllowAdvMessage**($pid, sid, role, pid_{receiver}, sid_{receiver}, role_{receiver}, m$) algorithm is used to decide whether an adversary may send a message m via an explicitly corrupted entity ($pid, sid, role$) to some other receiving entity ($pid_{receiver}, sid_{receiver}, role_{receiver}$) (where $role_{receiver}$

³⁰This ensures that **CorruptionStatus?** requests, by default, do not change the internal state of machines, which would complicate simulations needlessly.

is a role that is connected to the current machine, i.e., it is a known subroutine or a number denoting a connection to an unknown higher-level protocol). The algorithm must output either **true** or **false**, depending on whether the message is allowed. If a message m is not allowed, then the entity stops the current activation without forwarding m by returning an error message to the adversary. If **AllowAdvMessage** is not specified, it defaults to outputting **true** iff $pid = pid_{\text{receiver}}$. In other words, by default an adversary can interact only with subroutines/higher-level protocols in the name of the same party pid . This default has been chosen as in most cases we expect protocols to be designed such that parties call other protocols only in their own name; in such a setting, an adversary should not be able to directly use, e.g., subroutines belonging to other (uncorrupted) parties. Note that this default does not restrict the adversary from sending messages to entities with a different SID. If this property is desired, for example, to model disjoint protocol sessions that do not interact with each other, then this algorithm can be customized appropriately.

Initialization, EntityInitialization, MessagePreprocessing, Main: These algorithms specify the actual behavior of a machine in mode **Compute** for uncorrupted entities.

The **Initialization** algorithm is run exactly once per machine instance (*not per entity* in that instance) and is mainly supposed to be used for initializing the internal state of that instance. For example, one can generate global parameters or cryptographic key material in this algorithm.

The **EntityInitialization**($pid, sid, role$) algorithm is similar to **Initialization** but is run once for each entity ($pid, sid, role$) instead of once for each machine instance. More precisely, it runs directly after a potential execution of **Initialization** if **EntityInitialization** has not been run for the current entity ($pid, sid, role$) yet. This is particularly useful if a machine instance manages several entities, where not all of them might be known from the beginning.

After the algorithms **Initialization** and, for the current entity, **EntityInitialization** have finished, the current entity determines its initial corruption status (if not done yet) and processes a **Corrupt** request from the network/adversary, if any. Note that this allows for using the initialization algorithms to setup some internal state that can be used by the entity to determine its corruption status.

Finally, after all of the previous steps, if the current entity has not been explicitly corrupted,³¹ the algorithms **MessagePreprocessing** and **Main** are run. The **MessagePreprocessing** algorithm is executed first. If it does not end the current activation, **Main** is executed directly afterwards. While we do not fix how authors have to use these algorithms, one would typically use **MessagePreprocessing** to prepare the input m for the **Main** algorithm, e.g., by dropping malformed messages or extracting some key information from m . The algorithm **Main** should contain the core logic of the protocol.

If any of the optional algorithms are not specified, then they are simply skipped during

³¹As mentioned in Section 4.2.2, if an entity is explicitly corrupted, it instead acts as a forwarder for messages to and from the adversary.

computation. We provide a convenient syntax for specifying these algorithms in Section 4.2.6; a small illustrative example of a protocol specified using this template is given in Section 4.3 with more examples available in our case study in Chapter 5.

This concludes the description of our template. As already mentioned, in Appendix D we give a formal mapping of this template to protocols in the sense of the IITM model, which provides a precise semantic for the template and also allows us to carry over all definitions, such as realization relations, and theorems, such as composition theorems, of the IITM model to iUC (see Section 4.2.5).

4.2.4. Composing Protocol Specifications

Protocols in our framework can be composed to obtain more complex protocols. More precisely, two protocols \mathcal{Q} and \mathcal{Q}' that are specified using our template are called *connectable* if they connect via their public roles only. That is, if a machine in \mathcal{Q} specifies a subroutine role of \mathcal{Q}' , then this subroutine role has to be public in \mathcal{Q}' , and vice versa.

Two connectable protocols can be composed to obtain a new protocol \mathcal{R} containing all roles of \mathcal{Q} and \mathcal{Q}' such that the public roles of \mathcal{R} are a subset of the public roles of \mathcal{Q} and \mathcal{Q}' . Which potentially public roles of \mathcal{R} are actually declared to be public in \mathcal{R} is up to the protocol designer and depends on the type of protocol that is to be modeled (see Section 4.2.1). In any case, the notation from Section 4.2.1 of the form $(role_1^{pub} \dots role_i^{pub} \mid role_1^{priv} \dots role_j^{priv})$ should be used for this purpose.

For pairwise connectable protocols $\mathcal{Q}_1, \dots, \mathcal{Q}_n$ we define $\text{Comb}(\mathcal{Q}_1, \dots, \mathcal{Q}_n)$ to be the (finite) set of all protocols \mathcal{R} that can be obtained by connecting $\mathcal{Q}_1, \dots, \mathcal{Q}_n$. Note that all protocols \mathcal{R} in this set differ only by their sets of public roles. We define two shorthand notations for easily specifying the most common types of combined protocols: by $(\mathcal{Q}_1, \dots, \mathcal{Q}_i \mid \mathcal{Q}_{i+1}, \dots, \mathcal{Q}_n)$ we denote the protocol $\mathcal{R} \in \text{Comb}(\mathcal{Q}_1, \dots, \mathcal{Q}_n)$, where the public roles of $\mathcal{Q}_1, \dots, \mathcal{Q}_i$ remain public in \mathcal{R} and all other roles are private. This notation can be mixed with the notation from Section 4.2.1 in the natural way by replacing a protocol \mathcal{Q}_j with its roles, some of which might be public while others might be private in \mathcal{R} . Furthermore, by $\mathcal{Q}_1 \parallel \mathcal{Q}_2$ we denote the protocol $\mathcal{R} \in \text{Comb}(\mathcal{Q}_1, \mathcal{Q}_2)$ where exactly those public roles of \mathcal{Q}_1 and \mathcal{Q}_2 remain public that are not used as a subroutine by any machine in \mathcal{Q}_1 or \mathcal{Q}_2 .

We call a protocol \mathcal{Q} *complete* if every subroutine *role* used by a machine in \mathcal{Q} is also part of \mathcal{Q} . In other words, \mathcal{Q} fully specifies the behavior of all subroutines. Since a security analysis generally makes sense only for a fully specified protocol, we will (implicitly) consider this to be the default in the following.³²

³²On a technical level, the iUC framework including its realization relation and composition theorem are also well defined for incomplete protocols. This is because the iUC framework is an instantiation of the IITM model, which does not distinguish between subroutines and higher-level protocols and in particular allows both types of connections (i.e., I/O tapes) to remain unconnected and hence accessible to the environment/other protocols. Hence, if one considers an incomplete iUC protocol, then intuitively the environment takes over the role of any undefined subroutines. For simplicity of presentation and because this is really the standard case, we assume complete protocols in what follows.

4.2.5. Realization Relation and Composition Theorems

In the following, we define the universal composability experiment and state the main composition theorem of iUC. Since iUC is an instantiation of the IITM model, as shown by our mapping provided in Appendix D, both the experiment and theorem are directly carried over from the IITM model with responsive environments and merely restated using the abstraction level of iUC. Hence, we in particular do not need to re-prove the composition theorem.

Definition 4.1 (Realization relation in iUC). *Let \mathcal{P} and \mathcal{F} be two environmentally bounded complete protocols with identical sets of public roles. The protocol \mathcal{P} realizes \mathcal{F} (denoted by $\mathcal{P} \leq \mathcal{F}$) iff there exists a simulator (system) $\mathcal{S} \in \text{Adv}(\mathcal{F})$ such that for all $\mathcal{E} \in \text{Env}(\mathcal{P})$ it holds true that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$.³³*

Note that \mathcal{E} in $\{\mathcal{E}, \mathcal{P}\}$ connects to the I/O interfaces of public roles as well as the network interfaces of all roles of \mathcal{P} . In contrast, \mathcal{E} in the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ connects to the I/O interfaces of public roles of \mathcal{F} and the network interface of \mathcal{S} . The simulator \mathcal{S} connects to \mathcal{E} (simulating the network interface of \mathcal{P}) and the network interface of \mathcal{F} . This definition is based on the security notion of strong simulatability where the environment \mathcal{E} subsumes the attacker \mathcal{A} in the real world. As shown in Section 3.2.6, whether or not the adversary \mathcal{A} is considered does not change the realization relation in the IITM model and hence in iUC.

Now, the main composition theorem of iUC, which is a corollary of the composition of the IITM model (cf. Theorem 3.3), is as follows:

Corollary 4.1 (Concurrent composition in iUC). *Let \mathcal{P} and \mathcal{F} be two protocols such that $\mathcal{P} \leq \mathcal{F}$. Let \mathcal{Q} be another protocol such that \mathcal{Q} and \mathcal{F} are connectable. Let $\mathcal{R} \in \text{Comb}(\mathcal{Q}, \mathcal{P})$ and let $\mathcal{I} \in \text{Comb}(\mathcal{Q}, \mathcal{F})$ such that \mathcal{R} and \mathcal{I} have the same sets of public roles. If \mathcal{R} is environmentally bounded and complete, then $\mathcal{R} \leq \mathcal{I}$.³⁴*

Just as in the IITM model, we emphasize that this corollary also covers the special cases of protocols with joint state and global state. The iUC framework also provides a second composition theorem for unbounded self-composition of multiple sessions of a protocol, which, again, is a corollary of the corresponding general composition theorem in the IITM model (cf. Theorem 3.4). We provide this second theorem, including an example illustrating how this theorem can be used to perform a single session security analysis in iUC, in Appendix B. We discuss various types of composition supported by iUC, including composition for protocols with joint and/or global state, in more detail in Section 4.4.

³³Intuitively, the role names are used to determine which parts of \mathcal{F} are realized by which parts of \mathcal{P} , hence they must have the same sets of public roles. Roles of the same name offer the same (parameterized) number of I/O connections to the environment in both the real and ideal world.

³⁴Technically speaking, public roles of \mathcal{P} and \mathcal{F} have a different implicit prefix, i.e., a public role “ $\mathcal{P} : \text{role}$ ” in \mathcal{R} corresponds to/is replaced by the public role “ $\mathcal{F} : \text{role}$ ” in \mathcal{I} . Hence, technically the higher-level protocol \mathcal{Q} has to use a different (implicit) prefix depending on which subroutine is to be used. By slight abuse of notation, we write \mathcal{Q} in both systems \mathcal{R} and \mathcal{I} as the meaning is clear from the context.

Remark 4.1. Recall from Section 4.2.1 that the environment can use arbitrary and also arbitrarily many I/O connections of public roles to (unknown) higher-level protocols. Hence, the proof of the realization relation from Definition 4.1 has to hold true independently of (i) how many connections a public role provides to the environment (this is an arbitrary but fixed parameter in the proof) and (ii) which exact connections of a public role are used by the environment and which are used internally by (unknown) higher-level roles of \mathcal{P} , if any (this is also an arbitrary but fixed parameter). The first requirement is just a technical one as protocols are still free to, e.g., drop and disregard messages that do not arrive via a certain fixed set of connections, which effectively models that only a fixed set of connections is available. While (ii) is an actual requirement, we emphasize that it is trivially met by public roles where the only higher-level (unknown) protocol is the environment, such as all highest-level roles in protocols, and by public roles that offer the same kind of service/functionality to all (unknown) higher-level protocols, such as typical definitions of global functionalities/subroutines. Hence, these requirements are generally met for typical protocol definitions.

We note that one can actually drop both requirements as they are not needed by the realization relation and composition theorem of the underlying IITM model (cf. Definition 3.18 and Theorem 3.3). The IITM model only requires identical numbers of I/O connections for the environment in both real and ideal world, but does not make any provisions on how many there are and which exact connections are used by the environment and which are used internally. Hence, one can actually analyze and compose iUC protocols assuming that, e.g., the environment has access only to some fixed number of connections, and that certain I/O connections to (unknown) higher-level protocols (say, e.g., connections numbered 0 to 5) are always connected to specific roles within the same protocol and, in particular, are therefore not accessible to the environment. However, the presentation of the computational model (cf. Section 4.2.1), of the realization relation (Definition 4.1), and of the composition theorem (Corollary 4.1) in iUC become more involved without requirements (i) and (ii): one has to specify/consider which connections are used by which roles/the environment and how many are available in the first place. For example, if a protocol has been proven to be secure assuming that it uses the connection with number 0 of some public and hence globally available subroutine role, and another protocol has been proven to be secure using the same assumption, then it is not possible to compose both protocols with the same public subroutine role as there is only a single connection with number 0 available; security might no longer hold true while using a different connection. Hence, as requirements (i) and (ii) are typically met anyway, we have opted to include them here to drastically simplify the presentation of our framework.

4.2.6. Notation for Specifying Protocols

In the following we present a compact yet formally complete and unambiguous syntax for writing down the different blocks of the protocol specification template presented in Figure 4.3.

General notation

We recommend to use **typewriter font** for strings, **sans serif font** for global variables (i.e., variables that are persistent across multiple activations of the same instance of a machine) and functions, *italic font* for local (i.e., ephemeral) variables, and **bold font** for keywords (e.g., for sending or receiving).

Special variables

For notational convenience, each instance maintains two framework-specific global variables, namely $\text{entity}_{\text{cur}} = (\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}})$ and $\text{entity}_{\text{call}} = (\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}})$, both consisting of a PID, an SID, and a role. The former triple, $\text{entity}_{\text{cur}}$, specifies the entity that was accepted by **CheckID** and thus the entity that is currently being processed as the receiver of an incoming message. If the current activation is due to a message received on the I/O interface, then $\text{entity}_{\text{call}}$ specifies the entity that called the current entity by sending that message.

CheckID

To specify that an instance of a machine should manage all entities for a single party, one can use the informal statement

“accept all entities with the same PID”.

This is short for the following formal algorithm: “Let $(pid, sid, role)$ be the input of **CheckID**. If no entity has been accepted yet, then output **accept**. Otherwise, let $(pid_1, sid_1, role_1)$ be the first entity that was accepted at some point in the past (i.e., the first entity in the list `acceptedEntities`). Output **accept** iff $pid = pid_1$; output **reject** otherwise.”

The above specification of **CheckID** of a machine M says that each instance of M is responsible for all entities belonging to a single party pid , where different instances are responsible for different parties. In other words, a machine instance models a party. More specifically, if an instance is fresh, then it will just accept the first entity it sees. This fixes the party pid that is managed by this machine instance. For all following activations, an entity is accepted iff it belongs to pid . Note that, since entities of party pid are never rejected, there will never be a second instance of M that also accepts pid .

The informal statements

“accept all entities with the same SID”,
“accept all entities with the same SID and role”, etc.

are interpreted analogously to “accept all entities with the same PID”. We also allow for using the statement

“accept a single entity”,

which is formally defined as follows: “Let $(pid, sid, role)$ be the input of **CheckID**. If no entity has been accepted yet, then output **accept**. Otherwise, let $(pid_1, sid_1, role_1)$ be the first entity that was accepted at some point in the past (i.e., the first entity in the list **acceptedEntities**). Output **accept** iff $(pid, sid, role) = (pid_1, sid_1, role_1)$; output **reject** otherwise.”

Receiving inputs

The algorithms **Initialization**, **EntityInitialization**, **MessagePreprocessing**, and **Main** generally have to process incoming messages for various entities. We structure these algorithms as a sequence of blocks, where each block is of the generic form:

recv \langle message pattern \rangle **from** \langle sender \rangle **to** \langle receiver \rangle **s.t.** \langle condition \rangle : \langle code \rangle

Upon receiving an input, the instance sequentially checks whether the input matches one of these specifications and executes the first matching block. In particular, the ordering of blocks influences the behaviour of an instance if a message fits multiple blocks as only the first one is executed.

- The \langle message pattern \rangle describes the format of the message accepted by this code block. It is built from local variables, global variables, fixed strings, and special characters such as “(”, “)”, “,”, “_”, and “ \perp ”. To compare a message m to a pattern, first the values of all global and, if already defined, local variables are inserted into the pattern. The resulting pattern p is then compared to m , where undefined local variables match to arbitrary bit strings. If a block is entered after a successful match, then undefined local variables are initialized with the bit strings they matched with. We use the special symbol “_” in patterns to match with arbitrary bit strings, just as for undefined local variables, but without storing the results for later use.
- The \langle sender \rangle is either the constant bit string **NET** if a message is to be received on the network interface, the constant bit string **SUB** if a message is to be received on the I/O interface from some arbitrary (known) subroutine, the constant bit string **I/O** if a message is to be received on the I/O interface from some arbitrary (unknown) higher-level protocol/the environment, or of the form $(pid_{snd}, sid_{snd}, role_{snd})$ if a message is to be received from a specific sender entity on the I/O interface. In the latter case, pid and sid can be constructed just as a message pattern, whereas $role$ is a fixed bit string of a known subroutine, a fixed number indicating a connection to some (unknown) higher-level protocol, or a variable denoting an I/O connection to a subroutine or higher-level protocol (cf. paragraph “exchanging messages” in Section 4.2.1). If a concrete bit string is given, then, for better readability, authors are encouraged to prefix $role$ with the protocol it belongs to, e.g., “ \mathcal{F}_{sig} : **signer**”.

Again, “_” can be used as a wildcard symbol.

- The \langle receiver \rangle is an entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$ that denotes the intended receiving entity of a message and is built analogous to $(pid_{snd}, sid_{snd}, role_{snd})$ from above.

- In $\langle \text{condition} \rangle$ one can define arbitrary further conditions, e.g., depending on the current state of the instance, that need to be satisfied in order to enter the subsequent code block.
- Finally, $\langle \text{code} \rangle$ specifies arbitrary code that will be executed.

To omit unnecessary syntax in the above generic pattern of blocks, the $\langle \text{condition} \rangle$ and $\langle \text{receiver} \rangle$ parts can be omitted if no additional conditions need to be fulfilled to accept a message or if the message shall be accepted by this block for any receiver entity, respectively. If there is only one block that is *always* entered for all incoming messages, then the header can be omitted altogether and it suffices to provide $\langle \text{code} \rangle$ only.

Sending outputs

The activation of an instance ends when it sends outputs on one of its interfaces, or aborts with a special **abort** command in which case the environment gets activated by definition of the IITM model.

Send-commands are part of the $\langle \text{code} \rangle$ block introduced above and follow the general format:

send $\langle \text{message pattern} \rangle$ **from** $\langle \text{sender} \rangle$ **to** $\langle \text{receiver} \rangle$.

where all parts are as for receiving messages, but with swapped semantics for sending and receiving entities (e.g., NET can now only occur as $\langle \text{receiver} \rangle$). Note that it is not possible to use I/O or SUB as $\langle \text{receiver} \rangle$ for sending messages as they do not uniquely specify the receiver, i.e., the connection that is to be used for sending the message).

Analogous to before, $\langle \text{sender} \rangle$ can be omitted in the regular case where the message is sent in the name of the currently active entity, i.e., $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}})$. If in addition $\langle \text{receiver} \rangle$ is the same as the original callee, i.e., $(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}})$ respectively the network, then one can use the following shorthand notation:

reply $\langle \text{message pattern} \rangle$.

Waiting for Immediate Replies: Often, one would like to obtain some information from, e.g., a subroutine and then continue the computation where it left of. To accommodate this need, we complement the above generic commands with two additional constructions:

- The following command allows an instance to send output to a receiver and wait for a response of a specific format:

send $\langle \text{message pattern out} \rangle$ **from** $\langle \text{sender} \rangle$ **to** $\langle \text{receiver} \rangle$;
wait for $\langle \text{message pattern in} \rangle$ **s.t.** $\langle \text{condition} \rangle$.

Upon receiving the correct response from $\langle \text{receiver} \rangle$, the computation continues where it stopped, also preserving local variables. All other incoming messages will be dropped by this instance, ending the activation immediately (except for some framework specific meta messages related to corruption and initialization which are still processed to allow for, e.g., corruption of entities. See Appendix D.2.6 for more details). In other words, the instance is “stuck” until it receives the expected response. As this can easily disrupt the protocol

execution if the receiver does not answer immediately, this command should be used only sparsely and with special care (see also the remarks in Appendix D.2.6).

As before, $\langle \text{sender} \rangle$ can be omitted. In that case, the sender will default to the entity $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}})$.

- As already mentioned, iUC also supports responsive environments and adversaries where one can send restricting messages on the network interface that force the adversary to immediately return an answer to the sender (i.e., before any other interaction with the protocol can occur). One can do so via the following command:

```
send responsively  $\langle \text{message pattern out} \rangle$  from  $\langle \text{sender} \rangle$ ;  
wait for  $\langle \text{message pattern in} \rangle$  s.t.  $\langle \text{condition} \rangle$ .
```

where the $\langle \text{sender} \rangle$ can again be omitted. This command sends such a restricting message and receives the response; if the response does not match the expected criteria, then the initial message is repeated until the response is accepted. This is a useful construction in many cases where meta messages need to be exchanged with the adversary, e.g., during initialization steps of an instance, cf. Chapter 3 and Appendix D.2.6 for details.

Macros

We provide the following macros that can be used in algorithms:

- To obtain the corruption status of a subroutine entity $\text{entity}_{\text{sub}} = (\text{pid}_{\text{sub}}, \text{sid}_{\text{sub}}, \text{role}_{\text{sub}})$ one can use the following macro:

```
corr $(\text{pid}_{\text{sub}}, \text{sid}_{\text{sub}}, \text{role}_{\text{sub}})$ 
```

Formally, this macro sends the special message **CorruptionStatus?** to $\text{entity}_{\text{sub}}$ and waits for the response. The entity $\text{entity}_{\text{sub}}$ will, as explained in Section 4.2.2, respond with the corruption status of the entity (note that, when using default algorithms, the response to this request is immediate. In particular, control is returned to the caller of **corr** even if $\text{entity}_{\text{sub}}$ is corrupted). The macro then outputs this response.

- One can initialize a subroutine entity $(\text{pid}_{\text{sub}}, \text{sid}_{\text{sub}}, \text{role}_{\text{sub}})$ via the following macro:

```
init $(\text{pid}_{\text{sub}}, \text{sid}_{\text{sub}}, \text{role}_{\text{sub}})$ 
```

More specifically, this sends a special message **InitEntity** to $\text{entity}_{\text{sub}}$ and waits for a response. The entity $\text{entity}_{\text{sub}}$ will then run **Initialization**, **EntityInitialization**, and determine its initial corruption status (steps that have already been executed before are skipped). Importantly, $\text{entity}_{\text{sub}}$ will always return control to the caller of **init**, even if it gets corrupted, such that the computation can continue as expected.

We provide a formal definition of these macros in Appendix D.2.6.

Running externally provided algorithms

It is sometimes necessary to run an arbitrary algorithm \mathbf{alg} provided by the adversary or environment. For example, this is a common mechanism used in ideal functionalities such as $\mathcal{F}_{\text{sig-CA}}$ (see Figures 4.4 and 4.5 in Section 4.3) to provide security guarantees independently of a specific algorithm. However, as \mathbf{alg} is an arbitrary algorithm which might not terminate within polynomial time, a protocol \mathcal{P} running \mathbf{alg} would no longer be environmentally bounded.

To solve this issue, we introduce the following syntax: given a polynomial $p \in \mathbb{Z}[x]$, we write $\mathbf{alg}^{(p)}(x)$ to say that we simulate the algorithm \mathbf{alg} with input x , but abort the simulation after at most $p(\eta + |x|)$ steps. The overall output of the simulation is defined to be \perp if the simulation is aborted, and otherwise to be the output of \mathbf{alg} .

Note that since the polynomial p is fixed, a protocol \mathcal{P} can run $\mathbf{alg}^{(p)}(x)$ while still being environmentally bounded. Also note that for every PPT algorithm \mathbf{alg} , there is a runtime bound p such that $\mathbf{alg}^{(p)}(x)$ never aborts and thus behaves just as $\mathbf{alg}(x)$.

4.3. A Small Illustrative Example

In this section, we provide a small example that illustrates protocol specifications and composition in the iUC framework. More examples are available in our case study in Chapter 5. Our example is an ideal functionality $\mathcal{F}_{\text{sig-CA}}$ for digital signatures with globally available public key infrastructure and its realization $\mathcal{P}_{\text{sig-CA}}$. The general structure of both protocols was already shown and explained in Section 4.2.1 (cf. Figure 4.2 in that section). We now explain the behavior of both protocols in detail, with formal specifications using our template from Section 4.2.3 given in Figures 4.4 to 4.7.

The ideal functionality $\mathcal{F}_{\text{sig-CA}}$ consists of two roles, **signer** and **verifier**, that allow for the corresponding operations. Both roles are implemented by the same machine and instances of that machine manage entities that share the same SID. The SID sid of an entity is structured as a tuple (pid_{owner}, sid') , modeling a specific key pair of the party pid_{owner} . During the initialization of a machine instance, the instance asks the adversary to provide public and private keys as well as signing and verification algorithms. This is to make the security statement independent of a specific key distribution and specific algorithms. Afterwards, an instance in session (pid_{owner}, sid') allows the party pid_{owner} to register his public key in an ideal certificate authority subroutine \mathcal{F}_{CA} and then sign messages using his secret key. Signed messages are additionally added to a global set `msglist` in the instance of $\mathcal{F}_{\text{sig-CA}}$, which is used to prevent signature forgery. In contrast to the **signer** role, which limits access to a specific party, the **verifier** role in session (pid_{owner}, sid') can be accessed by arbitrary parties to verify a signature on a message m that was supposedly generated by the private key represented by (pid_{owner}, sid') . If verification is performed with the correct public key, the signing entity $(pid_{owner}, (pid_{owner}, sid'), \text{signer})$ is not corrupted, but the message m is not on in the set of signed messages `msglist`, then **verifier** outputs `false` as a message forgery has been detected. In all other cases, the output of the verification algorithm provided by the adversary is returned. This definition of $\mathcal{F}_{\text{sig-CA}}$ is closely related to standard definitions of ideal signature functionalities found in the literature (such as [94]), except that public keys are additionally registered with \mathcal{F}_{CA} .

The ideal certificate authority functionality \mathcal{F}_{CA} consists of two roles **registration** and **retrieval** that allow higher-level protocols to register and retrieve arbitrary bit strings, i.e., keys. Both roles are implemented by the same machine, and one instance of that machine manages all entities (i.e., the same keys are provided for all sessions. Of course, this could also be modeled differently to provide multiple independent instances per session ID, each with their own sets of keys). In the context of $\mathcal{F}_{\text{sig-CA}}$ (and its realization $\mathcal{P}_{\text{sig-CA}}$), the registration role is private and is only used by the **signer** role to register public keys. More specifically, if an entity $(pid, sid, \text{signer})$ from the **signer** role wants to register a key, and no key has been registered for the identifier (pid, sid) yet, then this key is stored internally in \mathcal{F}_{CA} under the identifier (pid, sid) . In contrast to the **registration** role, the **retrieval** role is public and allows arbitrary other protocols and the environment to retrieve the public key from the entity $(pid, sid, \text{signer})$ by requesting the key with identifier (pid, sid) .

Description of the protocol $\mathcal{F}_{\text{sig-CA}} = (\text{signer}, \text{verifier})$:

Participating roles: $\{\text{signer}, \text{verifier}\}$	
Corruption model: dynamic with secure erasures	
Protocol parameters:	
<ul style="list-style-type: none"> - $p \in \mathbb{Z}[x]$. 	$\left\{ \begin{array}{l} \text{Polynomial that bounds the runtime of the signing and verifications} \\ \text{algorithms provided by the adversary.} \end{array} \right.$

Description of $M_{\text{signer}, \text{verifier}}$:

Implemented role(s): $\{\text{signer}, \text{verifier}\}$	
Subroutines: $\mathcal{F}_{\text{CA}} : \text{registration}$	
Internal state:	
<ul style="list-style-type: none"> - $(\text{sig}, \text{ver}, \text{pk}, \text{sk}) \in (\{0, 1\}^* \cup \{\perp\})^4 = (\perp, \perp, \perp, \perp)$. - $\text{pidowner} \in \{0, 1\}^* \cup \{\perp\} = \perp$. - $\text{msglist} \subseteq \{0, 1\}^* = \emptyset$. - $\text{KeysGenerated} \in \{\text{ready}, \perp\} = \perp$. 	$\left\{ \begin{array}{l} \text{Algorithms and key pair.} \\ \text{Party ID of the key owner.} \\ \text{Set of recorded messages.} \\ \text{Has signer initialized his key?} \end{array} \right.$
CheckID ($\text{pid}, \text{sid}, \text{role}$):	
Check that $\text{sid} = (\text{pid}', \text{sid}')$.	
If this check fails, output reject .	
Otherwise, accept all entities with the same SID.	
	$\left\{ \begin{array}{l} \text{An instance manages all parties and} \\ \text{roles in a single session. See Section} \\ \text{4.2.6 for the formal definition of} \\ \text{this abbreviated statement.} \end{array} \right.$
Corruption behavior:	
<ul style="list-style-type: none"> - LeakedData($\text{pid}, \text{sid}, \text{role}$): If called while ($\text{pid}, \text{sid}, \text{role}$) determines its initial corruption status, use the default behavior of LeakedData. That is, output the initially received message and the sender of that message. Otherwise, if $\text{role} = \text{signer}$ and $\text{pid} = \text{pidowner}$, return KeysGenerated. In all other cases return \perp. - AllowAdvMessage($\text{pid}, \text{sid}, \text{role}, \text{pid}_{\text{receiver}}, \text{sid}_{\text{receiver}}, \text{role}_{\text{receiver}}, m$): Check that $\text{pid} = \text{pid}_{\text{receiver}}$. If $\text{role}_{\text{receiver}} = \mathcal{F}_{\text{CA}} : \text{registration}$, also check that $\text{role} = \text{signer}$ and $\text{sid} = (\text{pid}, \text{sid}')$ (for some sid'). If all checks succeed, output true, otherwise output false. 	$\left\{ \begin{array}{l} \text{Only the owner of a key may register} \\ \text{it, modeling that } \mathcal{F}_{\text{CA}} \text{ authenticates} \\ \text{the owner upon registration.} \end{array} \right.$
Initialization:	
<ul style="list-style-type: none"> send responsively InitMe to NET;^a wait for (Init, ($\text{sig}, \text{ver}, \text{pk}, \text{sk}$)). $(\text{sig}, \text{ver}, \text{pk}, \text{sk}) \leftarrow (\text{sig}, \text{ver}, \text{pk}, \text{sk})$. Parse sid_{cur} as (pid, sid). $\text{pidowner} \leftarrow \text{pid}$. 	
Main:	
See Figure 4.5.	

^aBy sending the message responsively, the adversary is forced to provide the expected answer before interacting with the protocol again. This makes the definition of $\mathcal{F}_{\text{sig-CA}}$ simpler as we do not have to specify what happens if the adversary does not provide the expected response. It also makes $\mathcal{F}_{\text{sig-CA}}$ easier to use for higher-level protocols as they get the guarantee that, when they send, e.g., a **Sign** request, the response will be returned immediately without the adversary being able to interfere in unexpected ways (just as is the case in an actual implementation of a signature scheme). This in turn simplifies proofs as there are less edge cases to consider. See also the discussions in Chapter 3.

Figure 4.4.: The ideal signature protocol $\mathcal{F}_{\text{sig-CA}}$ with public key infrastructure.

Description of $M_{\text{signer,verifier}}$ (continued):

Main:	
recv InitSign from I/O to (pidowner, -, signer): send (Register, pk) to (pid _{cur} , ϵ , \mathcal{F}_{CA} : registration); wait for (Register, -). KeysGenerated \leftarrow ready. reply (InitSign, success, pk).	<i>{ Only the owner of the key can create (and use) his signing key.</i> <i>{ Successful initialization. Note that signer can submit InitSign multiple times, always with the same effect.</i>
recv (Sign, m) from I/O to (pidowner, -, signer) s.t. KeysGenerated = ready: $\sigma \leftarrow \text{sig}^{(p)}(m, \text{sk})$. $b \leftarrow \text{ver}^{(p)}(m, \sigma, \text{pk})$. if $\sigma = \perp \vee b \neq \text{true}$: reply (Signature, \perp). else: add m to msglist. reply (Signature, σ).	<i>{ Run at most polynomially many steps of sig. See Section 4.2.6 for a formal definition.</i> <i>{ Sign and check that verification succeeds.</i> <i>{ Signing or verification test failed.</i> <i>{ Record m for verification and return signature.</i>
recv (Verify, m, σ, pk) from I/O to (-, -, verifier): $b \leftarrow \text{ver}^{(p)}(m, \sigma, pk)$. if $pk = \text{pk} \wedge b = \text{true} \wedge m \notin \text{msglist} \wedge (\text{pidowner}, \text{sid}_{\text{cur}}, \text{signer}) \notin \text{CorruptionSet}$: reply (VerResult, false). else: reply (VerResult, b).	<i>{ Verify signature.</i> <i>{ cf. Section 4.2.3 for CorruptionSet.</i> <i>{ Prevent forgery.</i> <i>{ Return verification result.</i>

Figure 4.5.: The ideal signature protocol $\mathcal{F}_{\text{sig-CA}}$ with public key infrastructure (continued).

Description of the protocol $\mathcal{F}_{CA} = (\text{registration}, \text{retrieval})$:

Participating roles: {registration, retrieval}	
Corruption model: incorruptible	
Description of $M_{\text{registration, retrieval}}$:	
Implemented role(s): {registration, retrieval}	
Internal state:	
- keys : $(\{0, 1\}^*)^2 \rightarrow \{0, 1\}^* \cup \{\perp\}$	<i>{ Mapping from a tuple (PID, SID) to stored keys; initially \perp.</i>
CheckID ($pid, sid, role$): Accept all entities.	<i>{ By this there is only a single machine instance that manages all entities.</i>
Main:	
recv (Register, key) from I/O to (-, -, registration): if keys[pid _{call} , sid _{call}] $\neq \perp$: reply (Register, failed). else: keys[pid _{call} , sid _{call}] = key reply (Register, success).	<i>{ Allows every higher-level protocol that connects to the registration role to register a key. The key is stored for the PID and SID of the caller of \mathcal{F}_{CA}.</i>
recv (Retrieve, (pid, sid)) from - to (-, -, retrieval): reply (Retrieve, keys[pid, sid]).	<i>{ Everyone, including NET, can retrieve keys registered by someone with PID pid and SID sid.</i>

Figure 4.6.: The ideal certificate authority functionality \mathcal{F}_{CA} .

Description of the protocol $\mathcal{P}_{\text{sig-CA}} = (\text{signer}, \text{verifier})$:

Participating roles: signer, verifier
Corruption model: dynamic with secure erasures
Protocol parameters:
 – an EUF-CMA signature scheme $\Sigma = (\text{gen}, \text{sig}, \text{ver})$.

Description of M_{signer} :

Implemented role(s): signer
Subroutines: \mathcal{F}_{CA} : registration
Internal state:
 – $(\text{sk}, \text{pk}) \in (\{0, 1\}^* \cup \{\perp\})^2 = (\perp, \perp)$. *{key pair.}*
 – $\text{pidowner} \in \{0, 1\}^* \cup \{\perp\} = \perp$. *{Party ID of the key owner.}*
 – $\text{KeysGenerated} \in \{\text{ready}, \perp\} = \perp$. *{Has signer initialized his key?}*
CheckID($\text{pid}, \text{sid}, \text{role}$):
 Check that $\text{sid} = (\text{pid}', \text{sid}')$. If that check fails, output **reject**.
 Otherwise, accept a single entity. *{An instance manages exactly one entity.}*
Corruption behavior:
 – **AllowAdvMessage**($\text{pid}, \text{sid}, \text{role}, \text{pid}_{\text{receiver}}, \text{sid}_{\text{receiver}}, \text{role}_{\text{receiver}}, m$):
 Check that $\text{pid} = \text{pid}_{\text{receiver}}$.
 If $\text{role}_{\text{receiver}} = \mathcal{F}_{\text{CA}}$: registration, also check that $\text{sid} = (\text{pid}, \text{sid}')$ (for some sid').
 If all checks succeed, output **true**, otherwise output **false**.
Initialization:
 $(\text{sk}, \text{pk}) \leftarrow \text{gen}(1^n)$.
 Parse sid_{cur} as (pid, sid) .
 $\text{pidowner} \leftarrow \text{pid}$.
Main:
recv InitSign from I/O to $(\text{pidowner}, \rightarrow, \rightarrow)$:
 send (Register, pk) to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{CA}}$: registration);
 wait for (Register, \rightarrow).
 KeysGenerated \leftarrow ready.
 reply (InitSign, success, pk).
recv (Sign, m) from I/O to $(\text{pidowner}, \rightarrow, \rightarrow)$ s.t. KeysGenerated = ready:
 $\sigma \leftarrow \text{sig}(m, \text{sk})$. *{Sign m, return signature.}*
 reply (Signature, σ).

Description of M_{verifier} :

Implemented role(s): verifier
CheckID($\text{pid}, \text{sid}, \text{role}$):
 Check that $\text{sid} = (\text{pid}', \text{sid}')$. If that check fails, output **reject**.
 Otherwise, accept a single entity. *{An instance manages exactly one entity.}*
Main:
recv (Verify, m, σ, pk) from I/O:
 $b \leftarrow \text{ver}(m, \sigma, \text{pk})$.
 reply (VerResult, b). *{Verify, return result.}*

Figure 4.7.: The real signature protocol $\mathcal{P}_{\text{sig-CA}}$ with public key infrastructure.

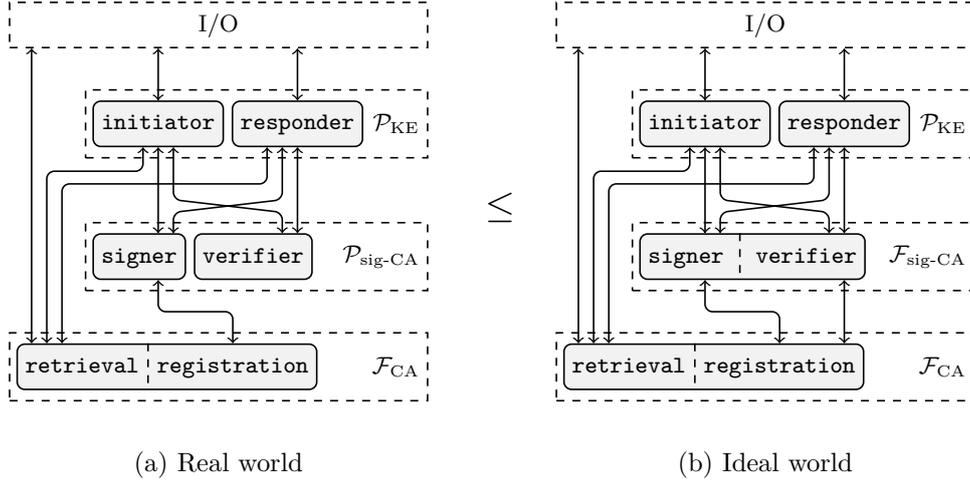


Figure 4.8.: The static structures of a key exchange protocol \mathcal{P}_{KE} using the ideal signature functionality with public key infrastructure $\mathcal{F}_{\text{sig-CA}}$ (right side) respectively its realization $\mathcal{P}_{\text{sig-CA}}$ (left side).

The realization $\mathcal{P}_{\text{sig-CA}}$ is a straightforward implementation of a digital signature scheme $\Sigma = (\text{gen}, \text{sig}, \text{ver})$, i.e., the **signer** role uses the signing algorithm to create signatures and the **verifier** role runs the verification algorithm to verify signatures. Note that the **signer** and **verifier** roles are implemented via separate machines, and each instance of those machines manages exactly one entity. Hence, different entities do not share any state. This directly models an actual digital signature scheme from reality, where signature generation and verification are independent and each party runs the verification algorithm on their own separate computer.

One can show the following for the realization $\mathcal{P}_{\text{sig-CA}}$:

Lemma 4.1. *If the digital signature scheme Σ used in $\mathcal{P}_{\text{sig-CA}}$ is existentially unforgeable under chosen message attacks (EUF-CMA-secure, cf. Appendix E.4), then*

$$\begin{aligned}
 &(\mathcal{P}_{\text{sig-CA}}, \mathcal{F}_{\text{CA}} : \text{retrieval} \mid \mathcal{F}_{\text{CA}} : \text{registration}) \\
 &\leq_R (\mathcal{F}_{\text{sig-CA}}, \mathcal{F}_{\text{CA}} : \text{retrieval} \mid \mathcal{F}_{\text{CA}} : \text{registration}) .
 \end{aligned}$$

Proof. Analogous to the proof in [94]. □

Now consider a higher-level protocol that uses $\mathcal{F}_{\text{sig-CA}}$, say, a key exchange protocol \mathcal{P}_{KE} with an **initiator** and **responder** role (a possible structure of such a protocol is shown in Figure 4.8). We can use the main composition theorem (cf. Corollary 4.1) to obtain that the protocol \mathcal{P}_{KE} using an actual signature scheme $\mathcal{P}_{\text{sig-CA}}$ realizes (and hence is as secure as) \mathcal{P}_{KE} using an ideal signature scheme $\mathcal{F}_{\text{sig-CA}}$:

Corollary 4.2. *If the protocol $(\mathcal{P}_{\text{KE}}, \mathcal{F}_{\text{CA}} : \text{retrieval} \mid \mathcal{P}_{\text{sig-CA}}, \mathcal{F}_{\text{CA}} : \text{registration})$ is environmentally bounded (and complete) and the conditions of Lemma 4.1 are fulfilled, then*

$$\begin{aligned}
& (\mathcal{P}_{KE}, \mathcal{F}_{CA} : \text{retrieval} \mid \mathcal{P}_{\text{sig-CA}}, \mathcal{F}_{CA} : \text{registration}) \\
& \leq (\mathcal{P}_{KE}, \mathcal{F}_{CA} : \text{retrieval} \mid \mathcal{F}_{\text{sig-CA}}, \mathcal{F}_{CA} : \text{registration}) .
\end{aligned}$$

Note that the **initiator** and **responder** roles can also connect to the public **retrieval** role (as shown in Figure 4.8), but do not have to. We further note that in the above theorem we chose to make the **signer** and **verifier** roles private, whereas the **retrieval** role remains public. While this is typically the most reasonable choice for key exchange protocols, the composition theorem also allows for other combinations such as making all of those roles public.

Remark 4.2. In the above formulation of $\mathcal{F}_{\text{sig-CA}}$ and \mathcal{F}_{CA} , we chose to make the **registration** role private, hence limiting access to key registration only to $\mathcal{F}_{\text{sig-CA}}$. This has the advantage that another protocol using $\mathcal{F}_{\text{sig-CA}}$, such as \mathcal{P}_{KE} from the above example, can assume that no other unknown protocols or the environment interfere with the protocol run by registering malicious keys for honest parties. This reflects the best practice of certifying and using keys only within a specific context and is in fact a necessary requirement for showing any form of security result for most protocols based on digital signatures.

It might seem that a downside of this modeling is that two separate protocols cannot directly use *the same* global certificate authority \mathcal{F}_{CA} to register their keys; this can be a desirable modeling since CAs typically manage keys for multiple protocols in practice. However, due to the flexibility of iUC, it is actually quite easy to extend security results also to the case where *a single global CA manages keys from multiple protocols*: Suppose one has already analyzed and proven secure a protocol \mathcal{P} relying on $\mathcal{F}_{\text{sig-CA}}$ and \mathcal{F}_{CA} . To share \mathcal{F}_{CA} with another protocol, say \mathcal{Q} , one first analyzes \mathcal{Q} with another copy of \mathcal{F}_{CA} (and $\mathcal{F}_{\text{sig-CA}}$), say \mathcal{F}'_{CA} (and $\mathcal{F}'_{\text{sig-CA}}$). After proving \mathcal{Q} to be secure, which is simpler than directly using a global CA where multiple protocols register keys, one can replace both \mathcal{F}_{CA} and \mathcal{F}'_{CA} with a joint state realization (using our general composition theorem, cf. Corollary 4.1) where keys are stored in and retrieved from the same \mathcal{F}_{CA} subroutine along with a protocol dependent tag (see also Section 4.4.2 for this novel type of joint state realization). Note that this approach can be iterated to support arbitrarily many protocols using the same \mathcal{F}_{CA} . This joint state realization again reflects the best practice mentioned above, where keys are certified for a specific purpose which is represented by the tag.

Remark 4.3. Our functionality \mathcal{F}_{CA} follows a similar idea as the existing global functionality \mathcal{G}_{bb} for certificate authorities by Canetti et al. [52]. Both functionalities allow users of higher-level protocols to register some bit string in their name which other users can then retrieve. A key difference is due to the concept of public and private roles provided by iUC, which allows for limiting access to key registration in \mathcal{F}_{CA} in the context of specific higher-level protocols. In particular, when used as a subroutine in the protocol $(\mathcal{F}_{\text{sig-CA}}, \mathcal{F}_{CA} : \text{retrieval} \mid \mathcal{F}_{CA} : \text{registration})$ as specified above, then we get the expected guarantee that the environment cannot register malicious keys in the name of honest users of $\mathcal{F}_{\text{sig-CA}}$. As already explained above, this is typically a requirement to be able to show any form of security results for higher-level protocols such as key exchanges.

In contrast to \mathcal{F}_{CA} in the iUC framework, the global functionality \mathcal{G}_{bb} in the UC model does not support making parts of the functionality “private”. Thus, everyone has full access to all operations of \mathcal{G}_{bb} , including key registration, allowing the environment to register keys in the name of (almost)³⁵ arbitrary users, even if they are supposed to be honest. This makes \mathcal{G}_{bb} harder to use by higher-level protocols compared to our functionality \mathcal{F}_{CA} .

³⁵The GUC extension for global state in UC, which is based on the UC version from 2005, does not contain a provision for restricting the environment in terms of accessing the global functionality. However, the UC model from 2013, which was the current version when \mathcal{G}_{bb} was published, prevents the environment from accessing protocols in the name of users that are part of the challenge session. Assuming that the GUC extension can also be applied to the 2013 version of UC, which has not been formally shown so far, then this essentially means that the environment may not register keys in the name of users that have a certain arbitrary but fixed SID. However, even then the environment is still free to register keys in the name of honest users with arbitrary other SIDs, which can in turn disrupt the security of higher-level protocols that either use \mathcal{G}_{bb} directly or, e.g., use an ideal signature functionality based on \mathcal{G}_{bb} .

4.4. Discussion

Recall from the introduction (cf. Chapter 1) that a major goal of and challenge in creating the iUC framework is to provide a flexible yet easy to use framework for universally composable protocol analysis and design. In this section, we briefly summarize and highlight some of the core concepts that allow us to retain the flexibility and expressiveness of the IITM model while drastically improving its usability with a handy set of conventions. We then highlight a selection of features that are supported by iUC, many of which are not supported by other (conventions of) models, including the prominent UC and GNUC models. Our case study in Chapter 5 further illustrates the expressiveness of iUC while remaining easy to use.

4.4.1. Core Concepts

The most crucial concepts of iUC, discussed next, are the separation of entities and machine instances, public and private roles, a model independent interpretation of SIDs, a highly customizable corruption mechanism, support for responsive environments as well as a general addressing mechanism, which enables some of these concepts.

Separation of entities and machine instances: Traditionally, universal compossibility models do not distinguish between a machine instance and its interpretation. Instead, they specify that, e.g., a *real protocol instance* always represents a single party in a single session running a specific piece of code. Sometimes even composition theorems depend on this view. This has the major downside that, if the interpretation of a machine instance needs to be changed, then existing models, conventions, and composition theorems are no longer applicable and have to be redefined (and, in the case of theorems, reproven). For example, a typical *joint state protocol instance* [51, 75, 94] manages a single party in *all sessions* and one role. Thus, in the case of the UC and GNUC models which formulated their models and theorems based on a different interpretation of protocol instances, the models had to be extended and reproven to support this type of joint state protocols, which includes new conventions and composition theorems. This is in contrast to iUC, which introduces the concept of *entities*. A protocol designer can freely define the interpretation of a machine instance by specifying the set of entities managed by that instance; the resulting protocol is still supported by our single template and the main composition theorem. This is a crucial feature that allows for the unified handling of real, ideal, joint state, and (in combination with the next concept) also global state protocols.

We emphasize that this generality is made possible by the highly customizable addressing mechanism (**CheckID** in the template) used in iUC, which in turn is based on the very general addressing mechanism of the IITM model.

Public and private roles: Similar to the previous point, traditionally global state is defined by adding a special new global functionality with its own sets of conventions and then proving specific global state composition theorems [41, 52, 75]. However, whether or not state is global is

essentially just a matter of access to that state. Our framework captures this property via the natural concept of *public roles*, which provides a straightforward way to make parts of a protocol accessible to the environment and other protocols. Thus, there is actually no difference between protocols with and without global state in terms of conventions or composition theorems in our framework.

Model independent interpretation of SIDs: In most other models, such as UC and GNUC, SIDs play a crucial role in the composition theorems. Composition theorems in these frameworks require protocols to either have disjoint sessions, where a session is defined via the SID, or at least behave as if they had disjoint sessions (in the case of joint state composition theorems). This has two major implications: Firstly, one cannot directly model a protocol where different sessions share the same state and influence each other. This, however, is often the case for real world protocols that were not built with session separation in mind. For example, many protocols such as those from our case study (cf. Section 5.3) use the same signing key in multiple sessions, but do not include a session specific SID in the signature (as would be required for a joint state realization). Secondly, sessions in ideal functionalities can consist only of parties sharing the same SID, which models so-called *global SIDs* (also called *pre-shared* or *pre-established SIDs*) [96]. That is, participants of a protocol session must share the same SID. This is in contrast to so-called *local SIDs* often used in practice, where participants with different SIDs can be part of the same protocol session (see Section 4.4.3 below for a more detailed explanation of global and local SIDs as well as their implications for security results). Because our main composition theorem is independent of (the interpretation of) SIDs, and in particular does not require state separation, we can also capture both shared state and local SIDs in our framework.

Just as for the concept of entities and instances, this flexibility is made possible by the general addressing mechanism of iUC (inherited from the underlying IITM model).

Highly customizable corruption mechanism: An important part of conventions for any model is to define how corruption of a protocol/a participant in a protocol is modeled. One way to do so is to fix one or several specific corruption mechanisms. For example, conventions for the GNUC model define a dynamic top-down corruption mechanism, where subroutines can only be corrupted after their respective higher-level protocol has been corrupted. This, however, is rather inflexible and forces protocol designers to break out of the existing conventions every time they want to model a different type of corruption such as, say, a higher-level protocol that can only be corrupted after all of its subroutines have been corrupted (this would be useful to model redundant subroutines). The iUC framework preserves a high degree of flexibility by allowing for customizing each individual step of corrupting and controlling individual entities, and hence iUC fully supports a wide variety of corruption models.

Support for responsive environments: iUC fully supports and makes use of the concept of responsive environments and adversaries introduced in Chapter 3. For example, entities in our

framework request their initial corruption status via a restricting message. Hence, the adversary has to provide the corruption status right away and the protocol run can continue as expected. Besides such restricting messages employed by the iUC framework itself, the protocol designer is also free to send his own restricting messages. Altogether, as explained in detail Chapter 3, this drastically simplifies protocol specifications and security proofs in iUC as one does not have to worry about edge cases that can lead to artificial purely modeling-related attacks.

The iUC framework uses and combines the above core concepts to support a wide range of protocols and composition types using just *a single template* and *one main composition theorem*. In what follows we list and discuss some important examples and use cases.

4.4.2. Composition Types

The general composition theorem, which in particular is agnostic to the specific protocols at hand, allows for combining and mixing, e.g., composition of protocols with disjoint session, composition of joint state protocols, composition of protocols with global state, composition of protocols with arbitrarily shared state, and realizations of global functionalities. iUC even supports new types of compositions that have not been considered in the literature so far, such as joint state realizations of two separate independent protocols (in contrast to traditional joint state realizations of multiple independent sessions of the same protocol). This sets iUC apart from most other UC-like models, which have more restricted composition theorems that work only for specific types of protocols that meet certain assumptions and have a special structure. We discuss various forms of composition in more detail in what follows.

Single Session Composition

While concurrent/parallel composition is the main compositional operation (cf. Section 4.2.5 and Corollary 4.1), our framework in addition also supports so-called single session composition (also called unbounded self-composition). Single session composition works on protocols that are built in such a way that instances of those protocols can be grouped into several disjoint sessions such that instances from different sessions do not share any state and do not interact with each other. For such protocols, the single session composition theorem, intuitively, states that if one session of a protocol \mathcal{P} realizes one session of a protocol \mathcal{F} , then an unlimited number of sessions of \mathcal{P} realizes an unlimited number of sessions of \mathcal{F} .

By default, our framework does not enforce a protocol structure with disjoint sessions (unlike many other universal composability models that assume disjoint sessions). On the contrary, we do not restrict the protocol designer and instead allow for, e.g., sharing arbitrary state between several protocol sessions by being able to manage several entities in the same instance of a machine. However, if so desired, one can easily define a protocol with disjoint sessions. On a high level, such a protocol has to ensure that (i) no instance accepts entities from different sessions (which makes the state disjoint) and (ii) entities send messages to other entities only

in the same session (which prevents interactions between different sessions). These properties are easy to obtain via appropriate definitions of the algorithms in our template:

For (i), one uses the **CheckID** algorithm to specify machine instances that do not accept entities from two or more different “sessions”, where a “session” can be defined in many different ways. For example, if one considers a “session” to be defined by a single shared SID, as is common in the UC and GNUC models, one can define **CheckID** such that all entities using the same SID are accepted by one instance. This essentially creates what is called an ideal functionality in the UC and GNUC models, which uses a single instance to manage all parties in a single session and fulfills (i). Alternatively, one can also, e.g., use the default definition of **CheckID** that accepts a single unique entity per instance and which creates what is called a real protocol in the UC and GNUC models. For (ii), one uses appropriate definitions of the **Initialization**, **EntityInitialization**, **MessagePreprocessing**, **Main**, and **AllowAdvMessage** algorithms that send messages to other entities only if they are in the same session.

For protocols defined in such a way, our framework offers the following single session composition theorem:

Corollary 4.3 (Unbounded self-composition (informal)). *Let \mathcal{P} and \mathcal{F} be two complete protocols such that they are environmentally bounded and \mathcal{P} realizes \mathcal{F} in a single session. That is, the environment may send messages only to a single session of \mathcal{P} or \mathcal{F} , respectively. Then $\mathcal{P} \leq \mathcal{F}$, i.e., \mathcal{P} realizes \mathcal{F} in an unbounded number of sessions.*

Proof. This is a direct implication of the unbounded self-composition theorem of the IITM model (cf. Theorem 3.4). See also the formalized version of this theorem in Appendix B, which includes a detailed argument. \square

We provide a more detailed description of single session composition, including a formal version of the above theorem and an example of a protocol with disjoint sessions, in Appendix B. Note that, just as for concurrent composition, we do not require a specific internal structure of \mathcal{P} and \mathcal{F} besides session disjointness. In addition, the definition of a “session” is not fixed and can be determined depending on the type of protocol. For example, sometimes it can be useful to define a “session” to comprise all entities that share the same prefix of their SIDs, or share the same (prefix of the) PID. This is in contrast to many other UC-like models, including the UC and GNUC models, that fix in their model and theorems how a “session” is defined.

Joint State Composition for Multiple Sessions

Modeling protocols with disjoint sessions (see above) is not always realistic. In many cases, some kind of state should be shared between instances of the same machine in different sessions. For example, cryptographic keys for signing and verifying messages are generally supposed to be reused across multiple sessions of a protocol. In order to be able to also capture these settings in universal composability models that assume disjoint sessions, the concept of joint state composition was introduced [51, 75, 94]. A joint state composition theorem intuitively

states that a protocol \mathcal{F} with disjoint sessions can be replaced by another protocol \mathcal{P} that shares state between multiple sessions if no environment can distinguish both cases.

Supporting joint state composition in models that assume disjoint sessions to be the default entails introducing a new set of syntax constructs, defining a new realization relation, and defining (and proving) an entirely new composition theorem. In contrast, our framework supports joint state seamlessly and out of the box, without needing any modifications or new theorems: as already mentioned above, protocols are able to share state by default, so no new syntax is necessary. To give an example, consider some protocol \mathcal{F} with disjoint sessions where an instance of \mathcal{F} manages all entities sharing a single SID. Such a protocol can reuse, e.g., some cryptographic key across multiple entities with the same SID, however, since different SIDs are handled by different instances, every session will use a different key. Now, one can define a joint state realization \mathcal{P}_{js} of \mathcal{F} (with the same public roles) where one instance of \mathcal{P}_{js} accepts *all* entities, also in different sessions. In \mathcal{P}_{js} , one can then use *the same* key for entities in multiple sessions as all entities are handled by the same instance. For a more concrete example of a joint state realization in our framework, we refer the reader to Appendix C.

In general, we say that a protocol \mathcal{P}_{js} has joint state if it shares some state between sessions, but is still supposed to behave just like a protocol \mathcal{F} that does not share state between sessions. This notion of a protocol with joint state is just a special case of the very general protocol definition of iUC, unlike in many other models. Hence, once we have shown that \mathcal{P}_{js} realizes \mathcal{F} (as per Definition 4.1), we can directly apply our main composition theorem (Corollary 4.1) as follows, where clearly this corollary applies no matter which inner structure \mathcal{P}_{js} has, and in fact, no matter whether it is a joint state protocol or not:

Corollary 4.4 (Concurrent composition with joint state). *Let \mathcal{P}_{js} be an environmentally bounded protocol with joint state and \mathcal{F} be an environmentally bounded protocol with disjoint sessions such that $\mathcal{P}_{js} \leq \mathcal{F}$. Let \mathcal{Q} be another protocol such that \mathcal{Q} and \mathcal{F} are connectable. Let $\mathcal{R} \in \text{Comb}(\mathcal{Q}, \mathcal{P}_{js})$ and let $\mathcal{I} \in \text{Comb}(\mathcal{Q}, \mathcal{F})$ such that \mathcal{R} and \mathcal{I} have the same sets of public roles.*

If \mathcal{R} is environmentally bounded, then

$$\mathcal{R} \leq \mathcal{I} \quad .$$

Proof. This is a mere special case of Corollary 4.1 and as such trivially implied. \square

Because both disjoint sessions and joint state are mere special cases in our framework, both the realization relation and the concurrent composition theorem remain unchanged. In particular, we do not have to change any of the definitions, syntax, theorems, or introduce additional requirements. Overall, this drastically simplifies the handling of joint state for protocol designers as they are able to work with the same syntax, definitions, and theorems. Also, by this, joint state can seamlessly be combined with other concepts (e.g., other forms of shared state, including global state).

We want to highlight that the corruption model of our framework is fully compatible with joint state in the sense that one can actually prove realizations. To understand why this is a non-trivial feature, consider an ideal signature functionality \mathcal{F}_{sig} with disjoint sessions and a potential joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ that reuses the same signature key in all sessions. Now, if an adversary corrupts the single signing key in $\mathcal{P}_{\text{sig}}^{\text{js}}$ and can thus forge messages for all sessions, this corresponds to infinitely many corrupted sessions in \mathcal{F}_{sig} . A simulator must be able to perform all of those corruptions, even though he has only polynomial runtime. Our framework deals with this issue by asking for the corruption state of newly created entities before any other operations are performed, i.e., the simulator does not have to pro-actively corrupt non-existing entities but can instead act re-actively when the environment triggers a new entity for the first time. Conversely, if an entity has not been initialized yet, it always returns `false` to `CorruptionStatus?` requests and hence behaves identical in both the joint state realization and the ideal functionality.

Joint State Composition for Multiple Protocols

So far, most of the literature has considered only the above type of joint state where a single instance of *one* machine realizes multiple instances of *one* machine.

Our framework, however, also supports various other types of joint state composition via the standard concurrent composition theorem (cf. Corollary 4.1). It, for instance, allows one to use one instance of *one* machine to realize instances of *multiple* different machines. For example, one can use a protocol \mathcal{P} to realize the combination of two protocols $\mathcal{F} \parallel \mathcal{F}'$, where one machine instance of \mathcal{P} realizes both an instance of \mathcal{F} and an instance of \mathcal{F}' . The concurrent composition theorem then implies that $\mathcal{Q} \parallel \mathcal{P}$ realizes $\mathcal{Q} \parallel \mathcal{F} \parallel \mathcal{F}'$ (for an arbitrary protocol \mathcal{Q}), i.e., we can replace multiple independent protocols by a single one that is able to reuse some state across different invocations.

To illustrate when and why this type of joint state is useful, consider the following example (another example was already given in Remark 4.2 in Section 4.3). Let \mathcal{R} be some higher-level protocol, such as a key exchange, using an ideal signature protocol $\mathcal{F}_{\text{sig}} = (\text{signer}, \text{verifier})$ as a subroutine. Analogously, let \mathcal{R}' be a different higher-level protocol also using an ideal signature protocol $\mathcal{F}'_{\text{sig}}$ as a subroutine (where \mathcal{F}_{sig} and $\mathcal{F}'_{\text{sig}}$ have the same machine code). The combined protocols $\mathcal{R} \parallel \mathcal{F}_{\text{sig}}$ and $\mathcal{R}' \parallel \mathcal{F}'_{\text{sig}}$ can be analyzed in isolation and proven to be secure. Now, the composition theorem implies that these protocols running concurrently, i.e., the combined protocol $\mathcal{R} \parallel \mathcal{R}' \parallel \mathcal{F}_{\text{sig}} \parallel \mathcal{F}'_{\text{sig}}$, are still secure. This combination contains two separate subroutines \mathcal{F}_{sig} and $\mathcal{F}'_{\text{sig}}$ that do not share any state between each other, i.e., \mathcal{R} and \mathcal{R}' use different signature keys. In some cases, one would like to obtain security even if \mathcal{R} and \mathcal{R}' use the same signature key; intuitively, this should be possible if \mathcal{R} and \mathcal{R}' sign messages from disjoint messages spaces such that signatures by \mathcal{R} do not impact \mathcal{R}' and vice-versa.

Our framework allows for showing this expected security result via an appropriate joint state realization and Corollary 4.1: one defines a joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ that has the same public

roles as $\mathcal{F}_{\text{sig}} \parallel \mathcal{F}'_{\text{sig}}$, i.e., two **signer** and two **verifier** roles. Internally, $\mathcal{P}_{\text{sig}}^{\text{js}}$ accepts all entities in one machine instance which then acts as a multiplexer for a single subroutine $\mathcal{F}''_{\text{sig}}$ (that, again, uses the same code as \mathcal{F}_{sig}). More specifically, signing requests arriving for any of the signer roles of $\mathcal{P}_{\text{sig}}^{\text{js}}$ are prefixed with a unique ID, depending on the role where they arrived, and then forwarded to the **signer** role of $\mathcal{F}''_{\text{sig}}$. In other words, $\mathcal{P}_{\text{sig}}^{\text{js}}$ uses the same subroutine and thus the same signing key for signing requests arriving for both public **signer** roles. Once we have shown that $\mathcal{P}_{\text{sig}}^{\text{js}} \parallel \mathcal{F}''_{\text{sig}} \leq \mathcal{F}_{\text{sig}} \parallel \mathcal{F}'_{\text{sig}}$, we can use the concurrent composition theorem (cf. Corollary 4.1) to conclude that $\mathcal{R} \parallel \mathcal{R}' \parallel \mathcal{P}_{\text{sig}}^{\text{js}} \parallel \mathcal{F}''_{\text{sig}} \leq \mathcal{R} \parallel \mathcal{R}' \parallel \mathcal{F}_{\text{sig}} \parallel \mathcal{F}'_{\text{sig}}$. Thus we can use the same signing key for both protocols and still retain security by adding unique prefixes to keep the message spaces of each protocol disjoint (as is common in many real world protocols). See also Remark 4.2 in Section 4.3 for another example of a joint state realization of multiple separate protocols.

We note that, while the above example was given for functionalities $\mathcal{F} = \mathcal{F}_{\text{sig}}$ and $\mathcal{F}' = \mathcal{F}'_{\text{sig}}$ that share the same machine code, this is by no means a requirement. The iUC framework clearly also supports joint state realizations of separate protocols \mathcal{F} and \mathcal{F}' with different machine code using the same technique. For example, if \mathcal{F} and \mathcal{F}' are two random oracles that output ideal hashes with different lengths and therefore have different machine code, the iUC framework still allows for replacing both oracles with a joint state realization that uses only a single random oracle \mathcal{F}'' , where the realization is analogous to the one for \mathcal{F}_{sig} and $\mathcal{F}'_{\text{sig}}$ above.

Global Functionalities and Global State

Sometimes it is desirable to define a protocol in such a way that it exposes some of its subroutines to other protocols, the idea being that some state can or should be shared with other arbitrary (and unknown) protocols. For example, a common reference string (CRS) is generally considered to be a publicly known resource, so it seems natural to make it globally available instead of modeling it as an internal subroutine that no other protocols can see or access. Another example is a subroutine modeling a public key infrastructure based on certificate authorities which should make public keys accessible for every protocol, not just one specific protocol (our example in Section 4.3 includes such a global public key infrastructure).

Just as for joint state, most universal composability models had to introduce additional extensions to model so-called global state [41, 52, 75]. This entails additional syntax, changes to the definition of the realization relation, and introducing (potentially multiple) new composition theorems. In contrast, again, our framework seamlessly supports global state out of the box without any modifications to definitions or syntax. This is due to the built-in concept of public and private roles: having a globally available subroutine is as simple as making a role public. For example, consider a protocol $\mathcal{F}_{\text{CRS}} = (\text{retrieveCRS})$ with a single role modeling a CRS, and a higher-level protocol $\mathcal{P} = (\text{somePublicRole} \mid \text{somePrivateRole})$ using \mathcal{F}_{CRS} as a subroutine. If one wants to model a CRS that is only locally available to \mathcal{P} , then one considers the combined protocol $(\text{somePublicRole} \mid \text{somePrivateRole}, \text{retrieveCRS})$ (which can also be written as

($\mathcal{P} \mid \mathcal{F}_{\text{CRS}}$) using the shorter notation from Section 4.2.4); to model a global CRS, one considers the combined protocol (`somePublicRole, retrieveCRS` | `somePrivateRole`) instead (which can also be written as $(\mathcal{P}, \mathcal{F}_{\text{CRS}})$).³⁶

In general, we say that a protocol \mathcal{P} has global state if (one or more) subroutine roles of \mathcal{P} are public and hence allow the environment to access state stored in those subroutines. Just as for protocols with joint state, a protocol with global state is again a special case of the very general protocol definition of iUC, unlike in many other models. Hence, once we have shown that a protocol \mathcal{P} with global state realizes some other protocol \mathcal{F} , we can directly apply the main composition theorem (Corollary 4.1) as follows, where clearly this corollary applies no matter which inner structure \mathcal{P} has, and in fact, no matter whether it is a protocol with or without global state:

Corollary 4.5 (Concurrent composition with global state). *Let \mathcal{P} and \mathcal{F} be two environmentally bounded protocols with global state such that $\mathcal{P} \leq \mathcal{F}$. Let \mathcal{Q} be another protocol such that \mathcal{Q} and \mathcal{F} are connectable. Let $\mathcal{R} \in \text{Comb}(\mathcal{Q}, \mathcal{P})$ and let $\mathcal{I} \in \text{Comb}(\mathcal{Q}, \mathcal{F})$ such that \mathcal{R} and \mathcal{I} have the same sets of public roles. If \mathcal{R} is environmentally bounded, then*

$$\mathcal{R} \leq \mathcal{I} \quad .$$

Proof. This is a mere special case of Corollary 4.1 and as such trivially implied. □

Again, just as for joint state, we emphasize that both the realization relation and the composition theorem remain unchanged. We do not have to change any of the definitions, syntax, theorems, or introduce additional requirements, which makes global state in our framework very user friendly.

We note that global state in our framework is particularly flexible, also compared to global state in other UC-like models, due to our concept of public and private roles. For example, it is possible to make only parts of a protocol publicly available, instead of the full protocol. Our example in Section 4.3 makes use of this feature to define a globally available public key infrastructure where key registration is protected. Furthermore, since public roles can be changed to be private when combined with another protocol, one can actually change global subroutines to be only locally available to a single protocol while retaining all security results and while still being able to use the composition theorem to replace that subroutine with its realization. This can be useful if, e.g., a subroutine has already been proven to be secure based on a globally available CRS, but a higher-level protocol using that subroutine, including the same CRS, can only be proven to be secure for a locally available CRS.

³⁶Locally available CRSs have different security properties than globally available ones as the simulator is able to determine a locally available CRS in the simulation. Hence, both local and global CRSs are interesting cases for security analyses. The difference is discussed in detail in [41].

Combinations of the Above

All of the above types of composition can easily be combined with each other. Here we give two examples that illustrate such combinations and further show the power and generality of our composition theorem.

It is possible to realize a global functionality with another protocol in iUC, i.e., to combine composition with global state with traditional composition. For example, assume we have a protocol \mathcal{P} using a global common reference functionality \mathcal{G}_{CRS} that provides one CRS per SID. Suppose we want to use a hash function modeled via a random oracle instead. So we could first show that we can realize \mathcal{G}_{CRS} via a global random oracle \mathcal{G}_{RO} that uses the current SID as input, hence providing independent random bit strings for each session. Then, our main composition theorem (cf. Corollary 4.1) directly implies that \mathcal{P} using \mathcal{G}_{RO} realizes \mathcal{P} using \mathcal{G}_{CRS} , i.e., all security results obtained for a CRS still hold true in an implementation via an (idealized) hash function. While this is just one simple example and the resulting statement seems expected, other models have struggled with being able to capture this natural statement. For example, the GUC extension for global state in the UC model needed an entirely new composition theorem for realizing global functionalities [52], and the GNUC model does not currently support this operation. In contrast, in iUC this is a mere special case of our main composition theorem.

The iUC framework also seamlessly supports combinations of joint state and global functionalities. For example, one can realize multiple global functionalities, where each of them might even use different machine codes, with a joint state realization that uses just a single global functionality using our main composition theorem (cf. Corollary 4.1). This was already explained in Remark 4.2 in Section 4.3, where we sketched how multiple separate global \mathcal{F}_{CA} functionalities can be realized with a single global CA functionality \mathcal{F}_{CA} . This type of composition has not been considered in the literature so far and is also not yet supported by the GNUC and UC models; even the extension of the UC model proposed in [52] only allows for replacing *one* global functionality with *one* other global functionality. In contrast, in iUC this is again just a mere special case of our main composition theorem.

4.4.3. Protocols

The iUC framework provides a very general and flexible protocol definition and template that allows for expressing various different types of protocols and protocol features. These types and features can all be mixed together within a single protocol, which can then be composed with other protocols via our main composition theorem. Here we give and discuss several examples of protocol types and features that iUC supports, some of which have already been sketched previously.

Protocol Types

The literature traditionally considers four main types of protocols: *real protocols*, *ideal protocols*, *protocols with joint state/joint state realizations*, and *protocols with global state*. A key difference between those classes is the interpretation of a single machine instance: One instance of a real protocol models one party in one session, an instance of an ideal protocol models multiple parties in one session, an instance of a joint state protocol models one party in multiple sessions, and an instance of a global subroutine usually models multiple parties in multiple sessions using and sharing the same state. While most other models define those four types of protocols separately, each of them with their own conventions, iUC captures all four types using the same template due to the separation of entities and instances. A protocol designer can easily create his own mapping between instances and entities using the **CheckID** algorithm, which defines the interpretation of an instance. For example, to model an ideal functionality as defined in the UC and GNUC models, one defines an instance that accepts all entities with a specific SID; to model a real protocol, instances accept only a single entity; to model a classical joint state protocol, instances accept all entities belonging to the same party.

This flexibility is not just limited to the above traditional protocol types but one is rather also able to express many other variations that have not been explored in the literature so far, such as instances that accept entities for fixed ranges of SIDs, which might be useful to model, e.g., using the same cryptographic key for a certain number of sessions before a new one is chosen.

Protocol Structure

The iUC framework provides a large degree of freedom in how protocols can be built from various connected machines. For example, the following is possible in iUC:

- Our framework supports *arbitrarily structured subroutine trees* and *protocols with shared subroutines* which can also be made *accessible to other protocols/the environment*. This is in contrast to many other models which often impose requirements on the structure of subroutines in protocols. For example, real protocols in the GNUC model are required to have a specific call tree structure, where each subroutine instance has a unique parent instance that called the subroutine. Hence, the same subroutine instance cannot be accessed by multiple parent instances. To give another example, both the UC and GNUC models require subroutines of real and ideal protocols to have disjoint protocol sessions where in particular subroutines are not shared between sessions (and also cannot be accessed by other protocols/the environment). Thus, protocol designers in those models are more limited in how (and which) protocols they can model.
- Ideal protocols in the UC and GNUC models consist of a dummy machine and a subroutine called ideal functionality. In a run, there are multiple instances of dummies with IDs (pid, sid) consisting of a PID and SID, and all dummies with the same SID share an instance of the ideal functionality with ID (\perp, sid) . The dummies are only responsible for forwarding

messages between the ideal functionality and a higher-level protocol/environment. This construction is necessary in the UC and GNUC models as there is no way to create one instance of a machine that is accessed with multiple IDs (say, all IDs belonging to the same session, but potentially different parties); instead, each ID defines a new instance. In contrast, there is *no need to include dummy machines* in iUC as we can directly define an ideal functionality that, e.g., accepts all entities from the same session, which simplifies the structure of ideal protocols.

- iUC fully supports ideal functionalities (and of course also other types of protocols) that are structured into arbitrary sub-components interacting with each other. Some of these components can also be made globally available, if so desired, which allows for mixing ideal functionalities with global state (such as in our example in Section 4.3; see also Section 5.2 for further examples of structured ideal functionalities). Compared to monolithic functionalities typically found in the literature, which use a single machine to implement all tasks of the ideal functionality, this not only makes ideal protocols easier to comprehend but also allows for replacing parts of those functionalities via the composition theorem.

Global and Local Session IDs

As already briefly mentioned in the core concepts above, most universal composability models assume protocols to have disjoint sessions, where a session is defined via the SID, and hence sessions in ideal functionalities can consist only of parties sharing the same SID. This models so-called *global SIDs*, i.e., the SID is globally known to all participants of the same session. As a result, ideal functionalities can only be realized by protocols where parties establish a session and agree on a shared global SID prior to/at the start of the actual protocol run (hence such SIDs are also often called *pre-established SIDs* or *pre-shared SIDs*). This SID has then to be used throughout the realization, e.g., by including it in signatures, such that entities with different SIDs and hence different sessions do not mix up their runs and complete a protocol run together (which is not possible in the ideal functionality).

Establishing global pre-shared SIDs is generally a good design principle for protocols and quite simple to do, e.g., by exchanging nonces prior to a protocol run, as discussed by Canetti [35] and Barak et al. [13]. However, many real world protocols, including all of the key exchange protocols in our case study (cf. Section 5.3), actually do not make use of such explicit global SIDs.³⁷ Instead, such real world protocols rely on so-called local SIDs (cf. [96]), i.e., the SID of an entity (*pid*, *sid*, *role*) models a value that is locally chosen and managed by each party *pid* and used only for locally addressing a specific instance of a protocol run of that party, but is not used as part of the actual protocol logic. In particular, multiple entities can form a “protocol session” even if they use different (local) SIDs. We emphasize that the difference between local

³⁷One reason might be that establishing an explicit SID at the start of the protocol by exchanging nonces incurs an additional round trip. Furthermore, many real world protocols, such as secure key exchanges, actually already establish some form of implicit SID, e.g., by creating a shared session key, such that an explicitly established global SID is not strictly necessary.

and global SIDs is not just a minor technicality or a cosmetic difference in how a protocol is modeled but rather has important security implications: as shown by Küsters et al. [96], there are natural protocols that are insecure when using locally chosen SIDs but become secure if a global SID for all participants in a session has already been established, i.e., security results for protocols with global SIDs do not necessarily carry over to actual implementations using local SIDs.

To facilitate the faithful and precise analysis of different kinds of protocols, our framework supports both global (pre-established) SIDs and local SIDs that are managed by each participant on their own and may very well differ between several participants in the same session (see Section 4.3 and Chapter 5 for examples of both types of SIDs). On a high level, one of the main differences is how the **CheckID** algorithm of the ideal protocol is specified: for global SIDs which are shared by all participants in the same session, an instance generally accepts entities with the same SID only. For local SIDs that need not be the same for different participants in the same session, an instance might accept entities with varying SIDs. Then, such an instance might internally group entities into new “sessions”, which models that, e.g., several entities with different local SIDs are executing a key exchange together and end up with a shared secret that dynamically determines who is part of the same “session”.

Being able to easily and faithfully model local SIDs is an important feature of iUC that sets our framework apart from many other UC-like models, including the UC and GNUC models, which are built around global SIDs and thus do not directly support local SIDs with their conventions. Modeling local SIDs in those models requires workarounds that ignore existing conventions; for example, one could model all sessions of a protocol in a single machine instance M , i.e., essentially ignoring the model’s intended SID mechanism and taking care of the addressing of different sessions with another layer of SIDs within M itself. This, however, has two major drawbacks: Firstly, it decreases overall usability of the models as this workaround is not covered by existing conventions of these models. Secondly, existing composition theorems of UC and GNUC do not allow one to compose such a protocol with a higher-level protocol modeled in the “standard way” where different sessions use different SIDs.³⁸

Shared State and Global State

In iUC, entities can easily and naturally share arbitrary state in various ways, even across multiple protocol sessions, if so desired. For example, one can define the **CheckID** algorithm to accept multiple entities, also with different SIDs, all of which then have access to the same internal state. Furthermore, entities can send and receive messages to and from arbitrary other entities (as long as their roles are connected), also with different SIDs, and hence also transmit arbitrary state to each other. State sharing is further illustrated, e.g., by the key exchanges from our case study (cf. Section 5.3), where every party uses just a single signature key pair

³⁸This is because such a higher-level protocol would then access the same subroutine session throughout many different higher-level sessions, which violates session disjointness as required by both UC and GNUC.

across arbitrarily many key exchanges. This allows for a very flexible and precise modeling of protocols. In particular, for many real world protocols this modeling is much more precise than so-called joint state realizations that are often used to share state between sessions in UC-like models that assume disjoint sessions to be the default, such as the UC and GNUC models. Joint state realizations have to modify protocols by, e.g., prefixing signed messages with some globally unique SID for every protocol session (which is not done by many real world protocols, including our case study). Thus, even if the modified protocol is proven to be secure, this does not imply security of the unmodified one. The UC and GNUC models do not directly support state sharing without resorting to joint state realizations or global functionalities. While one might be able to come up with workarounds similar to what we described for local SIDs above, this comes with the same drawbacks in terms of usability and flexibility.

Our concept of public and private roles allows us to not only easily model global state but also to specify, in a convenient and flexible way, machines that are only partially global. This is illustrated by \mathcal{F}_{CA} in our example in Section 4.3, which allows arbitrary other protocols to retrieve keys but limits key registration to one specific protocol to model that honest users will not register their signing keys for other contexts (which, in general, otherwise voids all security guarantees). This feature makes \mathcal{F}_{CA} easier to use as a subroutine than the existing global functionality \mathcal{G}_{bb} for certificate authorities by Canetti et al. [52], as discussed in Remark 4.3.

Corruption Models

As mentioned, our framework provides a very flexible corruption model that, at the same time, is easy to use. One can easily adapt the corruption model to various different situations by specifying one or more of the four corruption related algorithms in an appropriate way. To give just a few examples, one can use the **DetermineCorrStatus** algorithm to consider a higher-level protocol to be corrupted if one/a certain percentage/all of its subroutines are corrupted, modeling situations where security guarantees can still be obtained until too many subroutines are corrupted. A protocol might also consider itself to be corrupted once certain security assumptions are violated, such as an honest majority, because then the protocol can no longer provide any security guarantees (see also the discussion in Section 5.4). The **AllowCorruption** algorithm can be used to define incorruptible machines modeling, e.g., setup assumptions, or it can be used to force the adversary to corrupt all subroutines first, modeling that he can only take full (but not partial) control of a party/computer, which is a less fine grained corruption modeling and hence often easier to analyze. The **AllowAdvMessage** algorithm can be used to restrict access of corrupted entities to other, potentially honest entities. This can be useful to restrict, e.g., access to an uncorrupted signature key stored on a secure hardware token that is not directly accessible by corrupted software running on a PC. In addition to these algorithms, our framework also provides mechanisms to model both static and dynamic corruption which can be freely combined with arbitrary definitions of the above algorithms (see also Section 4.2.2). Since all of these algorithms come with sensible defaults, we do not overburden a protocol

designer. Instead, one is able to omit most or all of these algorithms and still obtain a fully specified and reasonable protocol. Last but not least, our corruption model is defined for and compatible with various different types of protocols such as real, ideal, joint state, and global state protocols as all of these protocols use the same underlying template. In particular, since all of these protocols use the same underlying mechanisms for modeling corruption, it is possible to seamlessly combine and compose all of them.

4.4.4. Capturing the SUC Model

To further illustrate the flexibility and expressiveness of our framework, we show that we can easily capture the SUC model by Canetti et al. [40] as a mere special case in our framework. The SUC model was created as a simpler version of the UC model that is tailored towards the setting of secure multiparty computation. The most important changes are the following:

- (i) A run/session consists of a fixed number of parties, each of them corresponding to one machine instance, instead of an unbounded number.
- (ii) The runtime definition is simplified, i.e., machine instances are only required to run in polynomial time (in their input from the environment) in the classical sense.
- (iii) All communication is over authenticated channels.
- (iv) A corruption model for ideal protocols is included (this was not the case for the UC model in the version from 2013 which SUC is based on).
- (v) Machines do not have any subroutines except for possibly (multiple instances/sessions of) a single ideal functionality that is accessed not directly but rather via an authenticated channel. That is, all program logic is contained in a single machine.

Since some of those changes have to break out of the UC model, in particular the first two, the authors of the SUC model had to create and prove a new composition theorem which takes up a major part of their paper.

It is easy to obtain all of the above properties within our framework by choosing appropriate definitions for all fields in our template:

- (i) The number of parties can be fixed by defining the **CheckID** algorithm such that PIDs are accepted only if they are in the range of $[1, \dots, n]$.
- (ii) The runtime definition in our framework is already simpler than in SUC, as we (intuitively) require only the whole protocol to run in polynomial time instead of individual machines and even allow for a negligible set of non-polynomial time runs.
- (iii) To model authenticated channels, one uses a subroutine $\mathcal{F}_{\text{auth}}$ for ideal authenticated channels that forwards messages while leaking their content.
- (iv) Corruption for all types of protocols, including ideal ones, is already defined in our framework and can be further customized, if desired.
- (v) It is straightforward to encode the whole protocol logic into a single **Main** algorithm, if so desired, while connecting to at most a single (ideal) functionality. This connection can

either be indirect via an authenticated channel just as in the SUC model, or via a simpler direct connection as a standard subroutine.

Importantly, we do not have to re-prove a composition theorem for creating what is just a mere instantiation. Furthermore, since we do not have to break out of our framework, such an instantiation can be combined and/or realized with arbitrary other protocols defined either in our framework or directly within the underlying IITM model, including those protocols that use joint state and global state. This is in contrast to SUC, which supports only disjoint sessions with local state and cannot directly be combined with other UC protocols as they use different computational models (only a mapping from SUC protocols to somewhat artificial UC protocols exists).

4.5. Related Work

As already explained, iUC is an instantiation of the IITM model. In what follows, we therefore relate iUC with the UC and GNUC models. This comparison summarizes key differences; more details on individual features of iUC compared to the other models are given in Section 4.4.

While both the UC and GNUC models also enjoy the benefits of established protocol modeling conventions, those are, however, less flexible and less expressive than iUC. Let us give several concrete examples: conventions in UC and GNUC are built around the assumption of having globally unique SIDs that are shared between all participants of a protocol session, and thus locally managed SIDs cannot directly be expressed (cf. Section 4.4.3 for details including a discussion of local SIDs). Both models also assume protocols to have disjoint sessions and thus their conventions do not support expressing protocols that directly share state between sessions, such as signature keys (while both models support joint state realizations to somewhat remedy this drawback, those realizations have to modify the protocols at hand, which is not always desirable; cf. Section 4.4.3). Furthermore, in both models there is only a single highest-level protocol machine with potentially multiple instances, whereas iUC supports arbitrarily many highest-level protocol machines. This is very useful as it, for example, allows for seamlessly modeling global state without needing any extensions or modifications to our framework or protocol template (as illustrated in Section 4.3). In the case of GNUC, there are also several additional restrictions imposed on protocols, such as a hierarchical tree structure where all subroutines have a single uniquely defined caller (unless they are globally available also to the environment) and a fixed top-down corruption mechanism; none of which is required in iUC.

There are also some major differences between UC/GNUC and iUC on a technical level which further affect overall usability as well as expressiveness. Firstly, both UC and GNUC had to introduce various extensions of the basic computational model to support new types of protocols and composition, including new syntax and new composition theorems for joint state, global state, and realizations of global functionalities [41, 51, 52, 75]. This not only forces protocol designers to learn new protocol syntax and conventions for different types of composition, but also indicates a lack of flexibility in supporting new types of composition (say, for example, a joint state realization that combines multiple separate global functionalities, cf. Section 4.4.2). In contrast, both composition theorems in iUC as well as our single template for protocols seamlessly support and hence enable a unified treatment of all of those types of protocols and composition, including some not considered in the literature so far (cf. Section 4.4.2). Secondly, there are several technical aspects in the UC model a protocol designer has to take care of in order to perform sound proofs: a runtime notion that allows for exhaustion of machines, even ideal functionalities, and that forces protocols to manually send runtime tokens between individual machine instances; a directory machine where protocols have to register all instances when they are created; “subroutine respecting” protocols that keep sessions disjoint. Technical requirements of the GNUC model mainly consist of several restrictions imposed on protocol structures (as mentioned above) which in particular keep protocol sessions disjoint. Unlike UC,

the runtime notion of GNUC supports modeling protocols that cannot be exhausted, however, GNUC introduces additional flow-bounds to limit the number of bits sent between certain machines. In contrast, as also illustrated by the protocols in our case study (cf. Chapter 5), iUC does not require directory machines, iUC’s notion for protocols with disjoint sessions is completely optional and can be avoided entirely, and iUC’s runtime notion allows for modeling protocols without exhaustion, without manual runtime transfers, and without requiring flow bounds (exhaustion and runtime transfers can of course be modeled as special cases, if desired).

The difference in flexibility and expressiveness of iUC compared to UC and GNUC is further illustrated by our case study in Chapter 5, where we model several real world key exchange protocols exactly as they would be deployed in practice. This case study is not directly supported by the UC and GNUC models as discussed in Section 5.4. A second illustrative example was given in Section 4.4.4, where we show that iUC can capture the SUC model [40] as a mere special case. The SUC model was proposed as a simpler version of the UC model specifically designed for secure multi party computation (MPC), but has to break out of (some technical aspects of) the UC model.

5. Case study

In this chapter, we model and then analyze the security of three real world key exchange protocols precisely as deployed in practice. This chapter is based on results that we first published in [89], however, [89] directly used the IITM model with responsive environments as the iUC framework had not been designed yet. In this thesis, we instead use the iUC framework to define and present all protocols, including the key exchange protocols and several ideal functionalities. This not only simplifies presentation and formal definitions due to the modeling conventions available in the iUC framework, therefore illustrating the usability improvements provided by iUC, but also shows that the iUC framework retains the necessary level of flexibility and expressiveness of the IITM model to be able to faithfully capture all protocols from this case study (see also the discussion in Section 5.4 where we explain why other UC-like models do not directly support this case study).

The structure of this chapter is as follows: We first define and realize our ideal functionality for cryptographic $\mathcal{F}_{\text{crypto}}$ in Section 5.1. We then propose our ideal key exchange functionalities $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ and $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ in Section 5.2 which we use for the analysis of three real world key exchange protocols in Section 5.3. Section 5.4 highlights several important aspects where this case study illustrates and makes use of the flexibility of iUC, also compared to other models. Related work is discussed in Section 5.5.

5.1. An Ideal Functionality for Cryptographic Primitives

We now present our ideal functionality $\mathcal{F}_{\text{crypto}}$ for cryptographic primitives. As already explained in the introduction (cf. Chapter 1), a higher-level protocol \mathcal{P} can use $\mathcal{F}_{\text{crypto}}$ for its cryptographic operations. Then, in order to show that $(\mathcal{P} \mid \mathcal{F}_{\text{crypto}}) \leq \mathcal{F}$, i.e., that \mathcal{P} (using $\mathcal{F}_{\text{crypto}}$ for its cryptographic operations) realizes some ideal functionality \mathcal{F} (e.g., a key exchange functionality), one can argue on a purely information-theoretic level, without resorting to reductions or hybrid arguments (at least for those primitives supported by $\mathcal{F}_{\text{crypto}}$). For example, $\mathcal{F}_{\text{crypto}}$ guarantees that only the (honest) owner of a Diffie-Hellman key can get access to keys that are derived from it, and only parties with access to these keys can, e.g., create a MAC with such keys. In all other cryptographic approaches for security protocols, one has to reduce these properties to the security assumptions for Diffie-Hellman key exchange, key derivation, and MAC schemes. Once $(\mathcal{P} \mid \mathcal{F}_{\text{crypto}}) \leq \mathcal{F}$ has been proven, using the composition theorems of the iUC framework one can replace $\mathcal{F}_{\text{crypto}}$ with its realization $\mathcal{P}_{\text{crypto}}$ (see Section 5.1.4) by which the ideal operations provided by $\mathcal{F}_{\text{crypto}}$ are replaced by the real counterparts.

As mentioned in the introduction, in [97] a first version of $\mathcal{F}_{\text{crypto}}$ was proposed (with an updated presentation given in [112]), which, however, does not support DH key exchange, a fundamental primitive for most real world key exchange protocols. We not only extend $\mathcal{F}_{\text{crypto}}$ to support this primitive but we also improve $\mathcal{F}_{\text{crypto}}$ in various other ways in order to overcome shortcomings of the previous version, as discussed in Section 5.1.2. Our extension of $\mathcal{F}_{\text{crypto}}$, in particular the treatment of DH key exchange, is non-trivial and needs care in order for it to be widely usable and realizable. In the following, we first recall $\mathcal{F}_{\text{crypto}}$ as defined in [97]. While this first version was defined in the original IITM model (without responsive environments), here we present an equivalent definition within the iUC framework and using iUC terminology. We do so for better readability as our extension is also based on the iUC framework. We then define our extension of $\mathcal{F}_{\text{crypto}}$ and propose a realization $\mathcal{P}_{\text{crypto}}$ of the extension. We finally show that $\mathcal{P}_{\text{crypto}}$ realizes $\mathcal{F}_{\text{crypto}}$.

5.1.1. The ideal functionality $\mathcal{F}_{\text{crypto}}$ from [97]

On a high level, the ideal functionality $\mathcal{F}_{\text{crypto}}$ from [97] allows its users to perform the following operations in an ideal way: i) generate symmetric keys, including pre-shared keys, ii) generate public/private keys and share public keys with each other, iii) derive symmetric keys from other symmetric keys, iv) encrypt and decrypt messages and ciphertexts, respectively (public-key encryption and both unauthenticated and authenticated symmetric encryption are supported), v) compute and verify MACs and digital signatures, and vi) generate fresh nonces. All symmetric and public keys can be part of plaintexts to be encrypted under other symmetric and public keys, which allows users from higher-level protocols to send and receive encrypted keys. Derived keys can be used in the same ways as freshly generated symmetric keys. In what follows, we describe $\mathcal{F}_{\text{crypto}}$ in more detail. For reference purposes we also provide a precise specification using our template in Appendix F (this specification includes our extension from the next section). We note that the following description is sufficient to understand the rest of this thesis even without reading the precise specification in the appendix.

Formally, the ideal functionality $\mathcal{F}_{\text{crypto}}$ consists of a single public role `crypto` implemented by one machine. An instance of this role manages all entities, i.e., in a run of $\mathcal{F}_{\text{crypto}}$ there is at most a single instance of `crypto` that handles all cryptographic operations.³⁹ The ideal functionality $\mathcal{F}_{\text{crypto}}$ is parameterized with a polynomial p and a leakage algorithm \mathcal{L} . The polynomial p is used to bound the runtime of algorithms provided by the adversary. The leakage algorithm \mathcal{L} takes as input the security parameter 1^n and a plaintext $x \in \{0, 1\}^*$ and outputs the information

³⁹A single instance/copy of $\mathcal{F}_{\text{crypto}}$ already provides all necessary cryptographic operations for higher-level protocols and hence is generally sufficient for their analysis. In particular, *all* entities of higher-level protocols are supposed to access the same instance of $\mathcal{F}_{\text{crypto}}$ such that cryptographic primitives can be freely combined with each other. We note however that, if so desired, it is trivial to define a version of $\mathcal{F}_{\text{crypto}}$ that provides multiple independent instances. Such a multi-instance version would accept all entities sharing the same SID, i.e., the SID would allow for addressing different instances. Each instance would then run independently of each other in exactly the same way as defined for the single-instance version of $\mathcal{F}_{\text{crypto}}$. All results from this work directly carry over to such a multi-instance version of $\mathcal{F}_{\text{crypto}}$.

that is leaked when a plaintext x is encrypted ideally. Typical examples are (i) $\mathcal{L}(1^\eta, x) = 0^{|x|}$ and (ii) the algorithm that outputs a random bit string of length $|x|$, both of which leak exactly the length of the message. We call a leakage algorithm *length preserving* if it always holds true that $|\mathcal{L}(1^\eta, x)| = |x|$ for all η and x . We say a leakage algorithm *leaks exactly the length (of a message)* if it is length preserving and there exists a probabilistic polynomial time algorithm T such that the probability distribution of $T(1^\eta, 1^{|x|})$ equals the probability distribution of $\mathcal{L}(1^\eta, x)$. Note that both of the previous examples leak exactly the length of a message. During the initialization of $\mathcal{F}_{\text{crypto}}$, the adversary is supposed to provide (stateless) algorithms for authenticated and unauthenticated symmetric encryption ($\text{enc}_{\text{auth}}, \text{dec}_{\text{auth}}, \text{enc}_{\text{unauth}}, \text{dec}_{\text{unauth}}$), MACing ($\text{mac}, \text{verify}_{\text{mac}}$), public key encryption ($\text{enc}_{\text{pke}}, \text{dec}_{\text{pke}}$), and signing ($\text{sig}, \text{verify}_{\text{sig}}$).

All symmetric keys in $\mathcal{F}_{\text{crypto}}$ are equipped with a key type that determines their usage. That is, a key k is of the form (t, k') where k' is the actual bit string used in algorithms while t is the key type. Keys of type **pre-key** are used to derive other keys, keys of type **unauthenc-key** and **authenc-key** are used for (un)authenticated encryption and decryption, and keys of type **mac-key** are used to create and verify MACs. This models the practice of using keys for a single purpose only and is necessary to realize $\mathcal{F}_{\text{crypto}}$ based on standard cryptographic assumptions.

Keys in $\mathcal{F}_{\text{crypto}}$ can be corrupted by the adversary. More precisely, the adversary can statically corrupt all asymmetric (signing/encryption) keys, i.e., he can corrupt them before they are used for the first time but not afterwards. The corruption status of asymmetric keys determines whether operations with these keys are performed ideally or without ideal security guarantees.⁴⁰ Similarly, the adversary can statically corrupt symmetric keys when they are generated or, in the case of pre-shared keys, when they are retrieved for the first time. For symmetric keys, $\mathcal{F}_{\text{crypto}}$ additionally tracks whether a key might have become *known* to the environment within the context of a realization of $\mathcal{F}_{\text{crypto}}$. This can occur, e.g., when the key itself was already corrupted during generation, or if the key has been encrypted using a key that was already known to the environment. For this purpose, $\mathcal{F}_{\text{crypto}}$ maintains a set keys of all symmetric keys and a set $\text{keys}_{\text{known}} \subseteq \text{keys}$ which contains all keys that might be *known* to the environment. The known/unknown status of symmetric keys is then used to determine whether symmetric operations are performed ideally or without ideal security guarantees. In the following, we will call a key *known* if it is in $\text{keys}_{\text{known}}$ and *unknown* if it is in the set $\text{keys}_{\text{unknown}} := \text{keys} \setminus \text{keys}_{\text{known}}$.

New symmetric keys are generated by requesting the actual bit string of the key from the adversary. The adversary is free to provide (almost) arbitrary responses to these requests; the functionality $\mathcal{F}_{\text{crypto}}$ ensures only that a new unknown symmetric key k is fresh (i.e., $k \notin$

⁴⁰Note that this differs from the default corruption model in iUC since the adversary *does not gain full control over messages sent to/received from $\mathcal{F}_{\text{crypto}}$* upon corruption. Instead, corruption disables any form of security guarantees. For example, digital signatures can trivially be forged for corrupted signing keys. This models that an attacker might gain access to a secret key even without fully controlling (part of) the computer of a user, i.e., without seeing which messages the user, say, signs with that key. If one desires to model a situation where the attacker gains full control upon corruption of a signature key, then this can easily be modeled on top of $\mathcal{F}_{\text{crypto}}$ by adding a dummy forwarder that gives the attacker full control over signing/verification requests once a signature key has been corrupted. By the composition theorem (cf. Corollary 4.1), all of our results directly carry over to this setting.

keys) and prevents key guessing of unknown keys when receiving a new known key k (i.e., $k \notin \text{keys}_{\text{unknown}}$). Note that, as the adversary provides the keys, he knows the actual value of symmetric keys that are marked as unknown in $\mathcal{F}_{\text{crypto}}$. This is not a contradiction: the known/unknown status keeps track whether a the actual value of a key is known/unknown *within the realization of $\mathcal{F}_{\text{crypto}}$* . $\mathcal{F}_{\text{crypto}}$ can provide ideal security guarantees only for those keys that have not been leaked by the realization as the environment could otherwise trivially distinguish the real and ideal worlds. Similar to symmetric keys, the adversary provides the asymmetric key pairs for all parties; he can do so at any point in time as long as a key pair has not been defined yet.

A user $\text{entity}_{\text{call}} = (\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}})$ from a higher-level protocol can access an entity of $\mathcal{F}_{\text{crypto}}$ in order to perform several cryptographic operations (see below).⁴¹ For every user, $\mathcal{F}_{\text{crypto}}$ stores a list of his symmetric keys. Additionally, $\mathcal{F}_{\text{crypto}}$ provides one signing and one encryption key pair for every PID pid_{call} . Importantly, while users gain access to the public keys, $\mathcal{F}_{\text{crypto}}$ does not provide the user with the actual bit strings of (uncorrupted) private/symmetric keys. Instead, users obtain pointers to their keys, which they can then use to indicate that a certain key should be used for some cryptographic operation. This system of pointers is necessary for the same reason as the known/unknown system: One cannot realize an ideal functionality that provides security guarantees for keys where the actual secret bit string is output to the user and hence the environment. More precisely, for asymmetric public key operations, the PID of the key owner uniquely determines the key that is to be used. For symmetric key operations, the user provides a pointer $\text{ptr} \in \mathbb{N}$ pointing to a specific symmetric key that is to be used. $\mathcal{F}_{\text{crypto}}$ then uses this pointer to retrieve the key from a mapping keymap , which maps users and pointers to their symmetric keys; that is, $\text{keymap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, \text{ptr})$ is the key of user $(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}})$ corresponding to the pointer ptr .

Pointers to symmetric keys may also be part of the messages given to $\mathcal{F}_{\text{crypto}}$ for encryption. Before a message is actually encrypted, the pointers are replaced by the keys they refer to. Upon decryption of a ciphertext, keys embedded in the plaintext are first turned into (new) pointers before the plaintext is given to the user. In order to be able to identify pointers/keys, we assume pointers/keys in plaintexts to be tagged according to their types. To describe tagging more formally and flexible, we introduce two extra types: **tuple** to structure messages and **data** for everything else, i.e., arbitrary bit strings.

To tag messages, any tagging function Tag can be used that maps a type t and bit strings x_1, \dots, x_n (where n might depend on t) to a tagged bit string $\text{Tag}_t(x_1 \dots, x_n)$ with the following properties: Tag is injective, computable and invertible in polynomial-time in the length of the input, and it is length regular, i.e., $|\text{Tag}_t(x_1 \dots, x_n)| = |\text{Tag}_t(x'_1, \dots, x'_n)|$ for every type t and bit strings $x_1, x'_1, \dots, x_n, x'_n$ where $|x_i| = |x'_i|$ for all $i \leq n$. Pointers are tagged with the type of the key they refer to. We say that a bit string x is *well-tagged* if $x = \text{Tag}_t(y)$ for some type $t \neq \text{tuple}$ and a bit string y or (recursively defined) $x = \text{Tag}_{\text{tuple}}(x_1, \dots, x_n)$ for some $n \geq 1$ and

⁴¹We note that the current entity of $\mathcal{F}_{\text{crypto}}$ is not used by the logic of $\mathcal{F}_{\text{crypto}}$. Instead, all logic depends on the calling entity $\text{entity}_{\text{call}}$.

well-tagged bit strings x_1, \dots, x_n . We say that a bit string x has type t if it is well-tagged and $x = \text{Tag}_t(x_1, \dots, x_n)$ for some $n \geq 1$ and x_1, \dots, x_n . We will only require that plaintexts to be encrypted are well-tagged; MACs, digital signature, decryption, and key derivation operations take arbitrary bit strings as input. Note that this tagging policy is very liberal. It is necessary to distinguish keys and tuples from other data in order to be able to parse plaintexts. Everything which is not a key or a tuple is considered to be of type **data**, which can be freely interpreted by the higher-level protocols using $\mathcal{F}_{\text{crypto}}$.

A user entity $\text{entity}_{\text{call}} = (\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}})$ from a higher-level protocol can send a message m to some arbitrary entity $(\text{pid}, \text{sid}, \text{crypto})$ of $\mathcal{F}_{\text{crypto}}$ to perform one of the following operations (the format of the message m for each respective operation, including parameters contained therein, is given in parentheses):

- **Generating fresh, symmetric keys** $[(\text{New}, t)]$. A user can generate a new symmetric key of type $t \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}$. The request is forwarded to the adversary who is supposed to provide such a key, say k . The adversary can decide to corrupt k right away, in which case (t, k) is known and therefore added to both $\text{keys}_{\text{known}}$ and keys , and otherwise (t, k) is unknown and therefore added only to keys . However, before adding (t, k) to a set, $\mathcal{F}_{\text{crypto}}$ ensures that k is fresh and key guessing is prevented, i.e., in case k is uncorrupted, it may not belong to keys , and in case k is corrupted, it may not belong to $\text{keys}_{\text{unknown}}$. If $\mathcal{F}_{\text{crypto}}$ accepts k , a new pointer ptr to (t, k) is created for user $\text{entity}_{\text{call}}$, i.e., $\text{keymap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, \text{ptr}) \leftarrow (t, k)$, and (New, ptr) is returned to $\text{entity}_{\text{call}}$. The value of the pointer, i.e., ptr , does not need to be secret. In fact, new pointers are created by increasing a counter starting from 0. There is a different counter for every user, i.e., a user cannot tell how many keys have been created by other users from observing his pointers. If the adversary decided to corrupt k , then the pointer ptr is recorded as corrupted for user $\text{entity}_{\text{call}}$.
- **Establishing pre-shared keys** $[(\text{GetPSK}, t, \text{name})]$. A user can ask for a pointer to a pre-shared symmetric key (PSK) of type $t \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}$ with identifier name . Then, $\mathcal{F}_{\text{crypto}}$ forwards this request to the adversary who is supposed to provide such a key, say k . Similarly to generating fresh keys, the adversary can decide to corrupt k . However, $\mathcal{F}_{\text{crypto}}$ only accepts k under the following conditions: (i) if a key (t, k') is already recorded for (t, name) , then $k = k'$ and k is not corrupted (i.e., the adversary did not decide to corrupt k), (ii) if (t, name) is (recorded as) corrupted, then k is corrupted, (iii) if k is corrupted, then $(t, k) \notin \text{keys}_{\text{unknown}}$, and (iv) if no key (t, k') is recorded for (t, name) and k is not corrupted, then $(t, k) \notin \text{keys}$. If $\mathcal{F}_{\text{crypto}}$ accepts k , then it creates a new pointer ptr to (t, k) for user $\text{entity}_{\text{call}}$ and returns $(\text{GetPSK}, \text{ptr})$ to the user. Furthermore, if k is corrupted, then $\mathcal{F}_{\text{crypto}}$ adds (t, k) to $\text{keys}_{\text{known}}$ (and keys), records (t, name) as corrupted PSK, and records ptr as corrupted for user $\text{entity}_{\text{call}}$. If k is not corrupted, then $\mathcal{F}_{\text{crypto}}$ adds (t, k) to keys and records (t, k) for (t, name) .

This allows users to establish shared keys: For example, two users $(pid_1, sid_1, role_1)$ and $(pid_2, sid_2, role_2)$ can obtain pointers to a fresh key k shared between pid_1 and pid_2 by each sending the request $(\text{GetPSK}, t, (pid_1, pid_2))$ to $\mathcal{F}_{\text{crypto}}$. While, by such a request, pid_1 (pid_2) gets a new pointer in every local session and role, if the PSK $(t, (pid_1, pid_2))$ is uncorrupted this pointer will always point to the same key k because of condition (i). For corrupted pre-shared keys, i.e., where $(t, name)$ is corrupted, the adversary can choose a different key for every user; conditions (ii) and (iii) guarantee that these keys are corrupted (more precisely, the pointers to corrupted pre-shared keys are corrupted) and do not belong to $\text{keys}_{\text{unknown}}$. Condition (iv) guarantees that new pre-shared keys are fresh, i.e., do not collide with any other key.

- **Store** $[(\text{Store}, t, k)]$. A user can manually store a bit string k as a (known) symmetric key of type $t \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}$ in $\mathcal{F}_{\text{crypto}}$. If (t, k) belongs to $\text{keys}_{\text{unknown}}$, $\mathcal{F}_{\text{crypto}}$ will return an error message to the user, modeling that unknown keys cannot be guessed. Otherwise, $\mathcal{F}_{\text{crypto}}$ adds (t, k) is added to $\text{keys}_{\text{known}}$ (and keys) and then creates a new pointer to (t, k) which is returned to the user.
- **Retrieve** $[(\text{Retrieve}, ptr)]$. A user can retrieve the key k a pointer ptr refers to. $\mathcal{F}_{\text{crypto}}$ marks the key $(t, k) = \text{keymap}(pid_{\text{call}}, sid_{\text{call}}, role_{\text{call}}, ptr)$ as known by adding it to $\text{keys}_{\text{known}}$ and then returns $(\text{Retrieve}, k)$ to the user.
- **Equality test** $[(\text{Equal?}, ptr, ptr')]$. A user can test whether two of his pointers ptr and ptr' refer to the same key (same type and same bit string).
- **Public key requests** $[(\text{GetPubKeyPKE}, pid')$ or $(\text{GetPubKeySig}, pid')]$. A user can ask for the public encryption/verification key, if any, of some party pid' . If $\mathcal{F}_{\text{crypto}}$ has recorded this public key (because the adversary previously provided it, see above), then it is returned to the user. Otherwise, an error is returned to the user. We note that if users request the public key of another party, then this assumes that public keys are distributed somehow, e.g., by some kind of public key infrastructure.
- **Key derivation** $[(\text{Derive}, ptr, t', s)]$. A user can derive a new symmetric key of type $t' \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}$ from a seed s and from the key $(t, k) = \text{keymap}(pid_{\text{call}}, sid_{\text{call}}, role_{\text{call}}, ptr)$ pointer ptr points to. The seed s is an arbitrary bit string. It is required that the key k from which the new key is derived is of type **pre-key**, i.e., $t = \text{pre-key}$. Then, $\mathcal{F}_{\text{crypto}}$ forwards this request (including the key (t, k) and its known/unknown status) to the adversary who is supposed to provide such a key, say k' . However, $\mathcal{F}_{\text{crypto}}$ only accepts k' under the following conditions: (i) if a key (t', k'') is recorded as being derived from (t, k) with seed s , then $k'' = k'$, (ii) if no key (t', k'') has been recorded as derived from (t, k) with seed s and $(t, k) \in \text{keys}_{\text{unknown}}$, then $(t', k') \notin \text{keys}$, and (iii) if no key (t', k'') has been recorded as derived from (t, k) with seed s and $(t, k) \in \text{keys}_{\text{known}}$, then $(t', k') \notin \text{keys}_{\text{unknown}}$. If $\mathcal{F}_{\text{crypto}}$ accepts k' and $(t, k) \in \text{keys}_{\text{unknown}}$, then $\mathcal{F}_{\text{crypto}}$ adds (t', k') to keys and records (t', k')

as derived from (t, k) with seed s . If $(t, k) \in \text{keys}_{\text{known}}$, then $\mathcal{F}_{\text{crypto}}$ adds (t', k') to $\text{keys}_{\text{known}}$ (and keys). Finally, $\mathcal{F}_{\text{crypto}}$ creates a new pointer ptr to (t', k') for user $\text{entity}_{\text{call}}$ and returns (Derive, ptr) to the user.

Condition (i) guarantees that key derivation is deterministic, i.e., key derivations from the same key with the same seed yield the same key. Similar to generating fresh keys, condition (ii) guarantees that new keys derived from unknown keys do not collide with any other key and condition (iii) guarantees that new keys derived from known keys at least do not collide with unknown keys. Note that we do not put any restrictions on how the adversary chooses derived keys. All security guarantees that $\mathcal{F}_{\text{crypto}}$ provides do not rely on this. In $\mathcal{F}_{\text{crypto}}$, a key derived from a key marked unknown is treated just like a freshly generated key which is marked unknown. For example, if the derived key is an encryption key, then it is used for ideal encryption, i.e., not the actual message is encrypted but its leakage, see below.

- **Encryption under symmetric keys** $[(\text{Enc}, ptr, x)]$. A user can encrypt a well-tagged message x under a key $(t, k) = \text{keymap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, ptr)$ pointer ptr points to where $t \in \{\text{authenc-key}, \text{unauthenc-key}\}$. For every pointer $\text{Tag}_{t'}(ptr')$ that is contained in x (for some $t' \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}$), $\mathcal{F}_{\text{crypto}}$ checks whether ptr' is a pointer of this user to a key of type t' , i.e., whether $\text{keymap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, ptr')$ is defined and yields (t', k') for some k' . If this check fails, $\mathcal{F}_{\text{crypto}}$ returns an error message to the user. If the check succeeds, $\text{Tag}_{t'}(ptr')$ in x is replaced by $\text{Tag}_{t'}(k')$. This is done for every pointer of the form $\text{Tag}_{t'}(ptr')$ in x , resulting in a message x' . We distinguish two cases:

1. If $(t, k) \in \text{keys}_{\text{unknown}}$, the leakage $\bar{x} = \mathcal{L}(1^\eta, x')$ of x' is encrypted under k using either enc_{auth} or $\text{enc}_{\text{unauth}}$ (the encryption algorithms provided by the adversary) depending on t . Let y denote the resulting ciphertext. Then, $\mathcal{F}_{\text{crypto}}$ checks if the decryption of y under k using either dec_{auth} or $\text{dec}_{\text{unauth}}$ (the decryption algorithms provided by the adversary), depending on t , yields the leakage \bar{x} . If this check fails, $\mathcal{F}_{\text{crypto}}$ returns an error message to pid . Otherwise, the pair (x', y) is stored for the key (t, k) (for later decryption) and (Enc, y) is returned to the user.
2. If $(t, k) \in \text{keys}_{\text{known}}$, all keys in x' are added to $\text{keys}_{\text{known}}$ as they are encrypted under a known key. Then, x' is encrypted under k using either enc_{auth} or $\text{enc}_{\text{unauth}}$ depending on t . The resulting ciphertext is given to the user.

- **Decryption under symmetric keys** $[(\text{Dec}, ptr, y)]$. A user can decrypt a ciphertext y (an arbitrary bit string) under a key $(t, k) = \text{keymap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, ptr)$ pointer ptr points to where $t \in \{\text{authenc-key}, \text{unauthenc-key}\}$. We distinguish two cases:

1. If $(t, k) \in \text{keys}_{\text{unknown}}$, $\mathcal{F}_{\text{crypto}}$ checks whether there exists exactly one x' such that (x', y) is stored for (t, k) (see above). If so, $\mathcal{F}_{\text{crypto}}$ creates new pointers to every key in x' and replaces the keys by the corresponding pointers. The resulting message x is given to the user. If there is more than one x' with (x', y) stored for (t, k) , an error is returned

to the user since unique decryption is not possible. If there is no such x' , the following is done: If $t = \text{authenc-key}$, an error is returned, since for authenticated encryption it should not be possible to generate valid ciphertexts outside of the functionality. If $t = \text{unauthenc-key}$, y is decrypted under k using $\text{dec}_{\text{unauth}}$ (the decryption algorithm provided by the adversary) and the following is done (*): If the resulting plaintext x' is not well-tagged, an error is returned. Furthermore, if x' contains a key that belongs to $\text{keys}_{\text{unknown}}$, an error is returned, modeling that these keys cannot be guessed. Otherwise, $\mathcal{F}_{\text{crypto}}$ adds all keys in x' to $\text{keys}_{\text{known}}$ (and keys), creates new pointers to every key in x' , and replaces the keys by the corresponding pointers. The resulting message x is given to the user.

2. If $(t, k) \in \text{keys}_{\text{known}}$, $\mathcal{F}_{\text{crypto}}$ decrypts y under k with dec_{auth} or $\text{dec}_{\text{unauth}}$, depending on t , and proceeds as in (*) above.

- **Encryption under public keys** $[(\text{PKEnc}, \text{pid}', pk, x)]$. A user can encrypt a well-tagged message x under the public key pk of party pid' . Such an encryption request is handled similarly as symmetric encryption requests. First, pointers in x are turned into keys, obtaining x' . Then, if pk is the recorded public encryption key of party pid' and it is not corrupted, the encryption is performed ideally, i.e., the leakage $\mathcal{L}(1^n, x')$ of x' is encrypted under pk using enc_{pke} (the public-key encryption algorithm provided by the adversary) and the pair (x', y) (where y is the ciphertext) is recorded for party pid' (for later decryption). If pk is not the recorded public encryption key of party pid' or the public encryption key of party pid' is corrupted, then all keys in x' are marked known and x' is encrypted under pk using enc_{pke} . The resulting ciphertext is returned to the user.

- **Decryption under private keys** $[(\text{PKDec}, y)]$. A user can decrypt a ciphertext y (an arbitrary bit string) under its private key (i.e., the private key of party pid_{call}). Such a decryption request is handled similarly as decryption requests under symmetric keys of type **unauthenc-key**. If the private decryption key of pid_{call} is corrupted or there is no recorded pair (x', y) for party pid_{call} (for any x'), then y is decrypted under the recorded private decryption key of party pid_{call} using the public-key decryption algorithm dec_{pke} (provided by the adversary). $\mathcal{F}_{\text{crypto}}$ returns an error message to the user if the resulting plaintext, say x' , is not well-tagged or there exists a key in x' that belongs to $\text{keys}_{\text{unknown}}$, modeling that these keys cannot be guessed. All keys in x' are added to $\text{keys}_{\text{known}}$ (and keys).

Otherwise, i.e., if the private decryption key of pid is not corrupted and there exists an x' such that (x', y) is recorded for party pid_{call} (see above), $\mathcal{F}_{\text{crypto}}$ checks whether there exists exactly one such x' . If this check fails, an error is returned to the user since unique decryption is not possible.

Finally, $\mathcal{F}_{\text{crypto}}$ creates new pointers to every key in x' and replaces the keys by the corresponding pointers. The resulting plaintext is given to the user.

- **Creating MACs** $[(\text{MAC}, ptr, x)]$. A user can MAC an arbitrary (uninterpreted) bit string x under a key $(t, k) = \text{keymap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, ptr)$ pointer ptr points to where $t = \text{mac-key}$. Then, $\mathcal{F}_{\text{crypto}}$ computes the MAC of x under k using mac (the MAC algorithm provided by the adversary). Let σ be the resulting MAC. If σ is not a valid MAC for x under k , i.e., it does not verify using $\text{verify}_{\text{mac}}$ (the MAC verification algorithm provided by the adversary), then $\mathcal{F}_{\text{crypto}}$ returns an error message to the user. Otherwise, it returns (MAC, σ) to the user. If $(t, k) \in \text{keys}_{\text{unknown}}$, $\mathcal{F}_{\text{crypto}}$ records x for the key (t, k) (for later verification); we allow an adversary to derive a new MAC from a given one on the same message, which is why $\mathcal{F}_{\text{crypto}}$ does not record σ along with x .
- **Verifying MACs** $[(\text{MACVerify}, ptr, x, \sigma)]$. A user can verify a MAC σ for some message x under a key $(t, k) = \text{keymap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, ptr)$ pointer ptr points to where $t = \text{mac-key}$. Then, $\mathcal{F}_{\text{crypto}}$ verifies the MAC using $\text{verify}_{\text{mac}}$ (the MAC verification algorithm provided by the adversary). If the MAC verifies but $(t, k) \in \text{keys}_{\text{unknown}}$ and x has not been recorded for (t, k) (see above), $\mathcal{F}_{\text{crypto}}$ returns an error message to the user, preventing forgery. Otherwise, $\mathcal{F}_{\text{crypto}}$ returns the result of the verification to the user.
- **Creating signatures** $[(\text{Sign}, x)]$. Similar to MACing, a user can sign an arbitrary (uninterpreted) bit string x under his private signing key. Then, $\mathcal{F}_{\text{crypto}}$ computes the signature of x under the recorded private signing key of party pid_{call} using sig (the signing algorithm provided by the adversary). Let σ be the resulting signature. If σ is not a valid signature for x under the recorded public verification key of party pid , i.e., it does not verify using $\text{verify}_{\text{sig}}$ (the signature verification algorithm provided by the adversary), then $\mathcal{F}_{\text{crypto}}$ returns an error message to the user. Otherwise, it returns (Sign, σ) to the user. If the private signing key of party pid is not corrupted, then $\mathcal{F}_{\text{crypto}}$ records x for pid (for later verification); we allow an adversary to derive a new signature from a given one on the same message, which is why $\mathcal{F}_{\text{crypto}}$ does not record σ along with x .
- **Verifying signatures** $[(\text{SigVerify}, \text{pid}', pk, x, \sigma)]$. A user can verify a signature σ for some message x under the public key pk for party pid' . Then, $\mathcal{F}_{\text{crypto}}$ verifies the signature using pk and $\text{verify}_{\text{sig}}$ (the signature verification algorithm provided by the adversary). If the signature verifies but pk is the recorded public verification key of party pid' , pk is uncorrupted, and x has not been recorded for pid' (see above), $\mathcal{F}_{\text{crypto}}$ returns an error message to the user, preventing forgery. Otherwise, $\mathcal{F}_{\text{crypto}}$ returns the result of the verification to the user.
- **Generating fresh nonces** $[(\text{NewNonce})]$. A user can generate a fresh nonce. The request is forwarded to the adversary who is supposed to provide a nonce, say x . However, $\mathcal{F}_{\text{crypto}}$ only accepts x if it is not already recorded as a nonce (for anybody), modeling that x is fresh. Then, $\mathcal{F}_{\text{crypto}}$ records x as a nonce and sends $(\text{NewNonce}, x)$ to the user.
- **Corruption status request.** $[(\text{IsSymKeyCorrupt?}, ptr), (\text{IsPKEncKeyCorrupt?}, \text{pid}'), \text{and } (\text{IsSigKeyCorrupt?}, \text{pid}')]$. A user can ask whether one of his pointers ptr is recorded as cor-

rupted. Similarly, a user can ask whether the public/private key pair for encryption/signatures of a party pid' is recorded as corrupted. This mechanism is used for modeling purposes; in particular, it allows the environment to make sure that the corruption status of a key is the same in the real and ideal worlds.

5.1.2. Extending $\mathcal{F}_{\text{crypto}}$ to Support Diffie-Hellman Key Exchange

We now present our extension to $\mathcal{F}_{\text{crypto}}$ that supports Diffie-Hellman key exchanges based on the standard Decisional Diffie-Hellman assumption (DDH assumption, cf. Appendix E.5). On a high level, the extension lets users generate secret Diffie-Hellman exponents (e) and the corresponding public key shares (g^e), called *DH shares* in what follows. Exponents can be combined with arbitrary DH shares, not necessarily generated by $\mathcal{F}_{\text{crypto}}$, to produce a new symmetric key. If an exponent is combined with a DH share created by $\mathcal{F}_{\text{crypto}}$, then the resulting key will only be accessible by the owners of the two exponents that were used to create the key. The resulting key can then be used to derive other keys, e.g., for encryption or MACing. Whether or not this key derivation is performed ideally depends on several factors, such as whether any of the exponents is known to the environment/adversary (see below). Furthermore, $\mathcal{F}_{\text{crypto}}$ guarantees that new exponents/DH shares are fresh, i.e., no other user has access to the same exponent and no keys were already created from the share. In the following, we describe our extension in detail; for reference purposes, we also provide a formal specification using the iUC protocol template in Appendix F.

Formally, we parameterize $\mathcal{F}_{\text{crypto}}$ with a $\text{GroupGen}(1^\eta)$ algorithm that is used to generate the Diffie-Hellman group. This algorithm takes as input the current security parameter η , runs in polynomial time in η (except with negligible probability), and outputs a description (G, n, g) of a cyclic group G where $|G| = n$ and g is a generator of G . We require that it is possible in polynomial time (in the length of the input and η) to check whether a bit string encodes a group member of such a group, and that the group operation is efficiently computable. We also require that group elements of a given group are encoded as bit strings such that all of those bit strings share the same length (the length might depend on η and G but is independent of and identical for specific group elements). Note that this is easy to achieve, e.g., by adding a suitable prefix.⁴²

Diffie-Hellman exponents are modeled analogously to symmetric keys in $\mathcal{F}_{\text{crypto}}$. That is, a user $\text{entity}_{\text{call}}$ gets pointers to his exponents, never the actual exponent, and can use these pointers to perform Diffie-Hellman key exchanges. For this purpose, $\mathcal{F}_{\text{crypto}}$ maintains a mapping expmap from users and pointers to their exponents, where $\text{expmap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, \text{ptr})$ is the key of user $(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}})$ corresponding to the pointer ptr . However, unlike ex-

⁴²This technical requirement is needed since we want to be able to encrypt DH keys as part of plaintexts, just as all other symmetric keys. However, standard encryption schemes do not hide the length of the input. Hence, we have to ensure that we do not accidentally leak information about DH keys by representing different keys using bit strings of different lengths. If one does not allow for encrypting DH keys, then this requirement can be dropped and arbitrary encodings can be used.

ponents, users *do* get the DH share g^e belonging to an exponent e . $\mathcal{F}_{\text{crypto}}$ maintains a set exp of all exponents and a set $\text{exp}_{\text{known}} \subseteq \text{exp}$ of *known* exponents (with an exponent in $\text{exp}_{\text{unknown}} := \text{exp} \setminus \text{exp}_{\text{known}}$ being called *unknown*). The known/unknown status of exponents keeps track of whether an exponent would have been leaked to the environment in the realization $\mathcal{P}_{\text{crypto}}$ and is therefore used to determine whether DH keys created from those exponents are considered known/unknown. Just as for symmetric keys, the environment provides the actual values of exponents in $\mathcal{F}_{\text{crypto}}$, even if they are considered unknown. $\mathcal{F}_{\text{crypto}}$ prevents exponent collisions (i.e., if a new unknown exponent e is created, then $e \notin \text{exp}$) and exponent guessing (i.e., if a new known exponent e is created, then $e \notin \text{exp}_{\text{unknown}}$). Additionally, to guarantee freshness of group elements computed from new (unknown) exponents, $\mathcal{F}_{\text{crypto}}$ also maintains a set blockedDHShares of blocked DH shares that contains group elements h that may not be generated when a new (unknown) exponent e is created, i.e., $g^e \neq h$. In particular, this set contains all DH shares that have been used to create a Diffie-Hellman key (see Section 5.1.3 for a more detailed explanation of the blockedDHShares mechanism).

We add another symmetric key type **dh-key** to $\mathcal{F}_{\text{crypto}}$ which represents Diffie-Hellman keys. Keys of this type may only be generated via a new **GenDHKey** command (see below) or be inserted into $\mathcal{F}_{\text{crypto}}$ via the existing **Store** command; they may not be created by any other commands. Keys of type **dh-key** can be used to derive new symmetric keys of arbitrary existing types, but they may not be used for encryption or creating MACs directly. Furthermore, just as all other key types, they can be encrypted as part of a plaintext.

During the initialization of $\mathcal{F}_{\text{crypto}}$, $\mathcal{F}_{\text{crypto}}$ now first executes **GroupGen**(1^n) and stores the generated group (G, n, g) . Then, $\mathcal{F}_{\text{crypto}}$ sends both the group (G, n, g) and a request for cryptographic algorithms to the adversary via a restricting message. By using a restricting message, we are guaranteed to receive an immediate response containing the expected algorithms; this is unlike in the original version of $\mathcal{F}_{\text{crypto}}$ which did not have access to responsive environments and hence had to deal with the special case of undefined algorithms in its specification. When this initialization is complete, $\mathcal{F}_{\text{crypto}}$ either continues to process the original message that activated it (if the first message was received from a user in a higher-level protocol) or returns control to the adversary (if the first message was received from the network).

In our extension, we add the following operations to $\mathcal{F}_{\text{crypto}}$ which can be called by a user $(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}})$ of a higher-level protocol:

- **Get generated group** [**GetDHGroup**]. The user can request the group (G, n, g) that was generated by $\mathcal{F}_{\text{crypto}}$ during initialization. $\mathcal{F}_{\text{crypto}}$ responds by sending $(\text{GetDHGroup}, (G, n, g))$ to the user.
- **Generate a fresh exponent** [**GenExp**]. The user can request a pointer to a new unknown exponent e . This request is forwarded to the adversary via a restricting message, who is supposed to provide an exponent $e \in \{1, \dots, n\}$. The functionality $\mathcal{F}_{\text{crypto}}$ then ensures that this exponent e is fresh, i.e., it does not collide with existing exponents ($e \notin \text{exp}$), and that g^e is not blocked from being generated (i.e., $g^e \notin \text{blockedDHShares}$). If any of these

freshness checks fails, $\mathcal{F}_{\text{crypto}}$ asks the adversary again for another e until the check succeeds. Then, $\mathcal{F}_{\text{crypto}}$ adds e to exp , creates a new pointer ptr for user $\text{entity}_{\text{call}}$ pointing to e , sets $\text{expmap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, ptr) \leftarrow e$, and returns $(\text{GenExp}, ptr, g^e)$ to the user.

Note that we do not allow the adversary to corrupt an exponent upon generation. That is because **GenExp** models honest generation of an exponent by honest users; to model dishonest generation of exponents, one can use the **StoreExp** command (see below) to insert an externally generated (known) exponent into $\mathcal{F}_{\text{crypto}}$.

- **Mark group element as used** $[(\text{BlockDHShare}, h)]$. The user can instruct $\mathcal{F}_{\text{crypto}}$ to manually block a group element $h \in G$ from being generated during a **GenExp** request. This is useful for higher-level protocols to ensure that, if they receive some DH share h , no future **GenExp** request will output the same DH share even if h was not originally created by $\mathcal{F}_{\text{crypto}}$. Upon receiving this command, $\mathcal{F}_{\text{crypto}}$ adds h to blockedDHShares and returns $(\text{BlockDHShare}, \text{ok})$. See Section 5.1.3 for a discussion of this command.
- **Store an exponent** $[(\text{StoreExp}, e)]$. The user can also insert a new (known) exponent $e \in \{1, \dots, n\}$ into $\mathcal{F}_{\text{crypto}}$. Upon receiving this request, $\mathcal{F}_{\text{crypto}}$ prevents guessing of unknown exponents by ensuring that $e \notin \text{exp}_{\text{unknown}}$. If the check succeeds, e is added to $\text{exp}_{\text{known}}$, a new pointer ptr for this exponent is created, $\text{expmap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, ptr) \leftarrow e$ is set, and $(\text{StoreExp}, ptr)$ is returned to the user. If the check fails, $(\text{StoreExp}, \perp)$ is returned to the user.
- **Retrieve an exponent** $[(\text{RetrieveExp}, ptr)]$. The user can retrieve the exponent a pointer ptr refers to. In this case, $\mathcal{F}_{\text{crypto}}$ retrieves $e \leftarrow \text{expmap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, ptr)$, adds e to $\text{exp}_{\text{known}}$, and outputs $(\text{RetrieveExp}, e)$ to the user.
- **Generate a new Diffie-Hellman key** $[(\text{GenDHKey}, ptr, h)]$. A user can ask $\mathcal{F}_{\text{crypto}}$ to create a new key of type **dh-key** from some group element $h \in G$ and the exponent to which ptr points. When receiving this request, $\mathcal{F}_{\text{crypto}}$ adds h to the set blockedDHShares to ensure that the group element h will not be generated by future **GenExp** requests (cf. Section 5.1.3 for a discussion of this operation). $\mathcal{F}_{\text{crypto}}$ then retrieves $e \leftarrow \text{expmap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, ptr)$ and further checks whether $h = g^e$; if so, then e is marked as known, i.e., it is added to the set $\text{exp}_{\text{known}}$ (see Section 5.1.3 for an explanation). A new pointer ptr' to a DH key is generated and then set to a key as follows:

First, $\mathcal{F}_{\text{crypto}}$ checks whether a key k (of type **dh-key**) has already been generated by the group elements g^e and h . If so, then the pointer ptr' is set to this key, i.e., $\mathcal{F}_{\text{crypto}}$ assigns $\text{keymap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, ptr') \leftarrow k$. This ensures that key generation is deterministic and both owners of the DH shares generate the same key. Otherwise, a new key is generated as follows.

If h belongs to an unknown exponent (i.e., there exists $d \in \text{exp}_{\text{unknown}}$ such that $h = g^d$) and e is marked unknown, then the adversary is asked via a restricting message to provide a fresh

unknown key $k \in G$ of type **dh-key**. Formally, this is done by sending the restricting message (**ProvideDHKey**, $e, d, \text{unknown}$) on the network.⁴³ The functionality $\mathcal{F}_{\text{crypto}}$ ensures that k is fresh, i.e., $(\text{dh-key}, k) \notin \text{keys}$ (and keeps asking for a new k if this is not the case), and then sets $\text{keymap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, \text{ptr}') \leftarrow (\text{dh-key}, k)$.

If the checks regarding the exponents fail, i.e., there is no $d \in \text{exp}_{\text{unknown}}$ such that $h = \mathbf{g}^d$ or e is marked known, then the adversary is asked via a restricting message to provide a known key $k \in G$ of type **dh-key**. Formally, this is done by sending the restricting message (**ProvideDHKey**, e, h, known) on the network. The functionality $\mathcal{F}_{\text{crypto}}$ prevents key-guessing of unknown keys, i.e., if $(\text{dh-key}, k) \in \text{keys}_{\text{unknown}}$, the functionality asks for another key. The functionality then sets $\text{keymap}(\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}}, \text{ptr}') \leftarrow (\text{dh-key}, k)$. Furthermore, if there is no $d \in \text{exp}$ such that $h = \mathbf{g}^d$, then the exponent e is marked known by adding it to $\text{exp}_{\text{known}}$ even if it was unknown before (see Section 5.1.3 for an explanation).

In any case, $\mathcal{F}_{\text{crypto}}$ records that the group elements \mathbf{g}^e and h have been used to create a key k and returns (**GenDHKey**, ptr') to the user.

In addition to the above commands, we improve the overall expressiveness and ease of use of $\mathcal{F}_{\text{crypto}}$ for building higher-level protocols as follows:

- As explained in Section 5.1.1, in [97] the adversary was allowed to corrupt a fresh key generated via the **New** command. As this command models a local computation performed by honest parties, we remove this ability. Keys generated by this command are now always uncorrupted and thus unknown. This follows the same reasoning given for the **GenExp** command; in particular, (known) keys generated by corrupted users can already be modeled using the **Store** command. We note that this change is not necessary for our realization $\mathcal{P}_{\text{crypto}}$ of $\mathcal{F}_{\text{crypto}}$ (see Section 5.1.4) but its purpose is rather to make $\mathcal{F}_{\text{crypto}}$ easier to use for building higher-level protocols. In particular, such higher-level protocols can now decide for themselves whether and when a key is generated honestly, including the special case where the attacker may freely decide whether a new symmetric key is corrupted (just as in the formulation of $\mathcal{F}_{\text{crypto}}$ from [97]).
- Every time $\mathcal{F}_{\text{crypto}}$ adds a symmetric key (t, k) to $\text{keys}_{\text{known}}$, it sends a restricting message (**AddedKnownKey**, (t, k)) to the adversary and waits for any response on the network before continuing. This makes explicit that $\mathcal{F}_{\text{crypto}}$ does not provide any guarantees on the secrecy of actual values or the status of keys. As the adversary is already asked to provide all unknown keys, there is no need to also leak them. While this change is, again, not necessary for our realization of $\mathcal{F}_{\text{crypto}}$, it reduces the burden imposed on simulators when using $\mathcal{F}_{\text{crypto}}$ as part of other composed ideal protocols (such as the ideal key exchange functionalities that we define in Section 5.2). This change allows a simulator for such protocols to maintain the

⁴³We note that it is important to tell the adversary the known/unknown status for our realization as this determines whether our simulator responds with \mathbf{g}^{ed} or $\mathbf{g}^c, c \xleftarrow{\$} \{1, \dots, n\}$. Also note that the adversary knows the actual values of e and d anyway, so there is no security loss by directly sending these values on the network.

same view of known and unknown keys within an internal simulation. For the same reason, our extension also notifies the adversary whenever an exponent e is added to $\text{exp}_{\text{known}}$ by sending a restricting message (`AddedKnownExponent`, e). We have kept this implicit in the above description to avoid clutter the important parts and because leaking the known status of exponents is not actually necessary to realize $\mathcal{F}_{\text{crypto}}$; in fact, our simulator given in Section 5.1.5 simply discards this information.

- We provide the network attacker/simulator with the ability to manually add new known/unknown symmetric keys to the key sets of $\mathcal{F}_{\text{crypto}}$ via a new (`AddKey`, k , *knownStatus*) command, where $k \notin \text{keys}$ and *knownStatus* $\in \{\text{known}, \text{unknown}\}$. Note that this additional command does not weaken the security guarantees provided by $\mathcal{F}_{\text{crypto}}$ to higher-level protocols: the attacker cannot change the status of existing keys, and new unknown keys will still be generated to be fresh. Just as the previous change, this change is also not necessary for our realization of $\mathcal{F}_{\text{crypto}}$ (our simulator from Section 5.1.5 does not even use the `AddKey` command) but the purpose is rather to further reduce the burden imposed on simulators when using $\mathcal{F}_{\text{crypto}}$ as part of other composed ideal protocols (such as the ones from Section 5.2). This addition mainly allows a simulator for such protocols to notify $\mathcal{F}_{\text{crypto}}$ about (unknown) keys that exist within an internal simulation and for which $\mathcal{F}_{\text{crypto}}$ should prevent the environment from guessing those simulated keys whenever the environment adds a (known) key.
- Recall from Section 5.1.1 that the adversary may statically corrupt private keys. We now allow the adversary to corrupt private signing keys dynamically, i.e., these keys can be corrupted by the adversary at any time. Corruption of private encryption keys remains static.
- In [97], the adversary was free to never provide public/private signing/encryption keys for some (or all) parties, causing the corresponding operations to fail, even though $\mathcal{F}_{\text{crypto}}$ was supposed to model a setting where all parties are assumed to have already generated (and published) their personal key pairs before the start of a protocol run. Due to responsive environments, we can not properly model this setting. More specifically, if an asymmetric operation is to be performed for a key pair that has not been initialized yet, then the adversary is first asked via a restricting message to provide a key pair (and potentially corrupt it). As a result of this change, $\mathcal{F}_{\text{crypto}}$ is easier to use for higher-level protocols, which no longer have to manually deal with errors caused by undefined keys.
- The ideal functionality $\mathcal{F}_{\text{crypto}}$ models cryptographic operations that, in reality, are performed locally and do not actually involve any network communication. In particular, if one, e.g., signs a message in reality, then the signature is returned immediately such that the computation can continue right away. We leverage the power of restricting messages to capture this property in our extension of $\mathcal{F}_{\text{crypto}}$. That is, we define network messages to be restricting if they are sent while some operation is in progress. This includes network messages mentioned in Section 5.1.1, which are also changed to be restricting in the

same situation. Furthermore, if an attacker does not provide an expected response, e.g., because a new symmetric key was not fresh, then the restricting message is repeated. As a result, our version of $\mathcal{F}_{\text{crypto}}$ now properly models local computations where the adversary/environment cannot interfere with the protocol run in unintended ways but responses to local computations are rather generated “immediately”.

5.1.3. Remarks and Discussion of our Extension

The ideal functionality $\mathcal{F}_{\text{crypto}}$ marks DH keys as unknown only if they were generated from two unknown exponents. In particular, if an unknown exponent e is used with a group element h which was not created by $\mathcal{F}_{\text{crypto}}$, then the resulting key is marked known and hence no security guarantees are given for this key. Otherwise, $\mathcal{F}_{\text{crypto}}$ would not be realizable: In a realization of $\mathcal{F}_{\text{crypto}}$, an environment might know the exponent d such that $h = \mathbf{g}^d$, in which case it is trivial to compute the DH key \mathbf{g}^{ed} . Hence, if $\mathcal{F}_{\text{crypto}}$ used such a key to derive other keys ideally, an environment could easily distinguish $\mathcal{F}_{\text{crypto}}$ and its realization $\mathcal{P}_{\text{crypto}}$.

We want to use the Decisional Diffie-Hellman (DDH) assumption for realizing $\mathcal{F}_{\text{crypto}}$. However, $\mathcal{F}_{\text{crypto}}$ provides operations that are not covered by the DDH experiment. To be more precise, the environment can use $\mathcal{F}_{\text{crypto}}$ to compute $(\mathbf{g}^e)^e = \mathbf{g}^{e^2}$ and h^e (where e is a secret exponent stored in $\mathcal{F}_{\text{crypto}}$ and h is an arbitrary group element not generated by $\mathcal{F}_{\text{crypto}}$). By the DDH assumption, we cannot guarantee that the environment does not learn anything about e itself or keys created with e in these cases. Indeed, if an adversary is able to calculate the function $f_a(h) \rightarrow h^a$ or the function $f'(\mathbf{g}^e) \rightarrow \mathbf{g}^{e^2}$ (where a is one of the exponents from the DDH experiment and h, \mathbf{g}^e are arbitrary group elements) he can break the DDH assumption (see, e.g., [1, 104] for details). Thus, we have to consider e to be known if it is used for such an operation, i.e., all keys derived from e will be just as in the realization without providing any ideal security guarantees.

The need for the `blockedDHShares` set and the `BlockDHShare` command might seem surprising at first: Typically, cryptographic libraries do not keep track of “seen” DH shares and then block them from being generated. However, this set and the corresponding command are necessary to lift an important property from the realization $\mathcal{P}_{\text{crypto}}$ to the case of the idealization. In the realization, it happens with negligible probability only that \mathbf{g}^e for some fresh exponent e equals some DH share h which might already have been used to create a key. However, $\mathcal{F}_{\text{crypto}}$ allows the adversary to choose the actual value of e , i.e., he might choose the exponent such that $\mathbf{g}^e = h$. To get the same guarantees as or even stronger guarantees than in the realization, $\mathcal{F}_{\text{crypto}}$ uses the set `blockedDHShares` to record all DH shares it has seen so far. With this set $\mathcal{F}_{\text{crypto}}$ makes sure that when creating a new exponent the corresponding DH share is “fresh”, i.e., does not belong to `blockedDHShares`. The command `BlockDHShare` allows higher-level protocols to notify $\mathcal{F}_{\text{crypto}}$ about DH shares they obtain such that $\mathcal{F}_{\text{crypto}}$ can add these shares to `blockedDHShares`. For example, when a responder in a DH-based key exchange protocol receives a DH share h , he would first add this share to $\mathcal{F}_{\text{crypto}}$ using the command `BlockDHShare` and then create his

own share. By this, $\mathcal{F}_{\text{crypto}}$ can make sure that the responder’s share is indeed fresh, and in particular, different from h . The responder can then use the `GenDHKey` command to derive a fresh DH key from h and his own DH share. We note that the `blockedDHShares` set does not exist in $\mathcal{P}_{\text{crypto}}$ while the `BlockDHShare` command in fact does nothing. Thus, after replacing $\mathcal{F}_{\text{crypto}}$ with $\mathcal{P}_{\text{crypto}}$, every call of the `BlockDHShare` command can be omitted entirely, yielding a natural protocol implementation.

While we opted for a definition of $\mathcal{F}_{\text{crypto}}$ with a single DH key type for simplicity, it is trivial to extend $\mathcal{F}_{\text{crypto}}$ to multiple DH key types to model two or more groups that are used simultaneously (the same technique has already been used in [112] to support multiple (different) algorithms for each of the existing symmetric operations in $\mathcal{F}_{\text{crypto}}$). Such an extension would use one set `exp` and `expknown` and separate pointers to exponents for every DH key type. All results presented in the following carry over to this setting.

5.1.4. Realization $\mathcal{P}_{\text{crypto}}$ of the Ideal Functionality for Cryptographic Primitives

In this section, we construct a realization $\mathcal{P}_{\text{crypto}}$ of $\mathcal{F}_{\text{crypto}}$. This realization implements all operations of $\mathcal{F}_{\text{crypto}}$ via common cryptographic schemes in a natural and expected way. Our realization is based on the realization of the original $\mathcal{F}_{\text{crypto}}$ from [97], but adds support for Diffie-Hellman key exchanges and the further improvements given in Section 5.1.2. In Section 5.1.5, we then prove that $\mathcal{P}_{\text{crypto}}$ indeed realizes $\mathcal{F}_{\text{crypto}}$ under standard cryptographic assumptions. This proof is quite involved and includes several reductions and hybrid arguments, but due to the composition theorems this is a once and for all effort. As already mentioned, protocol designers can use $\mathcal{F}_{\text{crypto}}$ for their security analysis and then replace it with $\mathcal{P}_{\text{crypto}}$ without re-doing any proofs.

Formally, $\mathcal{P}_{\text{crypto}}$ consists of a single public role `crypto` implemented by one machine which accepts all entities, just as $\mathcal{F}_{\text{crypto}}$. It is parameterized with three schemes Σ_{authenc} , $\Sigma_{\text{unauthenc}}$, Σ_{pub} for (un-)authenticated symmetric and public key encryption, a MAC scheme Σ_{MAC} , a signature scheme Σ_{Sig} , an algorithm `GroupGen`(η) with the same properties as for $\mathcal{F}_{\text{crypto}}$, and two families of pseudo-random functions (PRF) $F = \{F_\eta\}_{\eta \in \mathbb{N}}$ and $F' = \{F'_\eta\}_{\eta \in \mathbb{N}}$ that take as input a key and a seed and output a key (see Appendix E for standard definitions of these primitives and corresponding security notions). During initialization, $\mathcal{P}_{\text{crypto}}$ executes `GroupGen` and stores the result. Just as $\mathcal{F}_{\text{crypto}}$, the realization $\mathcal{P}_{\text{crypto}}$ keeps track of symmetric key types and uses them to decide which primitives may be executed with a given key (the family F is used for deriving keys from keys of type `pre-key`, while F' is used for key derivation from keys of type `dh-key`). The realization keeps track of the corruption status of keys in order to answer corruption status requests from the environment, but its behavior of cryptographic operations is independent of the corruption status otherwise. It also does not maintain the sets `keys`, `keysknown`, `exp`, `expknown` and does not include any checks on freshness or key/exponent collisions.

We now give a detailed description of how each of the operations of $\mathcal{F}_{\text{crypto}}$ is implemented in $\mathcal{P}_{\text{crypto}}$; see Appendix F for a formal specification using our template. We give the operations in

the same order as they were presented in the previous section, i.e., operations for Diffie-Hellman key exchanges introduced by our extension are given at the end.

- **Generating fresh, symmetric keys** $[(\text{New}, t)]$. To generate a new key of type t , $\mathcal{P}_{\text{crypto}}$ samples a key k uniformly at random from $\{0, 1\}^\eta$, creates a new pointer ptr for the (t, k) , and returns the pointer to the user.
- **Establishing pre-shared keys** $[(\text{GetPSK}, t, name)]$. Upon a request by a user to establish a pre-shared key of type t and with name $name$, $\mathcal{P}_{\text{crypto}}$ checks whether a key (t, k) is recorded for $(t, name)$ (and therefore $(t, name)$ is not corrupted). In this case, $\mathcal{P}_{\text{crypto}}$ creates a new pointer to (t, k) and returns it to the user. Otherwise, i.e., no key is recorded for $(t, name)$, we distinguish the following cases:

If $(t, name)$ is corrupted, the request is forwarded via a restricting message to the adversary who is supposed to provide a key k . Then, $\mathcal{P}_{\text{crypto}}$ creates a new pointer to (t, k) , records this pointer as corrupted, and returns it to the user.

If $(t, name)$ is not corrupted (but no key has been recorded for $(t, name)$), the request is forwarded to the adversary who is asked whether the key is corrupted. In the case the key is corrupted the adversary provides the key k , $\mathcal{P}_{\text{crypto}}$ marks $(t, name)$ as corrupted PSK, generates a new pointer to (t, k) , records this pointer as corrupted, and returns it to the user. If the key is not corrupted, $\mathcal{P}_{\text{crypto}}$ chooses the key k uniformly at random from $\{0, 1\}^\eta$ and records (t, k) for $(t, name)$. Then, $\mathcal{P}_{\text{crypto}}$ creates a new pointer to (t, k) and returns the pointer to the user.

- **Store, Retrieve, and Equality test** $[(\text{Store}, t, k), (\text{Retrieve}, ptr), \text{or } (\text{Equal?}, ptr, ptr')]$. These requests are handled as in $\mathcal{F}_{\text{crypto}}$, except that keys and $\text{keys}_{\text{known}}$ are not maintained.
- **Public key requests** $[(\text{GetPubKeyPKE}, pid') \text{ or } (\text{GetPubKeySig}, pid')]$. Upon a request of a user to obtain the public key of party pid' , if that public key (and the corresponding private key) has already been generated, $\mathcal{P}_{\text{crypto}}$ returns the public key. Otherwise, $\mathcal{P}_{\text{crypto}}$ asks the adversary whether he wants to corrupt the public/private key pair of party pid' via a restricting message.

If the adversary corrupts the key pair, then he also provides the values of the public and private keys. These are stored by $\mathcal{P}_{\text{crypto}}$.

If the adversary does not corrupt the key pair, then $\mathcal{P}_{\text{crypto}}$ executes the key generation algorithm of the encryption or signature scheme, respectively, with input 1^η to obtain and store a fresh key pair.

In both cases, $\mathcal{P}_{\text{crypto}}$ finishes by returning the public key of party pid' to the user.

- **Key derivation** $[(\text{Derive}, ptr, t', s)]$. Upon a request by a user to derive a new key of type t' from a key k of type $t \in \{\text{pre-key}, \text{dh-key}\}$ with seed s , $\mathcal{P}_{\text{crypto}}$ computes $k' = F_\eta(k, \text{Tag}_{t'}(s))$

or $k' = F'_\eta(k, \text{Tag}_{t'}(s))$ (the PRF is chosen depending on the type t as explained above), creates a new pointer ptr to (t', k') for this user, and returns ptr to the user. Note that the PRF is used with the seed $\text{Tag}_{t'}(s)$ instead of just s . This guarantees that keys of different types are derived with different seeds. This kind of tagging is common also in real world protocols in order to ensure that the same (derived) key is not used for different cryptographic operations.

- **Encryption under symmetric and public keys** $[(\text{Enc}, ptr, x)$ or $(\text{PKEnc}, pid', pk, x)]$. Upon a request by a user to encrypt a well-tagged message x under a symmetric key k of type $t \in \{\text{authenc-key}, \text{unauthenc-key}\}$, all pointers in x are replaced by the keys they refer to (just as in $\mathcal{F}_{\text{crypto}}$). Unlike $\mathcal{F}_{\text{crypto}}$, the resulting message, say x' , is then encrypted under k by running the encryption algorithm of Σ_{authenc} or $\Sigma_{\text{unauthenc}}$, depending on t , and the resulting ciphertext y is returned to the user.

Requests for encryption under public keys are handled similarly using the encryption algorithm of Σ_{pub} and the public key pk contained in the request. Note that, if the public/private key pair of party pid' has not been initialized yet, then this is done first using the same procedure as for public key requests.

- **Decryption under symmetric and private keys** $[(\text{Dec}, ptr, y)$ or $(\text{PKDec}, y)]$. Decryption requests for a ciphertext y under a symmetric key k of type $t \in \{\text{authenc-key}, \text{unauthenc-key}\}$ are answered by running decryption algorithm of Σ_{authenc} or $\Sigma_{\text{unauthenc}}$, depending on t , on the inputs k and y . If the decryption is successful and returns a well-tagged message, say x' , then all keys in x' are replaced by new pointers (just as in $\mathcal{F}_{\text{crypto}}$) and the resulting message x is returned to the party. Otherwise, an error is returned.

Requests for decryption under private keys are handled similarly using the decryption algorithm of Σ_{pub} and the private key recorded for the current party pid_{call} . Note that, if the public/private key pair of the current party has not been initialized yet, then this is done first using the same procedure as for public key requests.

- **Creating and verifying MACs** $[(\text{MAC}, ptr, x)$ or $(\text{MACVerify}, ptr, x, \sigma)]$. Upon a request to MAC a message x under a key k of type mac-key , $\mathcal{P}_{\text{crypto}}$ simply returns the MAC computed using the MAC algorithm of Σ_{MAC} . Upon a MAC verification request, $\mathcal{P}_{\text{crypto}}$ simply returns the result of the MAC verification algorithm of Σ_{MAC} .
- **Creating and verifying signatures** $[(\text{Sign}, x)$ or $(\text{SigVerify}, pid', pk, x, \sigma)]$. Upon a request to sign a message x , $\mathcal{P}_{\text{crypto}}$ simply returns the signature computed using the signing algorithm of Σ_{Sig} and the private key recorded for the current party pid_{call} . Upon signature verification request, $\mathcal{P}_{\text{crypto}}$ simply returns the result of the signature verification algorithm of Σ_{Sig} using the public key pk contained in the request. Note that, if the public/private key pair of the current party pid_{call} respectively the party pid' has not been initialized yet, then this is done first using the same procedure as for public key requests.

- **Generating fresh nonces** [(NewNonce)]. Upon generation of fresh nonces, $\mathcal{P}_{\text{crypto}}$ chooses the nonce uniformly at random from $\{0, 1\}^\eta$ and gives it to the user.
- **Corruption status request**. [(IsSymKeyCorrupt?, ptr), (IsPKEncKeyCorrupt?, pid), and (IsSigKeyCorrupt?, pid)]. These requests are handled just as in $\mathcal{F}_{\text{crypto}}$.
- **Get generated group** [(GetDHGroup)]. $\mathcal{P}_{\text{crypto}}$ returns the group description (G, n, g) that was generated during the initialization.
- **Generate a fresh exponent** [(GenExp)]. $\mathcal{P}_{\text{crypto}}$ chooses $e \xleftarrow{\$} \{1, \dots, n\}$, creates a pointer to e , and outputs $(\text{GenExp}, ptr, g^e)$ to the user.
- **Mark group element as used** [(BlockDHShare, h)]. $\mathcal{P}_{\text{crypto}}$ returns $(\text{BlockDHShare}, \text{ok})$.
- **Store an exponent** [(StoreExp, e)]. $\mathcal{P}_{\text{crypto}}$ stores $e \in \{1, \dots, n\}$, creates a new pointer ptr for this exponent, and returns ptr to the user.
- **Retrieve an exponent** [(RetrieveExp, ptr)]. $\mathcal{P}_{\text{crypto}}$ outputs the exponent e to which ptr points.
- **Generate a new Diffie-Hellman key** [(GenDHKey, ptr, h)]. Given a pointer ptr to an exponent e and a group element $h \in G$, $\mathcal{P}_{\text{crypto}}$ calculates the key $k := h^e$. A new pointer ptr' pointing to k is created and returned to the user.

5.1.5. Proving that $\mathcal{P}_{\text{crypto}}$ realizes $\mathcal{F}_{\text{crypto}}$

In this section, we prove that $\mathcal{P}_{\text{crypto}}$ realizes $\mathcal{F}_{\text{crypto}}$. We want to use standard cryptographic assumptions for this, but these assumptions provide security only in a certain context. For example, standard assumptions for symmetric encryption do not provide any security guarantees in the presence of key cycles where a key is (indirectly) encrypted by itself. This is why reasonable higher-level protocols generally avoid situations that are not covered by cryptographic assumptions; in contrast, environments in universal composability models are free to use $\mathcal{P}_{\text{crypto}}$ and $\mathcal{F}_{\text{crypto}}$ in any way they want and, in particular, they may create settings where the assumptions fail. In order to capture the expected use of $\mathcal{P}_{\text{crypto}}/\mathcal{F}_{\text{crypto}}$ as a subroutine of a reasonable higher-level protocol, we thus slightly restrict environments such that they expose certain natural properties of higher-level protocols. We note that this approach is established in the literature (see, e.g., [9]) and has also successfully been used for the original version of $\mathcal{F}_{\text{crypto}}$ from [97]. The next paragraphs describe and discuss our restriction in more detail.

Recall that we want to use the DDH assumption in order to prove $\mathcal{P}_{\text{crypto}} \leq \mathcal{F}_{\text{crypto}}$. The general idea is that the simulator in the proof of this statement will provide g^{ab} when asked for a known DH key, and g^c (for $c \xleftarrow{\$} \{1, \dots, n\}$) in case of an unknown DH key. However, this leads to the so-called commitment problem: Once the simulator has committed to g^c for an unknown key, neither a nor b may become known; otherwise the environment could calculate

\mathbf{g}^{ab} on its own and distinguish the real from the ideal world. We note that the commitment problem is not specific to our modeling of DH keys, but rather is a general issue in universal composability models that occurs in many cryptographic contexts (see, e.g., [42]). To address this problem, we restrict the environment (the higher-level protocol that uses $\mathcal{F}_{\text{crypto}}$) to not cause the commitment problem. That is, once an unknown exponent e (respectively the key share \mathbf{g}^e) has been used to create an unknown DH key \mathbf{g}^c , the environment may not perform any actions that cause e to become known. More specifically, the environment may no longer manually retrieve e from $\mathcal{F}_{\text{crypto}}$, create a DH key from e and the corresponding DH share \mathbf{g}^e (yielding \mathbf{g}^{e^2}), or use e with a DH share h where $\mathcal{F}_{\text{crypto}}$ does not know the secret exponent of h . Observe, however, that most protocols meant to achieve perfect forward secrecy fulfill this restriction: In such protocols, an exponent e is generated, used exactly once to generate a DH key, and then deleted from memory. Hence, after a key was created, the protocol will never access/use the exponent in any way again, and thus, also never cause the commitment problem. For example, this holds true for all protocols analyzed in Section 5.3. It might be possible to relax these restrictions, enabling an analysis of protocols that reuse the same exponent, by using the non-standard PRF-ODH assumption⁴⁴ [23, 79, 84] instead of the DDH assumption. Exploring a formulation of $\mathcal{F}_{\text{crypto}}$ based on this assumption is potential future work.

A similar commitment problem exists for encryption and key derivation. However, again most real world protocols do not cause this problem (see also [97] where this is discussed in detail for the original version of $\mathcal{F}_{\text{crypto}}$). This leads us to the following formal restriction of environments:

We say that an environment \mathcal{E} *does not cause the commitment problem* (is *non-committing*), if the following happens with negligible probability only: i) in a run of $\mathcal{E} | \mathcal{F}_{\text{crypto}}$, after an unknown key k has been used to encrypt a message or derive a new key, k becomes known later on in the run, i.e., is marked known by $\mathcal{F}_{\text{crypto}}$, and ii) in a run of $\mathcal{E} | \mathcal{F}_{\text{crypto}}$, after an unknown exponent e or the corresponding group element \mathbf{g}^e has been used to create an unknown DH key k , e becomes known later on in the run, i.e., is marked known by $\mathcal{F}_{\text{crypto}}$.

Besides the commitment problem, we also have to take care of key cycles. As mentioned, standard security definitions such as IND-CCA2, which we want to use for our realization, do not provide any security in this case. Indeed, security in the presence of key cycles is usually not required: real world protocols generally do not encrypt keys anymore once these keys have been used for the first time. Obviously, such protocols also do not produce key cycles. This observation leads to the following natural restriction of environments:

An environment \mathcal{E} is called *used-order respecting* if the following happens with negligible probability only: in a run of $\mathcal{E} | \mathcal{F}_{\text{crypto}}$ an unknown key k (i.e., k is marked unknown in $\mathcal{F}_{\text{crypto}}$) which has been used for encryption or key derivation at some point is encrypted itself by an

⁴⁴Informally, the PRF-ODH assumption states that, given a Diffie-Hellman key \mathbf{g}^{ab} which is used to key a pseudo random function $f(\mathbf{g}^{ab}, s)$, no adversary that knows \mathbf{g}^a and \mathbf{g}^b can distinguish a challenge output of the PRF from random, even when given access to an oracle $O(h, s) := f((h)^a, s)$ (where h is a group element and s is a salt).

unknown key k' used for the first time later than k .

We call an environment *well-behaved* if it is used-order respecting and does not cause the commitment problem. For such well-behaved environments, we can show that $\mathcal{P}_{\text{crypto}} \leq \mathcal{F}_{\text{crypto}}$ if all cryptographic primitives fulfill the standard cryptographic assumptions. As explained above, many real world protocols fulfill the requirements of well-behaved environments, and hence, if they are analyzed using $\mathcal{F}_{\text{crypto}}$, one can replace $\mathcal{F}_{\text{crypto}}$ with its realization afterwards.

In the following theorem, formally, instead of considering a specific set of environments, which would require re-stating and re-proving the underlying compositional model, we use a higher-level protocol $\mathcal{F}^* = (\text{crypto}^*)$ to manually enforce the properties of well-behaved environments for all environments. This technique has already been used successfully in the realization proof of the original $\mathcal{F}_{\text{crypto}}$ in [97]. The protocol \mathcal{F}^* consists of one machine (with one role) that uses $\mathcal{P}_{\text{crypto}}/\mathcal{F}_{\text{crypto}}$ as a private subroutine and forwards all messages to/from the environment while checking that the conditions of well-behaved environments are fulfilled.⁴⁵ If at some point one of the conditions is violated, instead of forwarding the current message, \mathcal{F}^* stops and blocks all future communication. We obtain the following theorem:

Theorem 5.1. *Let $\Sigma_{\text{unauth-enc}}$, $\Sigma_{\text{auth-enc}}$, Σ_{pub} be encryption schemes, Σ_{mac} be a MAC scheme, Σ_{sig} be a signature scheme, **GroupGen** be an algorithm as above, F be a family of pseudo-random functions, and F' be a family of pseudo-random functions for **GroupGen**. Let $\mathcal{P}_{\text{crypto}}$ be parameterized with these algorithms. Let $\mathcal{F}_{\text{crypto}}$ be parameterized with **GroupGen** and a leakage algorithm L which leaks exactly the length of the input. Then,*

$$\mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}} \leq \mathcal{F}^* \parallel \mathcal{F}_{\text{crypto}}$$

*if $\Sigma_{\text{unauth-enc}}$ and Σ_{pub} are IND-CCA2 secure, $\Sigma_{\text{auth-enc}}$ is IND-CPA and INT-CTXT secure, Σ_{mac} and Σ_{sig} are UF-CMA secure, **GroupGen** always outputs groups with $n \geq 2$ and such that the DDH assumption holds true for **GroupGen**.⁴⁶*

Before we prove this theorem, let us emphasize that while this theorem is stated and shown using an additional wrapper \mathcal{F}^* , one can actually remove this wrapper when $\mathcal{F}_{\text{crypto}}/\mathcal{P}_{\text{crypto}}$ is used by a higher-level protocol that constitutes a well-behaved environment. As mentioned above, this is the case for most real world protocols (in the case of DH key exchange, those real world protocols that are meant to achieve perfect forward secrecy). Formally, we have the following corollary:

Corollary 5.1. *Let $\mathcal{P}_{\text{crypto}}$ and $\mathcal{F}_{\text{crypto}}$ be such that the conditions of Theorem 5.1 are fulfilled. Let \mathcal{Q} be a higher-level protocol using $\mathcal{P}_{\text{crypto}}/\mathcal{F}_{\text{crypto}}$ such that in runs of $\mathcal{Q} \parallel \mathcal{P}_{\text{crypto}}$ and*

⁴⁵Note that this can be done by observing the I/O traffic and asking $\mathcal{F}_{\text{crypto}}$ about the corruption status of keys.

⁴⁶See Appendix E for the formal definitions of these security notions. We have to require $n \geq 2$ because the trivial group which contains only the neutral element fulfills the DDH assumption, but is not suitable for realizing $\mathcal{F}_{\text{crypto}}$. In particular, collisions of randomly chosen exponents do not happen with negligible probability if there is only one element. Alternatively, we could also require that n grows exponentially in η .

$\mathcal{Q} \parallel \mathcal{F}_{\text{crypto}}$ (with an arbitrary responsive environment) the conditions of well-behaved environments for $\mathcal{P}_{\text{crypto}}/\mathcal{F}_{\text{crypto}}$ are met with overwhelming probability. Then:

$$\mathcal{Q} \parallel \mathcal{P}_{\text{crypto}} \leq \mathcal{Q} \parallel \mathcal{F}_{\text{crypto}}$$

Proof. Let \mathcal{Q}' be the protocol that runs as \mathcal{Q} but uses \mathcal{F}^* as a subroutine instead of directly connecting to $\mathcal{P}_{\text{crypto}}/\mathcal{F}_{\text{crypto}}$. Let \mathcal{S} be the dummy simulator that only forwards messages. For any responsive environment $\mathcal{E} \in \text{Env}(\mathcal{Q} \parallel \mathcal{P}_{\text{crypto}})$, we have that $\{\mathcal{E}, \mathcal{Q} \parallel \mathcal{P}_{\text{crypto}}\}$ and $\{\mathcal{E}, \mathcal{S}, \mathcal{Q}' \parallel \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}\}$ behave identical except for a negligible set of runs where the conditions of well-behaved environments are violated. Therefore, we have $\mathcal{Q} \parallel \mathcal{P}_{\text{crypto}} \leq \mathcal{Q}' \parallel \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}$ and, by an analogous argument, $\mathcal{Q}' \parallel \mathcal{F}^* \parallel \mathcal{F}_{\text{crypto}} \leq \mathcal{Q} \parallel \mathcal{F}_{\text{crypto}}$. Theorem 5.1 and the composition theorem (cf. Corollary 4.1) imply that $\mathcal{Q}' \parallel \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}} \leq \mathcal{Q}' \parallel \mathcal{F}^* \parallel \mathcal{F}_{\text{crypto}}$, which in turn implies the corollary by transitivity of the \leq relation (Lemma 3.6). \square

We now show Theorem 5.1. The proof is quite involved: it consists of a series of hybrid systems where we replace parts of $\mathcal{P}_{\text{crypto}}$ with the ideal versions used in $\mathcal{F}_{\text{crypto}}$ and then show that no environment can distinguish these replacements. Each of these steps involves several reductions and hybrid arguments itself. In particular, some of these reductions are intertwined with each other, as, e.g., the security of symmetric encryption and key derivation rely on each other.

Proof of Theorem 5.1. Let us first give an overview of the proof. We start by defining a simulator \mathcal{S} , which uses and extends the general ideas for simulation from the original $\mathcal{F}_{\text{crypto}}$ [97]. The simulator is mostly straightforward in that it provides the algorithms used by $\mathcal{P}_{\text{crypto}}$ to $\mathcal{F}_{\text{crypto}}$ and generates keys (mostly) as in $\mathcal{P}_{\text{crypto}}$. As already mentioned, the main idea for our extension is that \mathcal{S} provides \mathbf{g}^c for unknown Diffie-Hellman keys, where c is chosen uniformly at random from $\{1, \dots, n\}$, and \mathbf{g}^{ab} for known ones. The proof itself then consists of series of hybrid systems:

In the first step, we define a hybrid system $\mathcal{P}_{\text{crypto}}^1$ where all asymmetric operations and nonce generation is handled as in $\mathcal{F}_{\text{crypto}}$ while all other operations are performed as in $\mathcal{P}_{\text{crypto}}$. The proof of this step is essentially the same as for the original $\mathcal{F}_{\text{crypto}}$ [97] since asymmetric operations work in basically the same way.

In the second hybrid system $\mathcal{P}_{\text{crypto}}^2$ we replace DH exponent handling with the ideal version. In particular, $\mathcal{P}_{\text{crypto}}^2$ prevents exponent guessing and collisions. Any distinguishing environment on this system can be reduced to the DDH assumption.

In the the third hybrid system $\mathcal{P}_{\text{crypto}}^3$ we replace real with ideal Diffie-Hellman key generation, however, without preventing key collisions or key guessing for those keys. That is, DH keys are now determined by the simulator in the way described above. This step requires a hybrid argument itself as we have to replace a polynomial number of unknown keys in the order of their creation. We can then reduce the distinguishing advantage of an environment for the r -th and $r + 1$ -th hybrid system to the DDH assumption. Importantly, we have to establish a

single negligible bound for the distinguishing advantage that is independent of r , as the sum of polynomially many different negligible functions is not necessarily negligible.

In the fourth hybrid system $\mathcal{P}_{\text{crypto}}^4$, symmetric encryption and key derivation are replaced with their ideal versions, and key guessing and key collision are prevented. Again, this step requires a hybrid argument which is quite involved as we have to consider symmetric encryption and key derivation simultaneously: All symmetric keys can be encrypted, thus the security of symmetric keys (including whether they can be guessed) depends on the security of the encryption scheme. However, Diffie-Hellman keys and key derivation keys can be used to create new symmetric keys, i.e., the security of the encryption scheme in turn depends on the security of the key derivation schemes. In the hybrid argument, we track the order in which unknown keys are used for the first time. The r -th hybrid system performs operations with the first r unknown keys ideally, and all other operations as in the realization. One can then reduce the distinguishing advantage of an environment for the r -th and $r + 1$ -th hybrid system to the security games of the encryption and key derivation schemes. Again, it is important to establish a negligible bound for the distinguishing advantage that is independent of r .

In the fifth and last step, we have to replace MACs with their ideal versions. Just as for the first step, the proof is essentially the same as in the original version of $\mathcal{F}_{\text{crypto}}$ [97] since our extension does not change the way MACs are computed and verified.

The simulator: The simulator \mathcal{S} works as follows:

- During initialization of $\mathcal{F}_{\text{crypto}}$, \mathcal{S} receives and stores the group (G, n, g) . \mathcal{S} responds by sending the algorithms from the parameters of $\mathcal{P}_{\text{crypto}}$ to $\mathcal{F}_{\text{crypto}}$.
- Upon generation of fresh keys, pre-shared keys, DH exponents, nonces, and initialization of public/private key pairs, \mathcal{S} chooses keys, exponents, and nonces just as $\mathcal{P}_{\text{crypto}}$ and forwards the results to $\mathcal{F}_{\text{crypto}}$ (including the corruption state, if applicable).
- Upon key derivation from an unknown key of type **pre-key**, \mathcal{S} chooses a key uniformly at random from $\{0, 1\}^\eta$. Otherwise, i.e., if the key of type **pre-key** is known, \mathcal{S} uses the pseudo-random function F just as $\mathcal{P}_{\text{crypto}}$.
- Upon key derivation from an unknown key of type **dh-key**, \mathcal{S} chooses a key uniformly at random from $\{0, 1\}^\eta$. Otherwise, i.e., if the key of type **dh-key** is known, \mathcal{S} uses the pseudo-random function F' just as $\mathcal{P}_{\text{crypto}}$.
- Upon generation of an unknown Diffie-Hellman key, \mathcal{S} chooses an exponent c uniformly at random from $\{1, \dots, n\}$ and returns the key g^c . Otherwise, i.e., for generation of a known Diffie-Hellman key, \mathcal{S} takes the exponent e and group element h from the request by $\mathcal{F}_{\text{crypto}}$ and returns the key h^e .
- Notifications from $\mathcal{F}_{\text{crypto}}$ of the type **AddedKnownKey** and **AddedKnownExponent** are ignored by \mathcal{S} , i.e., \mathcal{S} directly returns **ok** to $\mathcal{F}_{\text{crypto}}$.

- If $\mathcal{F}_{\text{crypto}}$ refuses to accept a response locally generated by the simulator (e.g., a new exponent for the **GenExp** command is actually not fresh and thus rejected), then \mathcal{S} resigns the simulation, i.e., it stops and blocks all future messages.
- Requests from the network to corrupt the signing key pair of some party pid are forwarded by \mathcal{S} to $\mathcal{F}_{\text{crypto}}$.

Observe that all runtime conditions are fulfilled: $\mathcal{P}_{\text{crypto}}$, $\mathcal{F}_{\text{crypto}}$, and the system $\{\mathcal{S}, \mathcal{F}_{\text{crypto}}\}$ are environmentally bounded. Here we need the requirements imposed on **GroupGen**, i.e., the algorithm runs in polynomial time (except for a negligible set of runs), group membership is efficiently decidable, and group operations are efficiently computable. Next, we can start with the first hybrid step.

Step 1: For this step, we define a machine $\mathcal{P}_{\text{crypto}}^1$ that works just as $\mathcal{P}_{\text{crypto}}$ except for signature handling, asymmetric encryption and decryption, and nonce generation, which work just as in $\mathcal{F}_{\text{crypto}}$. As these operations are unaffected by our extension concerning Diffie-Hellman key exchanges, the same argument as in [97] implies

$$\{\mathcal{E}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^1\} \quad (5.1)$$

for all responsive environments $\mathcal{E} \in \text{Env}(\mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}})$ based on IND-CCA2 security of Σ_{pub} and UF-CMA security of Σ_{sig} (cf. Appendix E.2 and E.4). Note that there are three main differences compared to the original proof: First, here we use responsive environments. However, because the original proof held for all environments, the same reasoning holds for the subset of all responsive environments in particular. Second, we use dynamic instead of static corruption for signatures. The same proof still holds in this case, as it relies on a result from [94] that was also proven for dynamic corruption. Third, here we use the iUC framework to specify $\mathcal{P}_{\text{crypto}}$ and $\mathcal{F}_{\text{crypto}}$. However, our specifications in the iUC framework are structurally and functionally very close to the original specifications in the IITM model and therefore the same arguments and proof techniques directly carry over.

Step 2: We define another machine $\mathcal{P}_{\text{crypto}}^2$ that works as $\mathcal{P}_{\text{crypto}}^1$ except for exponent handling, i.e., $\mathcal{P}_{\text{crypto}}^2$ guarantees freshness for unknown exponents and prevents exponent guessing for known exponents, just as $\mathcal{F}_{\text{crypto}}$. More specifically, $\mathcal{P}_{\text{crypto}}^2$ maintains the sets exp , $\text{exp}_{\text{known}}$, and blockedDHShares . The **GenExp** command asks the simulator to provide an exponent e which has to be fresh (i.e., $e \notin \text{exp}$ and $g^e \notin \text{blockedDHShares}$), and for **StoreExp** commands key guessing is prevented (i.e., $e \notin \text{exp}_{\text{unknown}}$). Upon receiving a **RetrieveExp** command, $\mathcal{P}_{\text{crypto}}^2$ marks the retrieved exponent as known. Upon receiving a **GenDHKey** command, $\mathcal{P}_{\text{crypto}}^2$ adds the DH share to blockedDHShares and marks the involved exponent as known iff $\mathcal{F}_{\text{crypto}}$ would mark it as known. Upon receiving a **BlockDHShare** command, $\mathcal{P}_{\text{crypto}}^2$ adds the group

element to `blockedDHShares`. We have to show that

$$\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^1\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^2\} \quad (5.2)$$

for all responsive environments $\mathcal{E} \in \text{Env}(\{\mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^1\})$.

For any such environment \mathcal{E} let $E_{\text{not-fresh}}$ be the event that, in a run of $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^1\}$, while processing a `GenExp` command the simulator provides an exponent e that is not fresh (because $e \in \text{exp}$ or $\mathbf{g}^e \in \text{blockedDHShares}$) and thus rejected. Furthermore, let $E_{\text{exp-guessing}}$ be the event that, in a run of $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^1\}$, the `StoreExp` command is used to store an exponent that already is in $\text{exp}_{\text{unknown}}$. Since both $\mathcal{P}_{\text{crypto}}^1$ and $\mathcal{P}_{\text{crypto}}^2$ behave exactly the same in runs that belong to neither $E_{\text{not-fresh}}$ nor $E_{\text{exp-guessing}}$, it suffices to show that both events have a negligible probability to prove (5.2).

Suppose $E_{\text{not-fresh}}$ was non-negligible, i.e., some exponent e is not fresh in a non-negligible set of runs. As the runtime of the environment \mathcal{E} is bounded by a polynomial $p_{\mathcal{E}}$, we have that there are at most polynomially many exponents e_i created during a run and at most polynomially many group elements h_i stored in `blockedDHShares`. Hence, when an exponent e is created, there is at most a polynomial number of group elements \mathbf{g}^{e_i} and $h_i \in \text{blockedDHShares}$ that \mathbf{g}^e it can collide with (note that $\mathbf{g}^{e_i} = \mathbf{g}^e$ iff $e_i = e$, i.e., we can talk about collisions of group elements only). As every new exponent e is also chosen independently of these group elements, $E_{\text{not-fresh}}$ being non-negligible implies that when choosing a single $e \xleftarrow{\$} \{1, \dots, n\}$ the probability for a collision of \mathbf{g}^e with a single (fixed) group element h is also non-negligible.

This allows for constructing an adversary A on the DDH assumption (cf. Appendix E.5). Recall that A receives the security parameter η , external input a , a group description $(\mathbf{G}, n, \mathbf{g})$, and a challenge $(\mathbf{g}^a, \mathbf{g}^b, h)$ where either $h = \mathbf{g}^{ab}$ (if $b' = 1$) or $h = \mathbf{g}^c$ for $c \xleftarrow{\$} \{1, \dots, n\}$ (if $b' = 0$). A has to guess the correct bit b' . It proceeds as follows: First, it generates an exponent $e \xleftarrow{\$} \{1, \dots, n\}$ and checks whether $\mathbf{g}^a = \mathbf{g}^e$. If not, A resigns by outputting 1. Otherwise, A uses $e = a$ to check whether $h = (\mathbf{g}^b)^a$ and outputs 1 if this check succeeds; it outputs 0 otherwise. It is trivial to see that the runtime of A is bounded by a polynomial.

We have that $\text{Adv}_{A, \text{GroupGen}}^{\text{DDH}} \geq f(1^\eta, a) \cdot \frac{1}{2}$ where f is a non-negligible function: Observe that A resigns for both $b' = 0$ and $b' = 1$ with the same probability and hence these runs do not influence the advantage of A . The adversary does not resign with a non-negligible probability f (as \mathbf{g}^e collides with \mathbf{g}^a with non-negligible probability, as explained above), in which case he will always guess correctly if $b' = 1$ and only guess incorrectly in case of $b' = 0$ if $c = a \cdot b$. As we assumed $n \geq 2$ in Theorem 5.1, this happens with probability at most $\frac{1}{2}$, which gives the claim. Because A violates the DDH assumption, we conclude that $E_{\text{not-fresh}}$ is negligible.

We still have to show that the event $E_{\text{exp-guessing}}$ is negligible. Suppose that it was non-negligible by contradiction. Then there is a non-negligible set of runs where there is some unknown exponent e such that the environment tries to use the `StoreExp` command to store e . This can be used by an adversary A to violate the DDH assumption.

Before we define A , observe that $\{\mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^1\}$ is environmentally bounded, as one eas-

ily verifies, and hence there is a polynomial q that bounds the runtime of the full system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^1\}$ with overwhelming probability. Thus, there is a non-negligible set of runs that are in $E_{\text{exp-guessing}}$ and that do not exceed the runtime bound q .

We can now define A : The adversary first guesses when the exponent e is created. Say, A guesses e to be created via the i -th **GenExp** command, where $1 \leq i \leq p_{\mathcal{E}}(\eta, |a|)$ (recall that there are at most $p_{\mathcal{E}}(\eta, |a|)$ exponents as the runtime of \mathcal{E} is bounded by $p_{\mathcal{E}}$). A then simulates a run of $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^1\}$ but with the group $(\mathbb{G}, n, \mathbf{g})$ from the DDH experiment. The output of the i -th **GenExp** command is replaced with \mathbf{g}^a . All following operations that are performed using the i -th exponent, such as DH key generation, are also instead performed using \mathbf{g}^a . If at some point \mathcal{E} tries to store a in $\mathcal{P}_{\text{crypto}}^1$, A remembers a for the next step and stops the simulation. Otherwise, A continues until either the i -th exponent is marked known, the runtime bound q is violated, or the run ends, in which case A resigns and outputs 1. If A does not resign, he can use a to calculate $(\mathbf{g}^b)^a$ and output 1 iff $h = \mathbf{g}^{ab}$. The adversary A runs in polynomial time as it simulates at most q steps of the system.

Observe that A is in fact able to simulate the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^1\}$ in the way described above, even though A replaced the i -th exponent e with a without knowing the actual value of a : If \mathcal{E} tries to retrieve a , then A resigns since the event $E_{\text{exp-guessing}}$ cannot occur anymore. If \mathcal{E} tries to generate a known DH key from exponent a and DH share h , where the secret exponent d such that $\mathbf{g}^d = h$ is not stored in exp , then again A resigns as the exponent a will be marked known. The same holds if \mathcal{E} tries to generate a DH key from a with \mathbf{g}^a . In every other case of DH key generation from a , the exponent will be used with some DH share \mathbf{g}^d where $d \in \text{exp}$ and $d \neq a$. Hence, A knows d and can calculate $(\mathbf{g}^a)^d$ even without knowing a .

Overall, we have that $\text{Adv}_{A, \text{GroupGen}}^{\text{DDH}} \geq \frac{1}{p_{\mathcal{E}}(\eta, |a|)} \cdot f(1^\eta, a) \cdot \frac{1}{2}$ for a polynomial $p_{\mathcal{E}}$ and a non-negligible function f : First observe that A resigns for both $b' = 0$ and $b' = 1$ with the same probability, so these runs do not affect its advantage. The adversary does not resign if he simulates a run where $E_{\text{exp-guessing}}$ occurs and the runtime bound p' is not violated (which happens with non-negligible probability f) and he guessed correctly when the event $E_{\text{exp-guessing}}$ occurs (which happens with probability at least $\frac{1}{p_{\mathcal{E}}(\eta, |a|)}$, where $p_{\mathcal{E}}$ is the runtime bound of \mathcal{E}). In runs where A does not resign, he will always guess the correct bit in case of $b' = 1$, and will only be incorrect in case of $b' = 0$ if $c = a \cdot b$ for $c \stackrel{\$}{\leftarrow} \{1, \dots, n\}$. As we require $n \geq 2$, this happens with probability at most $\frac{1}{2}$, which gives the claim.

Because A violates the DDH assumption with its non-negligible advantage, we conclude that $E_{\text{exp-guessing}}$ is negligible. This concludes Step 2 as both $E_{\text{not-fresh}}$ and $E_{\text{exp-guessing}}$ are negligible.

Step 3: In this step, we replace real Diffie-Hellman keys with ideal ones, however, without preventing key collisions or key guessing for DH keys. To be more precise, let $\mathcal{P}_{\text{crypto}}^3$ be the system that works just as $\mathcal{P}_{\text{crypto}}^2$ except for creating DH keys, which is done as in $\mathcal{F}_{\text{crypto}}$. That is, whenever a new key is generated during a **GenDHKey** command, $\mathcal{P}_{\text{crypto}}^3$ asks the simulator to provide the actual value k for the DH key. The key k is then used without checking for key collisions or key guessing (this is done in a later step as resistance to key guessing also depends

on the security of the key derivation and symmetric encryption scheme). In the following, we will show that

$$\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^2\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^3\} \quad (5.3)$$

for all $\mathcal{E} \in \text{Env}(\{\mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^2\})$.

Note that by the definition of the simulator both systems behave identical when generating known DH keys, i.e., only the generation of unknown DH keys has actually been changed. We prove (5.3) via a hybrid argument where we replace real with ideal unknown DH keys in the order of their creation. For this we define hybrid systems \mathcal{H}_r , $r \in \mathbb{N}$, which behave just as $\mathcal{P}_{\text{crypto}}^3$ for unknown DH keys up to (and including) the r -th unknown DH key, and otherwise behave as $\mathcal{P}_{\text{crypto}}^2$. More specifically, \mathcal{H}_r keeps track of the order in which unknown DH keys are created, where the first unknown DH key has order 1 and so on. If an unknown DH key of order i is created and $i \leq r$, then the DH key is created as in $\mathcal{P}_{\text{crypto}}^3$, otherwise it is created as in $\mathcal{P}_{\text{crypto}}^2$. For brevity of presentation, we define the following combined system:

$$\mathcal{D}_r := \{\mathcal{S}, \mathcal{F}^* \parallel \mathcal{H}_r\}$$

Now let $\mathcal{E} \in \text{Env}(\{\mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^2\})$. Recall that we write $\mathcal{R} \equiv_f \mathcal{Q}$ if the difference in the probabilities for outputting 1 in runs of \mathcal{R} and \mathcal{Q} is bounded from above by a function f . It is trivial to see that there is a negligible function f_0 such that

$$\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^2\} \equiv_{f_0} \{\mathcal{E}, \mathcal{D}_0\} \quad (5.4)$$

as both systems behave in exactly the same way. Furthermore, as there is a polynomial $p_{\mathcal{E}}$ that bounds the runtime of \mathcal{E} in runs with any system, there will be at most $p_{\mathcal{E}}(\eta, |a|)$ DH keys generated in runs of $\mathcal{E} \mid \mathcal{S} \mid \mathcal{F}^* \mid \mathcal{P}_{\text{crypto}}^3$. Hence, there is also a negligible function $f_{p_{\mathcal{E}}}$ such that

$$\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^3\} \equiv_{f_{p_{\mathcal{E}}}} \{\mathcal{E}, \mathcal{D}_{p_{\mathcal{E}}(\eta, |a|)}\} \quad (5.5)$$

as both systems behave exactly the same if no more than $p_{\mathcal{E}}(\eta, |a|)$ DH keys are created.

We now show a lemma that says that the systems $\{\mathcal{E}, \mathcal{D}_r\}$ and $\{\mathcal{E}, \mathcal{D}_{r+1}\}$ are indistinguishable for $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$, where the negligible function that bounds the distinguishing advantage is independent of r . This will immediately imply (5.3).

Lemma 5.1. *There exists a negligible function f' such that for all $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ the following holds true:*

$$\{\mathcal{E}, \mathcal{D}_r\} \equiv_{f'} \{\mathcal{E}, \mathcal{D}_{r+1}\}$$

Proof. We start by showing that for every $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ there is a negligible function f'_r that bounds the distinguishing advantage. We then explain how the argument can be extended to obtain a single negligible function f' that bounds the advantage for all $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$.

First, observe that we can find a single polynomial q and a single negligible function f such

that for all $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ the runtime of $\{\mathcal{E}, \mathcal{D}_r\}$ is bounded by q except for a negligible set of runs, where f bounds the probability of these runs from above. This is because \mathcal{E} has the same runtime $p_{\mathcal{E}}$ in all runs with any system, so can use at most $p_{\mathcal{E}}$ commands. Thus, the runtime of \mathcal{D}_r can be bounded by $p_{\mathcal{E}} \cdot t + c$ where t is a worst case bound for any command and c bounds the runtime of the initialization. Note that c does not depend on r as initialization occurs before any keys can be created. This runtime bound is violated only if **GroupGen** does not run in polynomial time, which happens with at most negligible probability.

Let $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ and suppose by contradiction that there is no negligible function f'_r such that $|\Pr[\{\mathcal{E}, \mathcal{D}_r\} = 1] - \Pr[\{\mathcal{E}, \mathcal{D}_{r+1}\} = 1]| \leq f'_r$. Note that the systems \mathcal{D}_r and \mathcal{D}_{r+1} are almost identical in their behavior; the only difference is the handling of unknown DH keys of order $r + 1$, which are generated by calculating \mathbf{g}^{ab} in \mathcal{D}_r while \mathcal{D}_{r+1} uses $\mathbf{g}^c, c \xleftarrow{\$} \{1, \dots, \mathfrak{n}\}$. We can use this to construct an adversary A on the DDH assumption; but first, we need to introduce some terminology.

In the following, let $[\mathcal{E}, \mathcal{D}_r]_q$ denote the system that behaves just as $\{\mathcal{E}, \mathcal{D}_r\}$ but runs at most $q(\eta, |a|)$ steps and, if this bound is violated, stops with output 1. As this bound is reached with at most negligible probability, the above assumption implies that $|\Pr[[\mathcal{E}, \mathcal{D}_r]_q = 1] - \Pr[[\mathcal{E}, \mathcal{D}_{r+1}]_q = 1]|$ is also non-negligible. Furthermore, let $E_{\text{key-created}}$ be the event that in a run of $[\mathcal{E}, \mathcal{D}_r]_q$ or $[\mathcal{E}, \mathcal{D}_{r+1}]_q$ an unknown DH key of order $r + 1$ is created. As the systems \mathcal{D}_r and \mathcal{D}_{r+1} behave exactly the same until an unknown DH key of order $r + 1$ is created, the event $E_{\text{key-created}}$ has the same probability in both systems and an environment can use only runs from $E_{\text{key-created}}$ to distinguish these systems. That is, we have

$$\begin{aligned}
& |\Pr[[\mathcal{E}, \mathcal{D}_r]_q = 1] - \Pr[[\mathcal{E}, \mathcal{D}_{r+1}]_q = 1]| \\
&= |\Pr[(\mathcal{E}, \mathcal{D}_r]_q = 1) \wedge E_{\text{key-created}}] \\
&\quad - \Pr[(\mathcal{E}, \mathcal{D}_{r+1}]_q = 1) \wedge E_{\text{key-created}}]| \\
&= \Pr[E_{\text{key-created}}] \cdot |\Pr[(\mathcal{E}, \mathcal{D}_r]_q = 1) | E_{\text{key-created}}] \\
&\quad - \Pr[(\mathcal{E}, \mathcal{D}_{r+1}]_q = 1) | E_{\text{key-created}}]| \tag{5.6}
\end{aligned}$$

We can now define the adversary A on the DDH assumption. Recall that an adversary on the DDH experiment (cf. Appendix E.5) gets $(\mathbf{G}, \mathfrak{n}, \mathbf{g}, \mathbf{g}^a, \mathbf{g}^b, h)$ as part of his input where either $h = \mathbf{g}^{ab}$ (if $b' = 1$) or $h = \mathbf{g}^c$ (if $b' = 0$). The adversary A guesses which exponents will be used to create the unknown DH key of order $r + 1$. To be more precise, as \mathcal{E} will create at most $p_{\mathcal{E}}(\eta, |a|)$ exponents, A has to guess an order $i \in \{1, \dots, p_{\mathcal{E}}(\eta, |a|) - 1\}$ for the first exponent and an order $j \in \{i + 1, \dots, p_{\mathcal{E}}(\eta, |a|)\}$ for the second one. A then continues to simulate a run of $[\mathcal{E}, \mathcal{D}_r]_q$ with the following changes: Instead of executing **GroupGen**, it uses the group $(\mathbf{G}, \mathfrak{n}, \mathbf{g})$ from the DDH experiment. Instead of generating the i -th exponent as in \mathcal{D}_r , A uses \mathbf{g}^a from the DDH experiment. When the j -th exponent is generated, A uses \mathbf{g}^b from the DDH experiment. The unknown DH key of order $r + 1$ is replaced with h from the DDH experiment. If no unknown DH key of order $r + 1$ is generated, or it is not generated from the i -th and j -th

exponent, the adversary aborts the simulation and outputs 1. If the simulation finishes without A aborting, then A forwards the bit that is output by $[\mathcal{E}, \mathcal{D}_r]_q$ at the end of the run. It is easy to see that A runs in polynomial time as the runtime of $[\mathcal{E}, \mathcal{D}_r]_q$ is bounded by q .

Observe that this simulation is possible even without knowing the actual values of a and b : The checks on exponent collisions and exponent guessing can be performed by using \mathbf{g}^a and \mathbf{g}^b instead, as a collides with some other exponent $e \in \{1, \dots, n\}$ if and only if \mathbf{g}^a collides with \mathbf{g}^e (as g is a generator of G). If the environment tries to retrieve a or b before an unknown DH key of order $r + 1$ was created, then A aborts the run as one of the exponents gets marked known and cannot be used to create unknown keys anymore. If a or b are used to create an unknown DH key of order smaller than $r + 1$, then this is easy to simulate as such keys are created ideally. If a or b are used to create a known DH key before the $r + 1$ -st one, then either they become known or they are created with a DH share \mathbf{g}^e where $e \in \text{exp}$ and $e \neq a$ or $e \neq b$, respectively. In the former case, A aborts, while in the latter case, it can calculate $(\mathbf{g}^a)^e$ or $(\mathbf{g}^b)^e$, respectively, as it knows e . If A guessed correctly and \mathbf{g}^a and \mathbf{g}^b have been used to create an unknown DH key of order $r + 1$, then both a and b will stay unknown during the whole run as the environment does not cause the commitment problem. Thus, neither of them will be retrieved, and if \mathbf{g}^a or \mathbf{g}^b are used to create another key after the $r + 1$ -st one, it will be created with a DH share \mathbf{g}^e where $e \in \text{exp}$ and $e \neq a$ or $e \neq b$, respectively. As explained above, A can simulate key generation in this case as he knows e .

Let $E_{\text{no-abort}}$ be the event that A does not abort. Observe that this event does not depend on the bit b' from the DDH experiment as the simulation is independent of b' until an unknown DH key of order $r + 1$ has been created, at which point A will no longer abort. This implies that runs where A aborts do not influence $\text{Adv}_{A, \text{GroupGen}}^{\text{DDH}}$ at all. Furthermore, we have that $\Pr[E_{\text{no-abort}}] \geq p_{\mathcal{E}}^{-2} \cdot \Pr[E_{\text{key-created}}]$ as A will not abort if, in the run it simulates, an unknown DH key of order $r + 1$ is created and he guessed the exponents correctly (we use $p_{\mathcal{E}}^{-2}$ to bound the probability for a correct guess from below. This estimation can still be improved, but is sufficient for the proof).

In every run where A does not abort, it perfectly simulates one run of either $[\mathcal{E}, \mathcal{D}_r]_q$ (if $b' = 1$) or $[\mathcal{E}, \mathcal{D}_{r+1}]_q$ (if $b' = 0$) from $E_{\text{key-created}}$. Formally, we can establish a bijection from runs in $E_{\text{key-created}}$ to runs of $\text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-1} / \text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-0}$ where A does not abort as follows: Let α be a run from $E_{\text{key-created}}$. Then there are two *different* exponents \bar{a} and \bar{b} that have been used to create an unknown DH key of order $r + 1$; let \bar{i} be the position of the first and \bar{j} be the position of the second one. We map α to the run β of $\text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-1} / \text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-0}$ where $a = \bar{a}$, $b = \bar{b}$, $i = \bar{i}$, $j = \bar{j}$. It is easy to verify that β is a run of A where the adversary does not abort.

Note that we can define this bijection only because unknown DH keys are always created from two different exponents, i.e., $\mathcal{F}_{\text{crypto}}$ does not allow creating unknown keys by pairing some unknown exponent e with \mathbf{g}^e . If $\mathcal{F}_{\text{crypto}}$ did allow this, then there might be runs where $\bar{i} = \bar{j}$, which cannot occur in A . It is impossible to extend A to also cover this case, as A might have to calculate $(\mathbf{g}^a)^a$ without knowing a if $b' = 1$ and $i = \bar{i} = \bar{j} = j$.

Overall, we have that

$$\begin{aligned}
& \text{Adv}_{A, \text{GroupGen}}^{\text{DDH}} \\
&= \left| \Pr \left[\text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-1} = 1 \right] - \Pr \left[\text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-0} = 1 \right] \right| \\
&= \Pr [E_{\text{no-abort}}] \cdot \left| \Pr \left[(\text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-1} = 1) | E_{\text{no-abort}} \right] \right. \\
&\quad \left. - \Pr \left[(\text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-0} = 1) | E_{\text{no-abort}} \right] \right| \\
&= \Pr [E_{\text{no-abort}}] \cdot \left| \Pr \left[([\mathcal{E}, \mathcal{D}_r]_q = 1) | E_{\text{key-created}} \right] \right. \\
&\quad \left. - \Pr \left[([\mathcal{E}, \mathcal{D}_{r+1}]_q = 1) | E_{\text{key-created}} \right] \right| \\
&\geq p_{\mathcal{E}}^{-2} \cdot \Pr [E_{\text{key-created}}] \cdot \left| \Pr \left[([\mathcal{E}, \mathcal{D}_r]_q = 1) | E_{\text{key-created}} \right] \right. \\
&\quad \left. - \Pr \left[([\mathcal{E}, \mathcal{D}_{r+1}]_q = 1) | E_{\text{key-created}} \right] \right| \\
&\stackrel{(5.6)}{=} p_{\mathcal{E}}^{-2} \cdot \left| \Pr \left[[\mathcal{E}, \mathcal{D}_r]_q = 1 \right] - \Pr \left[[\mathcal{E}, \mathcal{D}_{r+1}]_q = 1 \right] \right|
\end{aligned}$$

which is non-negligible by assumption. As this breaks the DDH assumption, we conclude that there must be a negligible function f'_r such that $\{\mathcal{E}, \mathcal{D}_r\} \equiv_{f'_r} \{\mathcal{E}, \mathcal{D}_{r+1}\}$.

To receive a negligible bound f' that holds for all $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$, one uses the same argument with an adversary A that first guesses r and then simulates $[\mathcal{E}, \mathcal{D}_r]_q$. Note that here we need that q bounds the runtime of $\{\mathcal{E}, \mathcal{D}_r\}$ with the same negligible probability independently of r . \square

With Lemma 5.1 we can now conclude the proof of the third step. We have that

$$\begin{aligned}
& \left| \Pr \left[\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \mid \mathcal{P}_{\text{crypto}}^2\} = 1 \right] - \Pr \left[\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \mid \mathcal{P}_{\text{crypto}}^3\} = 1 \right] \right| \\
&\leq f_0 + f_{p_{\mathcal{E}}} + \left| \Pr \left[\{\mathcal{E}, \mathcal{D}_0\} = 1 \right] - \Pr \left[\{\mathcal{E}, \mathcal{D}_{p_{\mathcal{E}}(\eta, |a|)}\} = 1 \right] \right| \\
&\leq f_0 + f_{p_{\mathcal{E}}} + \sum_{r=0}^{p_{\mathcal{E}}(\eta, |a|)-1} \left| \Pr \left[\{\mathcal{E}, \mathcal{D}_r\} = 1 \right] - \Pr \left[\{\mathcal{E}, \mathcal{D}_{r+1}\} = 1 \right] \right| \\
&\leq f_0 + f_{p_{\mathcal{E}}} + p_{\mathcal{E}}(\eta, |a|) \cdot f'
\end{aligned}$$

where f_0 is the negligible function from (5.4), $f_{p_{\mathcal{E}}}$ is the negligible function from (5.5), $p_{\mathcal{E}}$ is the polynomial that bounds the runtime of \mathcal{E} , and f' is the negligible function from Lemma 5.1. As this is negligible, we conclude that $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \mid \mathcal{P}_{\text{crypto}}^2\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \mid \mathcal{P}_{\text{crypto}}^3\}$.

Step 4: In this step we replace real symmetric encryption/decryption and key derivation with their ideal versions; MACs are handled in the next step. Furthermore, we prevent key collisions for fresh unknown keys, and key guessing of unknown keys for known keys inserted by the environment/adversary. This step follows the same argument as the proof of the original $\mathcal{F}_{\text{crypto}}$ [97] but with the following major changes: (i) all hybrid systems have to be extended

to also include DH key handling, (ii) we have to show that real key derivation from DH keys is indistinguishable from ideal key derivation from DH keys, and (iii) we have to show that key collisions and key guessing happen only with negligible probability for DH keys. We will reuse the original notation from [97] and only sketch the proof where it remains essentially unchanged, such that the reader can refer to [97] for full details.

Let $\mathcal{F}'_{\text{crypto}}$ be the machine that behaves exactly as $\mathcal{F}_{\text{crypto}}$ except for creating and verifying MACs, which is handled as in $\mathcal{P}_{\text{crypto}}$. In this step we prove that

$$\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^3\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}'_{\text{crypto}}\}$$

for all $\mathcal{E} \in \text{Env}(\{\mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^3\})$. This is done via a hybrid argument where one replaces real with ideal key handling in the order in which unknown symmetric keys of type $t \neq \text{mac-key}$ are used for the first time. Just as in [97], in our proof we will not explicitly deal with keys of type **unauthenc-key** for simplicity of presentation. The proof for keys of this type is analogous to the one for keys of type **authenc-key** but involves some additional technicalities, namely in cases where an unknown key of type **unauthenc-key** is used for decryption and then later on becomes known (this is not prevented by \mathcal{F}^* as it is not a commitment problem: no ideal encryption has been performed with this key). We refer the reader to [97] and [95] for details on how to deal with keys of type **unauthenc-key**; the same arguments also apply here.

For $r \in \mathbb{N}$ let $\mathcal{F}_{\text{crypto}}^{(r)}$ be the hybrid machine that behaves as $\mathcal{F}'_{\text{crypto}}$ but additionally keeps track of the order in which unknown keys are used for the first time to encrypt a message or derive a key. That is, $\mathcal{F}_{\text{crypto}}^{(r)}$ maintains a variable `nextused` (initially set to 1) that stores the order of the next unknown key that is used for the first time. Furthermore, $\mathcal{F}_{\text{crypto}}^{(r)}$ has a partial function `used` that maps unknown keys to their order, or \perp if they have not been used yet. If an unknown key k of order $i < r$ is used, then $\mathcal{F}_{\text{crypto}}^{(r)}$ performs the operation ideally, i.e., as in $\mathcal{F}'_{\text{crypto}}$, while operations with keys of order $i \geq r$ are performed as in $\mathcal{P}_{\text{crypto}}^3$. Key collisions in $\mathcal{F}_{\text{crypto}}^{(r)}$ are prevented for unknown keys provided by the simulator (but not for those generated directly by $\mathcal{F}_{\text{crypto}}^{(r)}$, such as those generated during a key derivation operation from an unknown key of order $\geq r$, which $\mathcal{F}_{\text{crypto}}^{(r)}$ performs non-ideally). Additionally, key guessing is relaxed as follows in $\mathcal{F}_{\text{crypto}}^{(r)}$: If `nextused` $\leq r$, then key guessing is prevented for all unknown keys. If `nextused` $> r$ (i.e., at least one operation with an unknown key has been performed non-ideally), then key guessing is prevented only for unknown keys of order $j \leq r$.

Next, we also need an oracle protocol $\mathcal{O}_b = (\text{oracle})$ that is parameterized with $b \in \{\text{real}, \text{ideal}\}$. This oracle models usage of a single symmetric key of type $t \neq \text{mac-key}$ either in a real or ideal way, depending on b , and will be used as a subroutine in our proof. Compared to the version of this oracle used in [97], our oracle has to be extended to also deal with keys of type **dh-key**. More formally, \mathcal{O}_b is a single machine (with one role) that accepts all entities and hence has at most one instance. It does not include any corruption modeling, i.e., sets the corruption model in the template to `custom`. Upon its first activation, it runs `GroupGen` and saves the result. A higher-level protocol can ask \mathcal{O}_b at any time for the generated group. Except

for this request, the oracle does not allow higher-level protocols to perform any tasks until the higher-level protocol has sent a special message (Init, t) to \mathcal{O}_b that contains a single key type $t \neq \text{mac-key}$. Upon receiving this key type t , \mathcal{O} either generates a key $k = g^c, c \xleftarrow{\$} \{1, \dots, n\}$ if $t = \text{dh-key}$, or $k \xleftarrow{\$} \{0, 1\}^\eta$ otherwise. Then, \mathcal{O}_b provides the higher-level protocol with an interface to use the generated key to perform real (i.e., as in $\mathcal{P}_{\text{crypto}}$) or ideal operations (i.e., as in $\mathcal{F}_{\text{crypto}}$, except that encryption/decryption does not include any pointer replacement for plaintexts) with that key, depending on b . The following lemma states that no environment can distinguish the real and ideal oracle because the underlying primitives are secure:

Lemma 5.2. *Let $\mathcal{O}_{\text{real}}$ and $\mathcal{O}_{\text{ideal}}$ be as above. Then it holds true that*

$$\{\mathcal{E}, \mathcal{O}_{\text{real}}\} \equiv \{\mathcal{E}, \mathcal{O}_{\text{ideal}}\}$$

for all $\mathcal{E} \in \text{Env}(\mathcal{O}_{\text{real}})$.

Proof. This lemma is easy to prove with standard reduction techniques as all primitives are secure in their respective security games. In [97] this lemma was already proven for encryption and key derivation from keys of type `pre-key`. The proof for key derivation from DH keys is the same as for key derivation from keys of type `pre-key`. Note in particular that DH keys in \mathcal{O}_b and the security experiment for PRFs based on DH keys (see Appendix E.6) are sampled in the same way, which allows for a reduction from one setting to the other. \square

Finally, we define the hybrid system $\hat{\mathcal{F}}_{\text{crypto}}^{(r)}$ that works as $\mathcal{F}_{\text{crypto}}^{(r)}$ but uses \mathcal{O}_b as a private subroutine protocol. Instead of running `GroupGen` itself, $\hat{\mathcal{F}}_{\text{crypto}}^{(r)}$ requests the group from \mathcal{O}_b . Furthermore, $\hat{\mathcal{F}}_{\text{crypto}}^{(r)}$ uses \mathcal{O}_b to handle operations performed with unknown keys of order r (i.e., the first key for which operations are performed real in $\mathcal{F}_{\text{crypto}}^{(r)}$). That is, $\hat{\mathcal{F}}_{\text{crypto}}^{(r)}$ still internally maintains a key k of order r , which might be encrypted by other keys as part of a plain text; however, if this key is used at some point, $\hat{\mathcal{F}}_{\text{crypto}}^{(r)}$ relays the call to the subroutine \mathcal{O}_b instead and forwards the output (if \mathcal{O}_b has not been initialized with a key type yet, then this is done first. Also, $\mathcal{F}_{\text{crypto}}$ still performs replacement operations for pointers in plaintexts before using \mathcal{O}_b to encrypt the resulting plaintext; similarly for decryption).

Now let $\mathcal{E} \in \text{Env}(\{\mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^3\})$. Let $p_{\mathcal{E}}$ be a polynomial such that the runtime of \mathcal{E} (in any run with any system) is bounded by $p_{\mathcal{E}}(\eta, |a|) - 1$; such a polynomial exists as \mathcal{E} is an environment. For brevity, we define the following combined systems for $r \in N$ and $b \in \{\text{real}, \text{ideal}\}$:

$$\begin{aligned} \mathcal{C}^{(r)} &:= \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}_{\text{crypto}}^{(r)}\} \\ \hat{\mathcal{C}}_b^{(r)} &:= \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \hat{\mathcal{F}}_{\text{crypto}}^{(r)} \parallel \mathcal{O}_b\} \end{aligned}$$

We first show an auxiliary lemma that allows us to ignore key collisions in our proofs. More formally, let $B_{\text{coll}}^{(r)}$ be the event that in a run of $\mathcal{C}^{(r)}$ the simulator \mathcal{S} generates an unknown symmetric key (of any type) such that $\mathcal{F}_{\text{crypto}}^{(r)}$ rejects the key as it collides with some other

existing key in keys. The following lemma states that this happens with negligible probability where the probability is independent of r .

Lemma 5.3. *There exists a single negligible function f_{coll} such that for all $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ it holds true that*

$$\Pr \left[B_{\text{coll}}^{(r)}(1^\eta) \right] \leq f_{\text{coll}}(1^\eta)$$

Proof. For symmetric keys of type $t \neq \text{dh-key}$, this directly follows from the fact that the simulator chooses fresh unknown keys uniformly at random from $\{0, 1\}^\eta$ and the fact that there are at most $p_{\mathcal{E}}(\eta, |a|)$ many keys during a run of any of those systems. Note that both properties are independent of r .

For keys of type $t = \text{dh-key}$, observe that such a key is generated by choosing a random exponent $c \xleftarrow{\$} \{1, \dots, n\}$ independently of the other keys (which are group elements themselves) that are currently stored in $\mathcal{F}_{\text{crypto}}^{(r)}$. This is the same setting as in Step 2, where we have shown that, when randomly and independently choosing polynomially many exponents e_i , there is only a negligible chance that any of the group elements \mathbf{g}^{e_i} collides with one of polynomially many existing group elements.

Thus, by the same argument as in step 2, we have that collisions of fresh unknown keys of type dh-key also happen with negligible probability if the DDH assumption holds true and groups always have size $n \geq 2$. Note in particular that the probability of a collision only depends on the size of the group n , the number of group elements contained in $\mathcal{F}_{\text{crypto}}^{(r)}$, and the number of exponents generated by $\mathcal{F}_{\text{crypto}}^{(r)}$, where the latter two are upper bounded by the polynomial $p_{\mathcal{E}}$. As all of these properties are independent of r , we have that the negligible bound of $B_{\text{coll}}^{(r)}(1^\eta)$ for keys of type $t = \text{dh-key}$ is also independent of r .

By combining the negligible upper bound for keys of type $t \neq \text{dh-key}$ with the negligible upper bound for keys of type $t = \text{dh-key}$ we obtain the claim. \square

We can use Lemma 5.3 to show that $\mathcal{C}^{(0)}$ is indistinguishable from $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^3\}$ and that $\mathcal{C}^{(p_{\mathcal{E}}(\eta, |a|))}$ is indistinguishable from $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}'_{\text{crypto}}\}$.

Lemma 5.4. *There exist negligible functions f_0 and $f_{p_{\mathcal{E}}}$ such that:*

$$\begin{aligned} \mathcal{C}^{(0)} &\equiv_{f_0} \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^3\} \\ \mathcal{C}^{(p_{\mathcal{E}}(\eta, |a|))} &\equiv_{f_{p_{\mathcal{E}}}} \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}'_{\text{crypto}}\} \end{aligned}$$

Proof. By definition of $\mathcal{C}^{(0)}$, all cryptographic operations with symmetric keys are performed as in the real world, i.e., as in $\mathcal{P}_{\text{crypto}}^3$. Hence, the only difference is that $\mathcal{C}^{(0)}$ prevents collisions of fresh unknown keys from the simulator with other existing keys, so the distinguishing advantage of any environment \mathcal{E} is upper bounded by f_{coll} . As f_{coll} is negligible by Lemma 5.3, we obtain the first equation.

For the second equation, observe that $\mathcal{C}^{(p_{\mathcal{E}}(\eta, |a|))}$ behaves identical to $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}'_{\text{crypto}}\}$ until an unknown key of order $p_{\mathcal{E}}(\eta, |a|)$ is used for the first time. Since the environment has runtime

at most $p_{\mathcal{E}}(\eta, |a|) - 1$, there will be at most $p_{\mathcal{E}}(\eta, |a|) - 1$ unknown keys. Hence, $\mathcal{C}^{(p_{\mathcal{E}}(\eta, |a|))}$ always behaves just as $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}'_{\text{crypto}}\}$, which gives the second equation. \square

We still have to show that the r -th hybrid is indistinguishable from the $(r + 1)$ -th hybrid. The next lemma is used for this as it allows us to consider $\hat{\mathcal{C}}_{\text{real}}^{(r)}$ and $\hat{\mathcal{C}}_{\text{ideal}}^{(r)}$ instead of $\mathcal{C}^{(r)}$ and $\mathcal{C}^{(r+1)}$.

Lemma 5.5. *There exist negligible functions f_{real} and f_{ideal} such that for $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$*

$$\mathcal{C}^{(r)} \equiv_{f_{\text{real}}} \hat{\mathcal{C}}_{\text{real}}^{(r)} \quad (5.7)$$

$$\mathcal{C}^{(r+1)} \equiv_{f_{\text{ideal}}} \hat{\mathcal{C}}_{\text{ideal}}^{(r)} \quad (5.8)$$

Proof. We show (5.7), the proof of (5.8) is analogous.

Observe that the systems $\mathcal{C}^{(r)}$ and $\hat{\mathcal{C}}_{\text{real}}^{(r)}$ are already very similar, with the only difference being the handling of the r -th key, i.e., the unknown key of order r , which is relayed to $\mathcal{O}_{\text{real}}$ in $\hat{\mathcal{C}}_{\text{real}}^{(r)}$. Note that the distribution of the r -th key in $\mathcal{C}^{(r)}$ is the same as the one of keys in $\mathcal{O}_{\text{real}}$: Even if the key was derived, then it was derived ideally (as it must have been derived from a key of order $< r$). Also note that the r -th key was always encrypted ideally (as the environment is used order respecting by the definition of \mathcal{F}^*) with a leakage function that leaks exactly the length, and that the r -th key in $\mathcal{F}'_{\text{crypto}}^{(r)}$ has the same length as the key in $\mathcal{O}_{\text{real}}$. Here we need our assumption on **GroupGen** that all group elements of a given group are represented using a bit string of the same length (that might depend on η and \mathbb{G}). By this, it does not make a difference that $\mathcal{F}'_{\text{crypto}}^{(r)}$ in $\hat{\mathcal{C}}_{\text{real}}^{(r)}$ actually encrypts a second independent bit string instead of the key that is contained in and used by $\mathcal{O}_{\text{real}}$. Hence, the only way to distinguish $\mathcal{C}^{(r)}$ and $\hat{\mathcal{C}}_{\text{real}}^{(r)}$ is when either a key collision or a fresh key occurs (as $\mathcal{F}'_{\text{crypto}}^{(r)}$ in $\mathcal{C}^{(r)}$ ensures that the unknown key of order r is fresh when it is generated by the simulator, while the key used in $\mathcal{O}_{\text{real}}$ may collide with some other existing key in $\hat{\mathcal{F}}_{\text{crypto}}^{(r)}$) or the environment can guess the key (as $\mathcal{F}'_{\text{crypto}}^{(r)}$ in $\mathcal{C}^{(r)}$ prevents guessing of the r -th key, but $\hat{\mathcal{F}}_{\text{crypto}}^{(r)}$ in $\hat{\mathcal{C}}_{\text{real}}^{(r)}$ cannot do this for the key in $\mathcal{O}_{\text{real}}$). See [97] for a formal mapping of runs from $\mathcal{C}^{(r)}$ to $\hat{\mathcal{C}}_{\text{real}}^{(r)}$ where neither a key collision or key guessing occurred.

As we have already shown that key collisions in general happen with negligible probability (by Lemma 5.3), we only have to show that this also holds for key guessing of the r -th unknown key. More specifically, let the event $B_{\text{guess}}^{(r)}$ be the set of all runs of $\mathcal{C}^{(r)}$ where the r -th key is guessed, i.e., after the r -th key has been created, the environment or the simulator tries to insert the r -th key as a known key (e.g., via the **Store** command or when asked to provide the value for a corrupted PSK). In [97], it was shown that $B_{\text{guess}}^{(r)}$ can be bounded by a negligible function that is independent of r for keys of type $t \neq \text{dh-key}$; the same argument still applies here. We will now prove the same result for keys of type $t = \text{dh-key}$.

We will do so by reducing the probability for key guessing of an unknown key of type $t = \text{dh-key}$ with order r to the security of the PRF family F' . Observe that, just as in the proof of Lemma 5.1, we can find a single polynomial q and a single negligible function f such that

for all $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ the runtime of $\mathcal{C}^{(r)}$ is bounded by q except for a negligible set of runs, where f bounds the probability of these runs from above. Hence, by the same argument as in Lemma 5.1, we only need to show how to construct an adversary on the PRF family F' for a single r (and thus finding a negligible function that bounds $B_{\text{guess}}^{(r)}$ only for that specific r). One can obtain a universal negligible bound that is independent of r by using an adversary that first guesses r and then proceeds in the same way.

Now let $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$. Suppose that $B_{\text{guess}}^{(r)}$ contains a non-negligible set of runs where the r -th key is of type **dh-key**. As $B_{\text{coll}}^{(r)}$ is negligible, this implies that $B_{\text{guess}}^{(r)} \setminus B_{\text{coll}}^{(r)}$ must also contain a non-negligible set of runs where the r -th key is of type **dh-key**. Let $B_{\text{dh-key-guess}}^{(r)}$ denote this set of runs in the following. We use this to construct an adversary A on the security of the PRF F' (cf. Appendix E.6).

Recall that A gets an oracle $O(\cdot)$ that allows the adversary to perform either real key derivation (if $b = 1$) or ideal key derivation (if $b = 0$), and a description (G, n, g) of a group generated via **GroupGen**. The adversary A first guesses two positions $1 \leq i, j \leq p_{\mathcal{E}}(\eta, |a|)$: The number i marks the i -th known key inserted by the environment and is used to guess which of the inserted keys will collide with an existing unknown key. The number j marks the j -th unknown key of type **dh-key** created by a **GenDHKey** request and is used to guess which DH key will be used as unknown key of order r . The adversary then internally simulates $\mathcal{C}^{(r)}$, except that A does not prevent any key collisions and does not prevent key guessing for the j -th unknown key of type **dh-key**. Instead of simulating **GroupGen** in $\mathcal{F}_{\text{crypto}}^{(r)}$, A uses the group (G, n, g) obtained from the experiment. If the unknown key of order r in the simulation is the j -th unknown key of type **dh-key**, then A uses $O(s)$ when asked to derive a key from salt s and the r -th unknown key. Otherwise, A derives keys just as in $\mathcal{C}^{(r)}$. At the end of the run, A chooses a fresh salt s' that has never been queried to the oracle O before, calculates $k_{\text{real}} := F'_{\eta}(k_i, s)$ (if k_i is not defined or not in the range of F'_{η} , then g is chosen as input key instead), queries $k_{\text{oracle}} := O(s)$, and outputs 1 iff $k_{\text{real}} = k_{\text{oracle}}$; otherwise, A outputs 0. Note that A is a polynomial time algorithm as the runtime of $\mathcal{C}^{(r)}$ exceeds q only if **GroupGen** does not run in polynomial time, however, A does not simulate **GroupGen**.

For brevity, we define E_{real} to be the event that A outputs 1 when running in the real experiment (i.e., $b = 1$) and E_{ideal} to be the event that A outputs 1 when running in the ideal experiment (i.e., $b = 0$), i.e.,

$$\text{Adv}_{A, F', \text{GroupGen}}^{\text{G-PRF}} = | \Pr [E_{\text{real}}] - \Pr [E_{\text{ideal}}] |$$

We first calculate $\Pr [E_{\text{ideal}}]$. Recall that in the ideal world, $O(\cdot)$ calculates the key k_{oracle} for a fresh salt s by sampling it uniformly at random from $\{0, 1\}^{\eta}$. Because this is independent of the calculation of k_{real} , the probability of $k_{\text{real}} = k_{\text{oracle}}$ is $2^{-\eta}$, no matter how k_{real} is distributed. As the adversary outputs 1 only if $k_{\text{real}} = k_{\text{oracle}}$, we have

$$\Pr [E_{\text{ideal}}] = 2^{-\eta}$$

For $\Pr[E_{\text{real}}]$, let e be a run from $B_{\text{dh-key-guess}}^{(r)}$. In the run e there is some unknown key k of type **dh-key**, say the \bar{j} -th unknown key that was created of this type, that is used as unknown key of order r . Furthermore, the environment or simulator will at some point try to insert k into $\mathcal{C}^{(r)}$ as a known key, say, as the \bar{i} -th known key (where \bar{i} is minimal in case this occurs multiple times in the run e). Now observe that, if A guesses $i = \bar{i}$ and $j = \bar{j}$ correctly, then A perfectly simulates the run e up to the point when the environment inserts the \bar{i} -th known key. This is because of the following: Observe that no collisions of fresh unknown keys occur in e by requirement, and hence it is no problem that A does not check for collisions. Also note that A guessed correctly that the $j = \bar{j}$ -th key of type **dh-key** is used as the r -th key, and key guessing of the r -th unknown key does not occur until the \bar{i} -th known key is inserted. Thus even without preventing key guessing for the j -th key of type **dh-key**, the run e is still simulated perfectly up to the insertion of the \bar{i} -th unknown key. Furthermore, observe that the view of the environment in the simulation (before the key guessing occurs) is exactly the same as in a run of $\mathcal{C}^{(r)}$ even though key derivation for the r -th key is handled by the oracle O of A . This is because the r -th key is always encrypted ideally as the environment is used-order respecting (and hence the resulting ciphertext does not depend on the actual value of the r -th key), and as soon as a single key is derived from the r -th key, it will no longer become known as the environment does not cause the commitment problem.

As A simulates such a run e perfectly up to the point of the key guessing if it guessed i and j correctly (which happens with probability $p_{\mathcal{E}}(\eta, |a|)^{-2}$), we have that in this case k_i equals the secret key of the experiment. Thus the check $k_{\text{real}} = k_{\text{oracle}}$ will always be true and A will output 1. This gives

$$\Pr[E_{\text{real}}] \geq p_{\mathcal{E}}(\eta, |a|)^{-2} \cdot \Pr\left[B_{\text{dh-key-guess}}^{(r)}\right]$$

Overall, we have

$$\begin{aligned} & \text{Adv}_{A, F, \text{GroupGen}}^{\text{G-PRF}} \\ &= \Pr[E_{\text{real}}] - \Pr[E_{\text{ideal}}] \\ &\geq p_{\mathcal{E}}(\eta, |a|)^{-2} \cdot \Pr\left[B_{\text{dh-key-guess}}^{(r)}\right] - 2^{-\eta} \end{aligned}$$

which is non-negligible by assumption. This violates the security of the PRF F' , so we conclude that there is a negligible function f_r that bounds $\Pr\left[B_{\text{guess}}^{(r)}\right]$. As explained above, by using an adversary that first guesses $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$, we obtain a negligible bound for $B_{\text{guess}}^{(r)}$ that is independent of r .

As both $B_{\text{coll}}^{(r)}$ and $B_{\text{guess}}^{(r)}$ are bounded by negligible functions independent of r , and the systems $\mathcal{C}^{(r)}$ and $\hat{\mathcal{C}}_{\text{real}}^{(r)}$ only differ if one of these events occurs, we have that there exists f_{real} such that for all $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$:

$$\mathcal{C}^{(r)} \equiv_{f_{\text{real}}} \hat{\mathcal{C}}_{\text{real}}^{(r)}$$

This concludes the proof of Lemma 5.5. □

We need just one more lemma to show Step 4. This lemma states that there also is a single negligible function that bounds the distinguishing probability of the systems $\hat{\mathcal{C}}_{\text{real}}^{(r)}$ and $\hat{\mathcal{C}}_{\text{ideal}}^{(r)}$.

Lemma 5.6. *There exists a negligible function f' such that for all $0 \leq r \leq p_{\mathcal{E}}(\eta, |a|)$ the following holds true:*

$$\hat{\mathcal{C}}_{\text{real}}^{(r)} \equiv_{f'} \hat{\mathcal{C}}_{\text{ideal}}^{(r)}$$

Proof. Observe that the system \mathcal{E}^{\S} that first chooses $r \xleftarrow{\S} \{0, \dots, p_{\mathcal{E}}(\eta, |a|)\}$ and then simulates the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \hat{\mathcal{F}}_{\text{crypto}}^{(r)}\}$ is a responsive environment for $\mathcal{O}_{\text{real}}$ and $\mathcal{O}_{\text{ideal}}$, respectively. This is because those oracles do not send any restricting messages, not even for obtaining an initial corruption status of an entity (recall that those oracles do not include any corruption modeling). Further note that, by the same reasoning as in the proof of Lemma 5.5, there exists a polynomial q that bounds the runtime of $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \hat{\mathcal{F}}_{\text{crypto}}^{(r)}\}$ in all runs for all $r \in \{0, \dots, p_{\mathcal{E}}(\eta, |a|)\}$, i.e., \mathcal{E}^{\S} can indeed perfectly perform the simulation while being environmentally bounded.

Hence, by Lemma 5.2, we have that there is f_{oracle} such that

$$\mathcal{E}^{\S} \mid \mathcal{O}_{\text{real}} \equiv_{f_{\text{oracle}}} \mathcal{E}^{\S} \mid \mathcal{O}_{\text{ideal}}$$

As \mathcal{E}^{\S} perfectly simulates the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \hat{\mathcal{F}}_{\text{crypto}}^{(r)}\}$ with probability $\frac{1}{(p_{\mathcal{E}}(\eta, |a|)+1)}$, we have for the negligible function $f' = (p_{\mathcal{E}}(\eta, |a|) + 1) \cdot f_{\text{oracle}}$:

$$\hat{\mathcal{C}}_{\text{real}}^{(r)} \equiv_{f'} \hat{\mathcal{C}}_{\text{ideal}}^{(r)}$$

This concludes the proof of Lemma 5.6. □

We can now conclude the proof of Step 4. Let $f_0, f_{p_{\mathcal{E}}}$ be the negligible functions from Lemma 5.4, let $f_{\text{real}}, f_{\text{ideal}}$ be the negligible functions from Lemma 5.5, and let f' be the negligible function from Lemma 5.6. We have that

$$\begin{aligned} & \left| \Pr \left[\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^3\} = 1 \right] - \Pr \left[\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}'_{\text{crypto}}\} = 1 \right] \right| \\ & \leq f_0 + f_{p_{\mathcal{E}}} + \left| \Pr \left[\mathcal{C}^{(0)} = 1 \right] - \Pr \left[\mathcal{C}^{(p_{\mathcal{E}})} = 1 \right] \right| \\ & \leq f_0 + f_{p_{\mathcal{E}}} + \sum_{r=0}^{p_{\mathcal{E}}-1} \left| \Pr \left[\mathcal{C}^{(r)} = 1 \right] - \Pr \left[\mathcal{C}^{(r+1)} = 1 \right] \right| \\ & \leq f_0 + f_{p_{\mathcal{E}}} + p_{\mathcal{E}} \cdot (f_{\text{real}} + f_{\text{ideal}}) \\ & \quad + \sum_{r=0}^{p_{\mathcal{E}}-1} \left| \Pr \left[\hat{\mathcal{C}}_{\text{real}}^{(r)} = 1 \right] - \Pr \left[\hat{\mathcal{C}}_{\text{ideal}}^{(r)} = 1 \right] \right| \\ & \leq f_0 + f_{p_{\mathcal{E}}} + p_{\mathcal{E}} \cdot (f_{\text{real}} + f_{\text{ideal}} + f') \end{aligned}$$

Because $f_0 + f_{p_\mathcal{E}} + p_\mathcal{E} \cdot (f_{\text{real}} + f_{\text{ideal}} + f')$ is negligible, we have that

$$\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}^3\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}'_{\text{crypto}}\}$$

This concludes the proof of Step 4.

Step 5: In this step we replace real MACs with their ideal versions. That is, we show that

$$\{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}'_{\text{crypto}}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}_{\text{crypto}}\}$$

for all $\mathcal{E} \in \text{Env}(\{\mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}'_{\text{crypto}}\})$. The proof from [97] still applies as our extension did not change how MACs are computed and verified.

Final step: We now combine the results from Steps 1 to 5. By Lemma 3.1, we have that the set of responsive environments for a system \mathcal{Q} is the same as the set of responsive environments for a system \mathcal{Q}' if no responsive environment can distinguish \mathcal{Q} and \mathcal{Q}' . Using this lemma, the results from Steps 1 to 5, and transitivity of the \equiv relation, we conclude that for all $\mathcal{E} \in \text{Env}(\mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}})$:

$$\{\mathcal{E}, \mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}_{\text{crypto}}\}$$

We still have to show that \mathcal{S} is a responsive simulator. By Lemma 3.7 it is sufficient to show that with overwhelming probability \mathcal{S} sends an expected answer either immediately or after sending some restricting messages to the environment. Observe that \mathcal{S} violates this behavior only if it resigns the simulation (e.g., after a key was rejected by $\mathcal{F}_{\text{crypto}}$). However, this happens with at most negligible probability as \mathcal{E} can observe a resignation of the simulator and use this to distinguish $\mathcal{F}^* \parallel \mathcal{P}_{\text{crypto}}$ and $\{\mathcal{S}, \mathcal{F}^* \parallel \mathcal{F}_{\text{crypto}}\}$. This implies that \mathcal{S} is responsive and therefore concludes the proof of Theorem 5.1. \square

5.2. Ideal Functionalities for Key Exchange With Key Usability

In this section we present our ideal functionalities for secure key exchanges: one functionality for mutual authentication, denoted by $\mathcal{F}_{\text{key-use}}^{\text{MA}}$, and one for unilateral authentication, denoted by $\mathcal{F}_{\text{key-use}}^{\text{UA}}$. These functionalities are of general interest and should be widely applicable. In Section 5.3, we use them to define security of and then analyze several authenticated Diffie-Hellman key exchange protocols from practice. In the following, we first present $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ and then describe how $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ differs.

The functionality $\mathcal{F}_{\text{key-use}}^{\text{MA}}$. On a high-level, similar to other key exchange functionalities (e.g., [49, 96]), the ideal functionality $\mathcal{F}_{\text{key-use}}^{\text{MA}} = (\text{initiator}, \text{responder})$ guarantees that an uncorrupted entity (modeling a key exchange user) that outputs a session key is in a session with an entity belonging to his intended communication partner, i.e., with the expected party ID, such that entities from the same session have complementary roles and only uncorrupted entities from the same session have access to the session key.

Our key exchange functionality is inspired by an ideal key exchange functionality from [96] but has important differences, which among others makes it more widely applicable. In particular, neither unilateral authentication nor perfect forward secrecy were considered in [96]. We discuss the differences compared to [96] in more detail at the end of this section. Compared to the most commonly used key exchange functionalities from or based on [49], $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ (and also $\mathcal{F}_{\text{key-use}}^{\text{UA}}$) has several distinguishing features, as explained next.

First, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ never directly outputs session keys to users from higher-level protocols. Instead it provides a user with a pointer and allows the user to perform ideal cryptographic operations with it (among others, symmetric encryption, MACing, and deriving new keys from the session key which in turn can then be also used with those primitives). This is an important feature as higher-level protocols that use $\mathcal{F}_{\text{key-use}}^{\text{MA}}$, such as secure channel protocols, can use the session key still in an ideal way, which simplifies the analysis of higher-level protocols and avoids reduction proofs.

Second, unlike most other formulations of key exchange functionalities in the literature, the above feature also makes it possible to realize $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ by key exchange protocols that use the session key during the key exchange. Most key exchange functionalities simply output a session key that was chosen uniformly at random, and thus, a realization must ensure that the session key is indistinguishable from a random one. However, this is not the case if the key was used during the actual key exchange, e.g. to encrypt a message, as then the environment can check whether the key that is output after a successful key exchange can be used decrypt said message. In contrast to functionalities based on session key indistinguishability, our functionality does not output the session key but only gives access to idealized cryptographic operations. As long as a key exchange protocol ensures separate domains of messages that are, e.g., encrypted with the session key during and after the key establishment phase, it can realize $\mathcal{F}_{\text{key-use}}^{\text{MA}}$. We emphasize that, while $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ technically provides a slightly weaker statement than full session

key indistinguishability, the security guarantees of $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ are sufficient for all practical purposes as the session key can be used for idealized cryptographic primitives.⁴⁷

Third, almost all formulations of functionalities (including key exchange functionalities) in the universal composability literature assume the existence of pre-established SIDs to define sessions of a key exchange (cf. Section 4.4). Therefore, they can only be realized by protocols where all participants of a session have already agreed on some shared SID outside of the protocol. However, many protocols from practice, such as the key exchanges that we analyze in Section 5.3, do not explicitly establish such a global SID before starting a key exchange; instead, they rather form sessions implicitly during the actual key exchange depending on how messages are routed by the network. To be able to also capture such protocols faithfully, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ does not rely on pre-established session IDs. Instead, it uses local session IDs that are chosen and managed by the higher-level instances. These local SIDs do not impact in any way which entities may form a key exchange session; their only purpose is to allow some party in a higher-level protocol to run and (locally) address multiple instances of the same key exchange protocol. Local sessions (of an initiator and a responder) are then paired into global sessions by the adversary/simulator, subject to certain restrictions that, e.g., guarantee authentication of parties. Uncorrupted entities belonging to the same global session then obtain pointers to an ideally generated session key. Observe that this mechanism captures the expected behavior of secure real world key exchanges in a natural way: at the start of such a protocol, it is typically not yet clear which instance of a key exchange of some party will later on derive the same session key as another instance of another party. However, we expect from a secure key exchange protocol that, once a protocol instance i of some party finishes, there is a uniquely defined protocol instance j of the expected partner such that i and j have performed the key exchange with each other.

Formally, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ consists of two roles **initiator** and **responder** that use $\mathcal{F}_{\text{crypto}}$ as a subroutine to implement idealized cryptographic operations using session keys (cf. Figure 5.1). Both roles are implemented by a single machine $M_{\text{initiator,responder}}$ that accepts all entities in a single instance, where such an entity $(pid, sid, role)$ of $M_{\text{initiator,responder}}$ models a key exchange user, i.e., the key exchange instance of party pid that is addressed using the locally chosen SID sid and runs in role $role \in \{\text{initiator}, \text{responder}\}$. $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ is parameterized with a symmetric key type $t_{\text{key}} \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}$ which determines the type of the keys that are output after a successful key exchange. The functionality $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ maintains a mapping $\text{userStatus} : (\{0, 1\}^*)^3 \rightarrow \{\perp, \text{started}, \text{inSession}, \text{exchangeFinished}, \text{sessionClosed}, \text{corrupted}\}$, initially set to \perp for every input, which stores the current status

⁴⁷In game-based security definitions for key exchanges there exists a similar situation: On the one hand, there are security models that require session key indistinguishability, such as the Bellare-Rogaway model and the CK model [15, 47], and on the other hand there is the ACCE model [79] and variations thereof. Both types of models define security for key exchanges, however, the ACCE model provides a slightly relaxed security statement that, intuitively, “only” requires cryptographic operations performed with the session key to be secure. This is considered to be sufficient in practice. In fact, the stronger notion of session key indistinguishability is typically just used as a tool to show the security of cryptographic primitives keyed with a session key anyway.

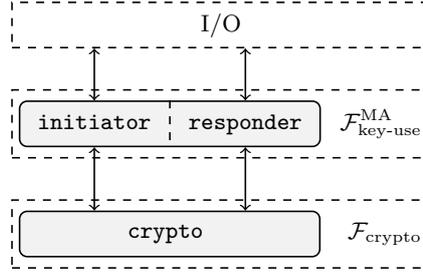


Figure 5.1.: The static structure of the ideal key exchange functionality with mutual authentication $\mathcal{F}_{\text{key-use}}^{\text{MA}}$. The static structure of the ideal key exchange functionality with unilateral authentication $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ is analogous.

of key exchange users. The functionality also stores the PID of the intended partner of a key exchange user $(pid, sid, role)$ via a mapping $\text{intendedPartner} : (\{0, 1\}^*)^3 \rightarrow \{0, 1\}^*$. Finally, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ keeps track of global key exchange sessions via a set $\text{sessions} \subseteq (\{0, 1\}^*)^3 \times (\{0, 1\}^*)^3$ containing pairs of entities; for each such pair there is a unique session key in $\mathcal{F}_{\text{crypto}}$. The following operations are provided by $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ to higher-level protocols (for reference purposes, we provide a formal specification using the iUC template in Appendix G):

- **Start a key exchange:** A key exchange user $(pid, sid, role)$ with $\text{userStatus}(pid, sid, role) = \perp$ starts a key exchange upon receiving $m = (\text{InitKE}, pid', m')$, where pid' denotes the party ID of the intended partner and $m' \in \{0, 1\}^*$ is an arbitrary bit string which the realization might use in the key exchange protocol but that is not further interpreted by $\mathcal{F}_{\text{key-use}}^{\text{MA}}$. Upon receiving this message, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ sets $\text{userStatus}(pid, sid, role) \leftarrow \text{started}$, sets $\text{intendedPartner}(pid, sid, role) \leftarrow pid'$, and forwards m to the adversary (in the name of the current user $(pid, sid, role)$).
- **Use established keys:** A user $(pid, sid, role)$ with the status $\text{userStatus}(pid, sid, role) = \text{exchangeFinished}$ can be used by higher-level protocols to access symmetric operations of the subroutine $\mathcal{F}_{\text{crypto}}$. To be more precise, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ forwards messages of the type **New**, **Equal?**, **Enc**, **Dec**, **MAC**, **MACVerify**, and **Derive** to $\mathcal{F}_{\text{crypto}}$ in the name of the current user $(pid, sid, role)$, as long as the message is a valid input for $\mathcal{F}_{\text{crypto}}$ that will lead to a response; in particular, pointers ptr in the requests have to actually point to a key that has already been generated. Upon receiving such a response, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ forwards this response to the higher-level protocol while internally keeping track of all pointers that the user has access to.
- **Close a key exchange:** A user $(pid, sid, role)$ with the status $\text{userStatus}(pid, sid, role) = \text{exchangeFinished}$ closes his session in $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ upon receiving **CloseSession**, by which he loses access to all of his keys. $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ sets $\text{userStatus}(pid, sid, role) \leftarrow \text{sessionClosed}$, notifies the adversary with a restricting message **CloseSession** (sent in the name of $(pid, sid, role)$),⁴⁸ and, after receiving any response from the adversary, returns $(\text{CloseSession}, \text{ok})$ to the user.

⁴⁸This models that one can observe whether some session is still active by monitoring the network of a party. Keeping this information secret is generally not a goal of secure key exchange protocols.

The adversary may corrupt a user $(pid, sid, role)$ before he has completed the key exchange or after he has closed a session, but not while the user is still in an completed and active session (i.e., not when $userStatus(pid, sid, role) = \text{exchangeFinished}$; see the discussion below). The status $userStatus(pid, sid, role)$ of that user is then changed to **corrupted**. The adversary gains full control over the corrupted user except for access to the subroutine $\mathcal{F}_{\text{crypto}}$, which is prevented. This models perfect forward secrecy as the adversary does not gain access to any keys after the session is closed, even if he corrupts one of the parties, and hence he also cannot, e.g., decrypt any of the ideally encrypted ciphertexts sent during a session.

Besides corrupting users, the adversary may declare two local sessions to belong to a global session and he decides when an (uncorrupted) user has successfully completed a key exchange:

- **Create global sessions:** The adversary may send the message $(\text{GroupSession}, (pid_I, sid_I, \text{initiator}), (pid_R, sid_R, \text{responder}))$ to $\mathcal{F}_{\text{key-use}}^{\text{MA}}$, where $userStatus(pid_I, sid_I, \text{initiator}) \in \{\text{started}, \text{corrupted}\}$, $userStatus(pid_R, sid_R, \text{responder}) \in \{\text{started}, \text{corrupted}\}$, and both users are not yet part of a global session. The functionality $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ then sets the state of uncorrupted users to **inSession** and adds $((pid_I, sid_I, \text{initiator}), (pid_R, sid_R, \text{responder}))$ to the set of global sessions **sessions**. $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ then uses the $(\text{GetPSK}, name)$ command of $\mathcal{F}_{\text{crypto}}$, where $name \in \{0, 1\}^*$ is a fresh value that has not been used before, to get pointers to an unknown key k of type \mathbf{t}_{key} for the two users (if the received key is corrupted and hence known, then $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ asks for another key using a different fresh $name'$ until its gets an uncorrupted one). Finally, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ returns $(\text{GroupSession}, \text{success})$ to the adversary.

This command enforces three expected properties of secure key exchange sessions: Firstly, honest users are in a global session with a single uniquely defined different user, who itself is not part of any other sessions (ruling out replay attacks in particular). Secondly, the roles of users in the same session are complementary, i.e., it is not possible for two initiators or two responders to accidentally exchange a key. Thirdly, users in the same session end up with the same session key which is not known/accessible by anyone outside of that session. We note that, while we allow the adversary to pair an uncorrupted user with a corrupted one, the corrupted user will not get access to the session key in $\mathcal{F}_{\text{crypto}}$ (as already mentioned above). Observe that the fourth expected property, namely authentication, is not checked in this step. Instead, authentication is checked right before the key exchange is finished and a session key is output, which allows for a wider variety of realizations without weakening the security guarantees provided by this functionality (see also the discussion below).

- **Finish a key exchange:** The adversary may send the message **FinishKE** to some honest user $(pid, sid, role)$ with $userStatus(pid, sid, role) = \text{inSession}$ to complete the key exchange for that user. This message is accepted only if the user $(pid, sid, role)$ is in a global session with his intended partner, i.e., he is in a session with $(pid', sid', role')$ such that $pid' = \text{intendedPartner}(pid, sid, role)$. The functionality $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ then sets $userStatus(pid, sid, role) \leftarrow \text{exchangeFinished}$ and outputs $(\text{Established}, ptr)$ to the higher-level protocol that started

the key exchange of user $(pid, sid, role)$, where ptr is the pointer to the previously established session key k .

As mentioned above, this step ensures authentication of the session partner and therefore completes all expected properties of secure key exchanges.

The functionality $\mathcal{F}_{\text{key-use}}^{\text{UA}}$. The functionality $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ is similar to $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ but models unilateral authentication of the responder towards the initiator only. That is, it gives an initiator the same guarantees as $\mathcal{F}_{\text{key-use}}^{\text{MA}}$, while a responder may accept any connection without authentication. More formally, $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ differs from $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ as follows (for reference purposes, we provide a formal specification in Appendix G):

- Responders no longer indicate an intended session partner when starting a key exchange.
- The adversary may instruct $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ to output a key (**FinishKE**) for an uncorrupted user of role responder that has already started a key exchange even if that user is not yet part of a global session. This models a lack of authentication of the initiator, i.e., the responder might output a session key even if he is not in a session with a corresponding initiator instance.
- If an honest responder user is instructed to output a session key, no checks regarding the identity of the session partner are performed. Furthermore, unless the responder is in a global session with an honest initiator, the session key may be corrupted/known.
- Responder users that have already output a key may still be mapped into a global session with an initiator user if i) the responder is not yet part of a global session and ii) his session key is uncorrupted/unknown. The session partner will then receive the same session key.

Discussion. The functionality $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ assumes that responders know the expected PID of the initiator at the start of the key exchange. One could easily define a variant where the responder learns the PID of the initiator only at the end of the key exchange. Note, however, that all PIDs, both from users and from their expected peers, are determined freely by the environment for $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ anyway, so the environment can always choose the identities at the start of a run appropriately.

The corruption model of both $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ and $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ requires the corruption status of instances to stay unchanged after completing and before closing a key exchange session. This is not strictly necessary for the ideal functionalities themselves (we could easily define them to model full dynamic corruption). But due to the commitment problem realizations typically have to adopt a similar or more restricted corruption model anyway. Therefore, we chose to also restrict the corruption model of $\mathcal{F}_{\text{key-use}}^{\text{MA}}/\mathcal{F}_{\text{key-use}}^{\text{UA}}$ as this makes these functionalities easier to use by higher-level protocols. We note that this is not a strong restriction compared to full dynamic corruption, as session keys from key exchange protocols are usually very short lived, and hence, the window for corruption is small.

While $\mathcal{F}_{\text{key-use}}^{\text{MA}}/\mathcal{F}_{\text{key-use}}^{\text{UA}}$ are inspired by a functionality proposed in [96], the functionalities differ in several important aspects: As mentioned before, unilateral authentication is not considered in [96]. Also, $\mathcal{F}_{\text{key-use}}^{\text{MA}}/\mathcal{F}_{\text{key-use}}^{\text{UA}}$ model perfect forward secrecy, unlike the functionality in [96]. The functionality in [96] supports only symmetric encryption as an operation for higher-level protocols, and hence is insufficient for modeling the cryptographic operations of many higher-level protocols. Furthermore, most common ideal functionalities for mutually authenticated key exchange in the literature, including the functionality of [96] but also, e.g., the one by Canetti and Krawczyk [49], impose overly strict security requirements. Thus, there are some reasonable protocols that cannot realize these functionalities. To be more precise, those functionalities require that the views of both users in a session are identical when the first user outputs its key. In other words, if, e.g., Alice wants to talk to Bob and outputs a session key, then the protocol must not only ensure that Alice’s session partner is indeed Bob, but also that Bob believes he is talking to Alice (even if Bob has not even finished his part of the protocol yet). However, this is not the case in some protocols such as the SIGMA protocol family, which transmit and check the identity of the initiator only in the last protocol message. Hence, while the initiator knows that he is talking to his intended communication partner when he outputs a key, the responder has not yet confirmed the identity of the initiator, and thus their views may differ. Even though these protocols cannot realize the functionalities from [49, 96] and the like, the SIGMA protocol family is still reasonable as this protocol family ensures that the responder learns/confirmes the correct identity of the initiator before outputting his own session key (as we show in Section 5.3.2). By relaxing the requirements on establishing a global session and instead performing additional checks to authenticate the session partner when a session key is output, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ allows for the analysis of a wider variety of protocols while still retaining all security properties that are expected from a secure key exchange.

5.3. Security Analysis of Real World Key Exchange Protocols

In this section, we model and analyze several key exchange protocols to illustrate the usefulness of our framework. More specifically, we analyze one key exchange from the ISO 9798-3 protocol family [78] and the SIGMA protocol with identity protection [82]. Both protocols are meant to provide mutually authenticated key exchange. We also analyze the non-static mode of the OPTLS protocol [85] for unilaterally authenticated key exchange that served as the basis for the key exchange protocol in TLS 1.3 draft-09 [108], and point out a subtle bug in the original game-based proof.

We show that these protocols realize $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ and $\mathcal{F}_{\text{key-use}}^{\text{UA}}$, respectively, where the OPTLS protocol requires a small modification. In our modeling of these protocols, we use $\mathcal{F}_{\text{crypto}}$ to perform all cryptographic operations. By Theorem 5.1 respectively Corollary 5.1, $\mathcal{F}_{\text{crypto}}$ can then be replaced by its realization $\mathcal{P}_{\text{crypto}}$ so that the protocols use the actual cryptographic primitives. Due to the use of $\mathcal{F}_{\text{crypto}}$, the proofs are quite simple as they rely on high level information-theoretic arguments only; they do not need a single reduction, not even any probabilistic reasoning. At the same time, we obtain strong universal composability guarantees for the protocols. Moreover, the support for local session IDs in our iUC framework allows for a faithful modeling of the protocols. As discussed in Section 4.4 and in Section 5.2, other universal composability approaches typically impose pre-established (global) session IDs on the protocols, and hence, modify the protocols quite severely (see also related work in Section 5.5).

5.3.1. ISO Key Exchange Protocol

The ISO 9798-3 standard [78] defines several key exchange protocols that use digital signature schemes to provide authentication. We analyze one of these protocols, depicted in Figure 5.2, which is supposed to provide mutual authentication on top of a Diffie-Hellman key exchange. In what follows, we will just say “the ISO protocol” to refer to this protocol.

The modeling of the ISO protocol $\mathcal{P}_{\text{ISO}} = (\text{initiator}, \text{responder})$ in our framework is straightforward (for reference purposes we provide a formal specification using our template in Appendix H). We use two machines $M_{\text{initiator}}$ and $M_{\text{responder}}$ that implement the `initiator` and `responder` role, respectively, where both machines use $\mathcal{F}_{\text{crypto}}$ as a subroutine to perform all cryptographic operations (cf. Figure 5.3). Instances of $M_{\text{initiator}}/M_{\text{responder}}$ manage a single user $(pid, sid, role)$ each, and execute the ISO protocol for that user according to Figure 5.2, where each party pid has a single signing key in $\mathcal{F}_{\text{crypto}}$ that he reuses throughout all of his key exchanges. As soon as an instance receives some DH share, it uses the `BlockDHShare` command to ensure that $\mathcal{F}_{\text{crypto}}$ “knows” this share, and hence, fresh exponents do not create key shares that collide with it.⁴⁹ If verification of an incoming message fails, then the user aborts the key exchange.⁵⁰ At the end of the protocol, users create a DH key from g^x and g^y and use this

⁴⁹As mentioned earlier, this operation can be omitted when $\mathcal{F}_{\text{crypto}}$ is replaced with its realization. The resulting protocol is a natural implementation of the ISO protocol.

⁵⁰This constitutes a best practice in reality. Indeed, trying to recover from unexpected situations in a key

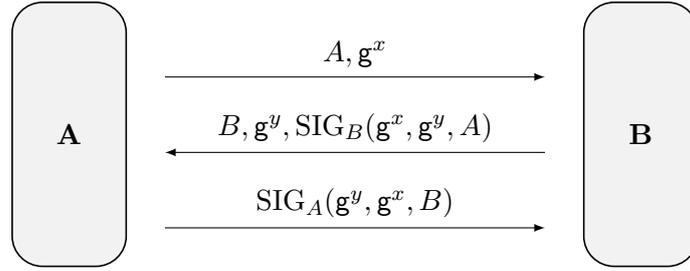


Figure 5.2.: The ISO protocol for mutually authenticated Diffie-Hellman key exchange between two parties A and B. At the end of the protocol, parties share a key g^{xy} that is then used to derive a session key.

to derive the session key of a fixed key type t_{key} that $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ supports; the security proof is independent of the specific key type. The users then output a pointer to that session key and subsequently provide the same interface as $\mathcal{F}_{\text{key-use}}^{\text{MA}}$, i.e., they allow a higher-level protocol to use $\mathcal{F}_{\text{crypto}}$ to perform (ideal) cryptographic operations with the session key.

There are two technical details that we have to take care of in our modeling of the protocol. Firstly, when an initiator instance receives (and accepts) the final message, it must output a pointer to the exchanged key to the environment while also providing the third protocol message m to the network. However, an instance of a machine in the IITM model, just as in other models for universal composability, can output only a single message before its activation ends. We deal with this issue by leaking the message m via a restricting message to the adversary, hence forcing him to return control to the instance, and then outputting the pointer to the higher-level protocol. Thus, the next time the adversary/environment is activated it knows the final message m and can, e.g., deliver it to its intended recipient.⁵¹ Secondly, a user of \mathcal{P}_{ISO} has already obtained some pointers to symmetric keys before generating a pointer with ID n to the session key. This is unlike $\mathcal{F}_{\text{key-use}}^{\text{MA}}$, where the first pointer with ID 0 points to the session key. We address this mismatch by letting \mathcal{P}_{ISO} perform some pointer mapping, i.e., if a higher-level protocol requests pointer 0 then this is internally mapped to the pointer with ID n , and so on.

We consider a corruption model between static corruption and full dynamic corruption. That is, a user $(pid, sid, role)$ of \mathcal{P}_{ISO} can become corrupted before he starts a key exchange (i.e., before sending `InitKE`) or after his session has been closed, but not while the key exchange is in progress or while the session key is still being used. We do not use full dynamic corruption since DH-based key exchanges with full dynamic corruption are not secure in a universal composability sense; not even if dynamic corruption is allowed only during the key establishment

exchange, instead of just starting a fresh key exchange session, can lead to serious attacks such as the KRACK attack on the WPA2 protocol [113].

⁵¹Alternatively, we could also store the message m and only output the pointer to the environment. We would then allow the adversary to manually request the stored message m via some special request on the network. This was what we did in our original publication in [89]. Both modelings have essentially the same effect, namely, they guarantee that a pointer is returned to the environment once the initiator completes the key exchange, which is precisely what happens in reality, while also allowing the adversary to learn and then use the message m once he becomes active again. We have chosen to actively leak m via a restricting message in this thesis since it yields a slightly cleaner protocol specification.

phase, i.e., while no session key has been output yet. Intuitively, this is because the simulator might have to commit to an ideally generated session key once the first user of a session finished the key exchange. However, the adversary can then corrupt the second user, obtain his secret DH exponent, and check whether the DH session key was replaced with an ideal one. This is also possible if the session key is never output directly but rather used to key some cryptographic primitive, such as in our setting. In this case the adversary can instead simply check whether cryptographic operations were performed with the expected DH key, e.g., by checking a MAC that was computed with the session key. However, as already discussed for $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ in Section 5.2, our corruption model is actually not a very strong restriction compared to full dynamic corruption and still yields a reasonable modeling of reality.

On a more technical level, explicit corruption of a user $(pid, sid, role)$ (that has not yet started or already closed his key exchange) is allowed only if the public key of party pid is already corrupted. This models that the adversary has access to the signing key via a corrupted user, allowing for message forgery, and thus the signing key must be considered corrupted as well. Once a user is explicitly corrupted, the adversary may also access $\mathcal{F}_{\text{crypto}}$ in its name, except for establishing pre-shared keys, asymmetric encryption/decryption, and nonce generation.⁵² However, the adversary may only create new pointers to known keys/exponents in $\mathcal{F}_{\text{crypto}}$. That is, the adversary may not use the **New** or **GenExp** commands; instead, he can use the **Store** and **StoreExp** commands to insert a new known key/exponent into $\mathcal{F}_{\text{crypto}}$. Also, the corrupted instance ensures that the adversary cannot access any pointers to symmetric keys/exponents of a closed key exchange session, as the ISO protocol deletes this information after a session.

In addition to explicit corruption, a user $(pid, sid, role)$ of \mathcal{P}_{ISO} with intended session partner pid' also considers itself to be (implicitly) corrupted if the signing key of pid or pid' was corrupted before $(pid, sid, role)$ has completed the key exchange. This reflects that authentication of parties is based on the security of the long term secrets, and hence no security guarantees can be given if any of the long term secrets of the session peers is corrupted. For this reason, such users also mark their exponents as known when they are created.⁵³ Since we want to consider a corruption model where corruption of a user is possible only before a key exchange has started or after the protocol is finished but not during the actual key exchange, we have to deal with the situation that long term secrets, namely, the signing keys in $\mathcal{F}_{\text{crypto}}$, can technically be dynamically corrupted at any point during a key exchange. We address this as follows: When the key exchange is started, a user determines his corruption status once (based on its own corruption status and the corruption status of the involved signing keys in $\mathcal{F}_{\text{crypto}}$) and stores this status. Upon finishing the key exchange and before outputting the key, the user checks that its corruption status did not change and only then outputs the key. Otherwise, the user

⁵²These operations are not used in the ISO protocol or after the key exchange, i.e., they do not affect security, and therefore blocking/ignoring them simplifies the presentation and security analysis without weakening the statement.

⁵³On a technical level, this is done via a **RetrieveExp** command. Note that, once we replace $\mathcal{F}_{\text{crypto}}$ with $\mathcal{P}_{\text{crypto}}$, both known and unknown exponents behave identical. Hence, this command can then be omitted, yielding a natural implementation of the ISO protocol.

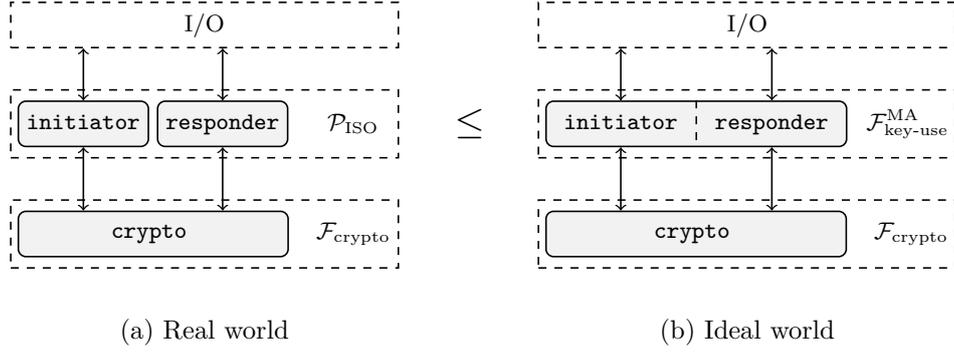


Figure 5.3.: The static structures of the real world protocol, the ISO protocol \mathcal{P}_{ISO} , and the ideal world protocol, the ideal key exchange functionality $\mathcal{F}_{\text{key-use}}^{\text{MA}}$, from Theorem 5.2.

stops and blocks all future requests as the adversary tried to break out of the corruption model.

The following theorem states that the ISO protocol is a secure universally composable mutually authenticated key exchange protocol (see also Figure 5.3 for a depiction of the static structure of the protocols in the real and ideal world).

Theorem 5.2. *Let \mathcal{P}_{ISO} be the ISO protocol as described above, let $\mathcal{F}_{\text{crypto}}$ be the ideal crypto functionality with some fixed parameters, and let $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ be the ideal functionality for mutually authenticated key exchanges with the same parameter \mathbf{t}_{key} used by \mathcal{P}_{ISO} . Then the following holds true:*

$$(\mathcal{P}_{\text{ISO}} \mid \mathcal{F}_{\text{crypto}}) \leq (\mathcal{F}_{\text{key-use}}^{\text{MA}} \mid \mathcal{F}_{\text{crypto}}).$$

As mentioned before, the proof of this theorem does not require any reductions, not even probabilistic reasoning, which greatly simplifies the overall proof.

Proof. In the following, we say that a party pid is corrupted if the signing key of party pid is corrupted. Also recall that we call a user $(pid, sid, role)$ corrupted if he returns `true` when asked for its corruption status by the environment, and that we call a user $(pid, sid, role)$ explicitly corrupted if the adversary directly corrupted $(pid, sid, role)$. Since there is the same protocol $\mathcal{F}_{\text{crypto}}$ in both the real and ideal world, we will denote $\mathcal{F}_{\text{crypto}}$ in the ideal world by $\mathcal{F}'_{\text{crypto}}$ instead to make the proof easier to follow.

We have to define a simulator \mathcal{S} and show that $\{\mathcal{E}, (\mathcal{P}_{\text{ISO}} \mid \mathcal{F}_{\text{crypto}})\} \equiv \{\mathcal{E}, \mathcal{S}, (\mathcal{F}_{\text{key-use}}^{\text{MA}} \mid \mathcal{F}'_{\text{crypto}})\}$ for all environments $\mathcal{E} \in \text{Env}((\mathcal{P}_{\text{ISO}} \mid \mathcal{F}_{\text{crypto}}))$. The simulator \mathcal{S} internally simulates the full protocol $(\mathcal{P}_{\text{ISO}} \mid \mathcal{F}_{\text{crypto}})$, including $\mathcal{F}_{\text{crypto}}$, and works as follows:

- \mathcal{S} keeps the corruption statuses of users (i.e., entities) in $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ and simulated users of \mathcal{P}_{ISO} synchronized. Note that this is indeed possible as users of \mathcal{P}_{ISO} cannot become corrupted (directly or indirectly) if they have finished but not closed their key exchange.
- When \mathcal{S} has to initialize $\mathcal{F}_{\text{crypto}}$ in the simulation, \mathcal{S} first sends a message to $\mathcal{F}'_{\text{crypto}}$ to initialize it and receives a group (G, n, g) in response which is used for the simulation of

$\mathcal{F}_{\text{crypto}}$. \mathcal{S} then asks the environment for the cryptographic algorithms and forwards them to $\mathcal{F}'_{\text{crypto}}$.

- When $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ indicates that a user $(pid, sid, role)$ has started a key exchange, \mathcal{S} does the same in its internal simulation.
- If an uncorrupted initiator $(pid, sid, \text{initiator})$ in the simulation finishes a key exchange and outputs a pointer to a session key, then \mathcal{S} instructs $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ to create a session from that user $(pid, sid, \text{initiator})$ and the user $(pid', sid', \text{responder})$ that created the signature in the second protocol message. The subroutine $\mathcal{F}'_{\text{crypto}}$ of $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ will then ask \mathcal{S} to provide the value for the session key; \mathcal{S} provides the same value that is used in its simulation as session key. Finally, \mathcal{S} instructs $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ to output the session key pointer for $(pid, sid, \text{initiator})$.
- If an uncorrupted responder $(pid, sid, \text{responder})$ in the simulation finishes a key exchange and outputs a pointer to a session key, \mathcal{S} instructs $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ to output the session key pointer for $(pid, sid, \text{responder})$.
- While a session key is used, the simulator may be asked by $\mathcal{F}'_{\text{crypto}}$ to provide new keys (e.g., when deriving keys). In this case, \mathcal{S} simulates the same operation in $\mathcal{F}_{\text{crypto}}$ and forwards the keys to $\mathcal{F}'_{\text{crypto}}$.
- The simulator \mathcal{S} keeps the key sets (i.e., keys and $\text{keys}_{\text{known}}$) of the simulated $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$ of $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ synchronized. More specifically, when the simulator is notified of a known key via a `AddedKnownKey` message by $\mathcal{F}'_{\text{crypto}}$, then \mathcal{S} adds this key to keys and $\text{keys}_{\text{known}}$ of the simulated $\mathcal{F}_{\text{crypto}}$ and returns control to $\mathcal{F}'_{\text{crypto}}$. Furthermore, whenever a new key (known or unknown) is added to $\mathcal{F}_{\text{crypto}}$ that is not already present in $\mathcal{F}'_{\text{crypto}}$ (such as session keys or keys created during the key usage phase of the key exchange; see above), then \mathcal{S} adds the same key with the same status to $\mathcal{F}'_{\text{crypto}}$ via the `AddKey` command.
- If \mathcal{S} is notified by $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ that some user $(pid, sid, role)$ has closed his session, \mathcal{S} updates the internal simulation accordingly and responds with `ok`.
- \mathcal{S} uses the internal simulation to process inputs/outputs for/from corrupted instances.

We now show that \mathcal{S} is a good simulator. As explained in Section 5.1.2, due to the use of restricting messages we can conveniently assume that all operations performed by $\mathcal{F}_{\text{crypto}}$ are atomic and without any side effects on the machines of \mathcal{P}_{ISO} . Hence, there are no edge cases that we have to consider while, e.g., verifying or creating a signature to process an incoming message, which simplifies the overall proof.

First, observe that \mathcal{S} can indeed keep the key sets of $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$ synchronized. In particular, by a simple inductive argument it follows that the simulator can always add new keys k from the simulated $\mathcal{F}_{\text{crypto}}$ to $\mathcal{F}'_{\text{crypto}}$ as k does not exist yet in $\mathcal{F}'_{\text{crypto}}$. This implies that $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$ behave identical in terms of freshness checks, key collisions, and choice of real/ideal operations. Also observe that the simulation of honest users which have closed their key exchange is trivially perfect.

Hence there are essentially four cases where we have argue that the simulation is perfect: Honest initiators during key establishment, honest responders during key establishment, honest users that have already finished (but not closed) a key exchange, and corrupted users.

Case 1: Let $(pid_I, sid_I, initiator)$ be an uncorrupted initiator that wants to establish a session with party pid' . It is easy to see that the simulator can perfectly simulate the behavior of such a user up to the point when it outputs a key as the behavior does not depend on any secret data from $\mathcal{F}_{key-use}^{MA}$ or \mathcal{F}'_{crypto} . In particular, honest users will use \mathcal{F}_{crypto} only to create/verify signatures, and exchange Diffie-Hellman keys; both of these operations are unavailable in \mathcal{F}'_{crypto} and can thus be internally simulated by \mathcal{S} without needing to synchronize the simulation with the ideal world.

We have to argue that \mathcal{S} finds a user in the responder role belonging to the intended partner pid' that can be paired with $(pid_I, sid_I, initiator)$: If $(pid_I, sid_I, initiator)$ outputs a session key pointer, then it must have accepted the second message of the ISO protocol and the signing key of its intended partner pid' must still be uncorrupted (otherwise, the initiator would consider itself to be corrupted). Hence, there is some entity belonging to the intended partner pid' , say the user $(pid', sid', role')$, that has signed the message $m = (g^x, g^y, pid_I)$, where x is the secret exponent of $(pid_I, sid_I, initiator)$ and y is the secret exponent of $(pid', sid', role')$. This user is uncorrupted: On the one hand, it cannot be explicitly corrupted by the adversary as the party pid' is still uncorrupted. On the other hand, as $(pid', sid', role')$ considers pid_I to be the partner of the key exchange (as acknowledged in the signature), we know that $(pid', sid', role')$ also does not consider itself to be corrupted due to corrupted signing keys. Next, we argue that this user is a responder, i.e., $role' = \mathbf{responder}$: If it were an initiator, then the signed message m would imply that this instance received and accepted the second protocol message containing a message $m' = (g^y, g^x, pid_I)$ signed by an uncorrupted key exchange user of party pid_I , where x is the secret exponent of that user. However, as x/g^x is created ideally, there is only one such user that would sign such a message, namely $(pid_I, sid_I, initiator)$, who does not output any signatures before accepting the second message. This implies $role' = \mathbf{responder}$. We still have to show that $(pid', sid', role')$ was not yet assigned to a session by \mathcal{S} : The simulator pairs responders with those honest initiators that accept the second message, but as x/g^x is unique, the only honest initiator that accepts this message is $(pid_I, sid_I, initiator)$. Hence, we have that $(pid', sid', role')$ is not yet part of a global session and can be paired with $(pid_I, sid_I, initiator)$. Finally, observe that both x/g^x and y/g^y have been created ideally (with $g^x \neq g^y$ as both were generated by different honest users) and are only used for computing the unknown DH key g^{xy} . Thus the session key that is derived from g^{xy} is also considered unknown in \mathcal{F}_{crypto} , as is the case in \mathcal{F}'_{crypto} . Note that the simulator can indeed provide the exact same session key from the simulation to \mathcal{F}'_{crypto} as the key sets are synchronized. Also note that, since both DH key generation and key derivation are performed ideally, only $(pid_I, sid_I, initiator)$ and $(pid', sid', role')$ can get a pointer to this key, which matches the behavior of $\mathcal{F}_{key-use}^{MA}$.

Case 2: Now, let $(pid_R, sid_R, \text{responder})$ be an uncorrupted responder that wants to establish a session with party pid' . By the same argument as above we have that the simulation is perfect until the session key is output. Hence, we only have to show that, when $(pid_R, sid_R, \text{responder})$ outputs a pointer to a session key, then it is already in a global session (in $\mathcal{F}_{\text{key-use}}^{\text{MA}}$) with an initiator belonging to pid' that has output a pointer to the same session key.

Observe that, if an honest user $(pid_R, sid_R, \text{responder})$ outputs a session key pointer, then he has accepted the third protocol message and pid' must still be uncorrupted. In other words, there is an entity of pid' , say the user $(pid', sid', role')$, that has signed the message $m = (g^y, g^x, pid_R)$, where y is the secret exponent of $(pid_R, sid_R, \text{responder})$ and x is the secret exponent of $(pid', sid', role')$. This user is uncorrupted by the same argument as in Case 1. We now argue that this user is an initiator, i.e., $role' = \text{initiator}$: Suppose by contradiction that $role' = \text{responder}$, i.e., the message was signed by a responder whose secret exponent is x and who has received the group element g^y . Recall that, whenever a responder receives a DH key share g^y in the first message, he immediately uses the `BlockDHShare` command on g^y . Thus, afterwards no (honest) user will be able to generate g^y via a `GenExp` command. Hence, the user $(pid_R, sid_R, \text{responder})$ must have generated y/g^y before $(pid', sid', role')$ has received the first protocol message, i.e., the user $(pid_R, sid_R, \text{responder})$ received the first protocol message before $(pid', sid', role')$ received his first message. However, by the same argument, $(pid', sid', role')$ must have generated x/g^x before $(pid_R, sid_R, \text{responder})$ has received the first protocol message, i.e., the user $(pid', sid', role')$ received the first message before $(pid_R, sid_R, \text{responder})$ received his first message. As this is a contradiction, we conclude that $role' = \text{initiator}$. We still have to argue that $(pid_R, sid_R, \text{responder})$ is already in a global session with $(pid', sid', role')$: As $(pid', sid', role')$ has signed a message, it has already completed its part of the key exchange and thus is in a session with some responder entity of party pid_R (as acknowledged in the signature on the final message m). More specifically, as argued above in Case 1, this will be the honest responder entity of party pid_R that signed the message $m' = (g^x, g^y, pid')$. However, the user $(pid_R, sid_R, \text{responder})$ is the only honest user that would sign such a message as y/g^y is unique, so $(pid', sid', role')$ is in a session with $(pid_R, sid_R, \text{responder})$. Note that both users use the same unknown exponents x and y to generate a DH key, with $x \neq y$, and those exponents are never paired with any other DH shares. Thus the responder indeed outputs a pointer to the same key as his session partner and that session key is unknown.

Case 3: Now consider an honest user in the key usage phase. As argued above, such a user in the real world/internal simulation will have a pointer to an unknown session key in $\mathcal{F}_{\text{crypto}}$ which can only be accessed by himself and his session partner. As \mathcal{S} creates global sessions in $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ from those users, and because key sets of $\mathcal{F}_{\text{crypto}}$ and $\mathcal{F}'_{\text{crypto}}$ are synchronized, this implies that in the ideal world both users end up with pointers to the same unknown key in $\mathcal{F}'_{\text{crypto}}$. The ideal world then offers the exact same operations using that session key as the real world. Hence, the behavior of both real and ideal world are identical.

Case 4: Finally consider a corrupted user. The simulator has full control over the I/O connection to higher-level protocols of such a user. If the user was explicitly corrupted by the adversary (i.e., it is under the control of the adversary) either before or after the key exchange, then the adversary gets access only to known keys which are used in a non-ideal way, namely, by running the actual cryptographic algorithms. Note in particular that if the attacker corrupts a user that has closed a key exchange he can still not gain new pointers to keys that were used during or after the key exchange as he does not have access to the secret exponent. Thus, the simulator can simply run the actual cryptographic algorithms to perfectly simulate the exact behavior of $\mathcal{F}_{\text{crypto}}$.

For an implicitly corrupted user $(pid, sid, role)$, observe that he still follows the protocol and hence he is easy to simulate in his key exchange phase by the same argument as for honest users. Once $(pid, sid, role)$ generates a session key, that key is either not shared with an honest user or $(pid, sid, role)$ blocks all operations instead of outputting and using the session key. To see this, suppose that an honest user outputs a pointer to the session key of $(pid, sid, role)$. By the arguments from Case 1 and Case 2, this implies that $(pid, sid, role)$ was mapped into a session with that user (as at most one other entity obtains a pointer to the same session key as an honest user). Furthermore, as also shown above, sessions are created only from two honest users that have both started the key exchange, i.e., $(pid, sid, role)$ was uncorrupted when the session was created. Hence, the corruption status of $(pid, sid, role)$ has changed after starting the key exchange session, which is a violation of the corruption model and causes $(pid, sid, role)$ to block. As both blocked users and corrupted users with session keys that are not shared by honest users are trivial to simulate, we have that the simulation is also perfect in this case.

To conclude the proof, observe that \mathcal{S} is indeed a responsive simulator as it fulfills the runtime conditions and it responds immediately to restricting messages as long as the environment does the same, which happens with overwhelming probability. This concludes the proof. \square

By Corollary 5.1, we can now replace $\mathcal{F}_{\text{crypto}}$ by its realization $\mathcal{P}_{\text{crypto}}$ which implies that the ISO protocol (when using the actual cryptographic operations) is a universally composable mutually authenticated key exchange protocol. In the formulation of the following corollary we use a wrapper \mathcal{F}^* for I/O traffic as introduced in Theorem 5.1. More specifically, we modify \mathcal{F}^* in a straightforward way and put it on top of the protocol \mathcal{P}_{ISO} in the real world and on top of $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ in the ideal world in order to make sure that, after a session key has been exchanged and while it is being used, the requirements of well-behaved environments are still met.

Corollary 5.2. *Let \mathcal{P}_{ISO} , $\mathcal{F}_{\text{key-use}}^{\text{MA}}$, and \mathcal{F}^* be as described above, and let $\mathcal{F}_{\text{crypto}}$ and $\mathcal{P}_{\text{crypto}}$ as required by Theorem 5.1. In particular, we have that $\mathcal{P}_{\text{crypto}} \leq \mathcal{F}_{\text{crypto}}$ and \mathcal{F}^* enforces well-behaved environments during the key usage phase. Then the following holds true:*

$$(\mathcal{F}^* \mid \mathcal{P}_{\text{ISO}}, \mathcal{P}_{\text{crypto}}) \leq (\mathcal{F}^* \mid \mathcal{F}_{\text{key-use}}^{\text{MA}}, \mathcal{F}_{\text{crypto}})$$

Proof. This statement follows directly from Corollary 5.1, Theorem 5.2, and transitivity of the \leq relation (cf. Lemma 3.6). For Corollary 5.1 note in particular that $\mathcal{Q} := (\mathcal{F}^* \mid \mathcal{P}_{\text{ISO}})$ indeed ensures the well-behaved property for the subroutine $\mathcal{F}_{\text{crypto}}$ of \mathcal{P}_{ISO} : Explicitly corrupted users do not have access to unknown keys or exponents, so they cannot violate the well-behaved property. Implicitly corrupted users have access to unknown exponents (and therefore keys) only if their corruption status has changed after the start of the key exchange, in which case they block instead of generating any keys. Honest users during the key establishment phase can violate the well-behaved property only by causing the commitment problem for Diffie-Hellman keys, i.e., set an unknown exponent to known after it was used to create an unknown key. However, honest users use their exponent only once right before they output the session key, and, as shown in our proof, this will be with a different DH key share of another honest user. Hence, honest users do not cause the commitment problem during the key establishment phase. During the key usage phase, \mathcal{F}^* enforces the well-behaved property of users following the protocol. \square

5.3.2. SIGMA Key Exchange Protocol

The SIGMA protocol with identity protection [82] is depicted in Figure 5.4. Unlike the ISO protocol, it uses the exchanged DH key to derive three other keys, two of which are used during the key exchange to ensure authentication and confidentiality of party IDs, while the third is used as session key. A key difference between both protocols is that the ISO protocol sends user identities in the clear over the network, whereas the SIGMA protocol protects user identities from eavesdroppers on the network by encrypting them before they are sent. As a result, the identity of the initiator cannot be transmitted in the first message when the key share of the responder is still unknown. Instead, the identity of the initiator is confirmed only in the third protocol message. As mentioned previously in Section 5.2, this can lead to situations where Alice as an initiator has finished the key exchange with her expected partner Bob, but Bob as a responder actually wants to perform a key exchange with Charlie and hence has a different view on the key exchange when Alice outputs the session key. Nevertheless, as we show below, Bob will learn the correct identity of his actual session partner in the third message and abort the key exchange if the identity is not as expected.

We model the SIGMA protocol $\mathcal{P}_{\text{SIGMA}} = (\text{initiator}, \text{responder})$ analogous to the ISO protocol \mathcal{P}_{ISO} from Section 5.3.1 (for reference purposes we provide a formal specification using our template in Appendix H), except for one difference: for now, implicitly corrupted users do not mark their exponents as known. To simplify the presentation, we handle this aspect in a separate step after the security analysis. In our modeling we use unauthenticated encryption to encrypt messages in the protocol, however, our proof is actually independent of whether encryption is authenticated and in fact whether encryption is used at all. This is because encryption is only needed to protect the identities of the parties, but not to ensure the basic security properties of a key exchange as required by $\mathcal{F}_{\text{key-use}}^{\text{MA}}$.

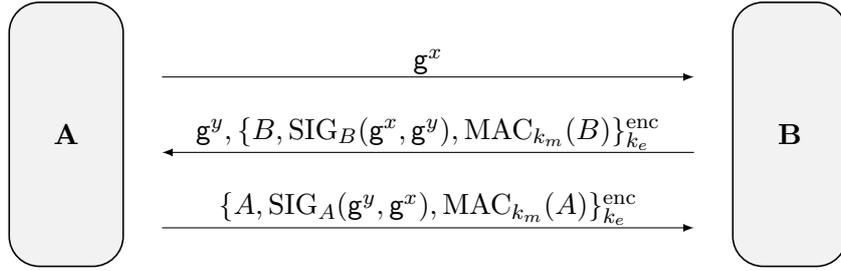


Figure 5.4.: The SIGMA protocol with identity protection. The keys k_e and k_m are derived from g^{xy} , where k_e is used for encryption and k_m is used for MACs during the key exchange. A third key k_s is also derived from g^{xy} and used as session key.

The following theorem states that the SIGMA protocol is a secure universally composable mutually authenticated key exchange protocol.

Theorem 5.3. *Let $\mathcal{P}_{\text{SIGMA}}$ be the SIGMA protocol, let $\mathcal{F}_{\text{crypto}}$ be the ideal crypto functionality with some fixed parameters, and let $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ be the ideal functionality for mutually authenticated key exchanges with the same parameter t_{key} used by $\mathcal{P}_{\text{SIGMA}}$. Then the following holds true:*

$$(\mathcal{P}_{\text{SIGMA}} \mid \mathcal{F}_{\text{crypto}}) \leq (\mathcal{F}_{\text{key-use}}^{\text{MA}} \mid \mathcal{F}_{\text{crypto}}).$$

Proof. The proof is similar to the proof of Theorem 5.2. We use the same terminology, notation, and even the same simulator \mathcal{S} , except that \mathcal{S} now internally simulates the SIGMA protocol instead of the ISO protocol. The simulator creates global sessions by combining an uncorrupted instance of an initiator with the instance of a responder that created the signature on $m = (g^x, g^y)$ in the second message. Just as for the ISO protocol, in the following we will not consider runs where the environment does not answer restricting messages immediately, as this happens with negligible probability.

By the same reasoning as in the proof of Theorem 5.2, we have that key sets of $\mathcal{F}_{\text{crypto}}$ in the simulation and $\mathcal{F}'_{\text{crypto}}$ in the ideal world are synchronized and that honest users with closed sessions are trivially simulated perfectly. Therefore, we only have to argue that the simulation works for four cases, namely, honest initiators during key establishment, honest responders during key establishment, honest users that have already finished a key exchange, and corrupted users.

Case 1: Let $(pid_I, sid_I, \text{initiator})$ be an uncorrupted initiator that wants to exchange a key with party pid' . This user is simulated perfectly until it outputs a session key as no I/O traffic and no secret information (in $\mathcal{F}_{\text{key-use}}^{\text{MA}}/\mathcal{F}'_{\text{crypto}}$) is involved. We have to argue that \mathcal{S} indeed finds a responder belonging to pid' that can be paired with $(pid_I, sid_I, \text{initiator})$: Once $(pid_I, sid_I, \text{initiator})$ outputs a pointer to a session key, because this user is uncorrupted, it must hold true that pid_I and pid' are also still uncorrupted. Furthermore, this user has accepted a signature by pid' on the message $m = (g^x, g^y)$. Thus we have that there must be an entity

belonging to party pid' , say the user $(pid', sid', role')$, that has generated this signature. Suppose that the user $(pid', sid', role')$ wants to establish a session with the party pid'' . Observe that $(pid', sid', role')$ was not explicitly corrupted by the adversary since this requires pid' to be corrupted first. Note, however, that the user $(pid', sid', role')$ might consider itself corrupted nevertheless because its intended session partner pid'' might be corrupted. In any case, as this user was not explicitly corrupted, we have that y is marked unknown: it was honestly generated and, as confirmed by the signature received by the initiator, only ever paired with g^x . The same also holds true for g^x from the initiator. Furthermore, we have $g^x \neq g^y$: the signature was created by a user $(pid', sid', role')$ honestly following the protocol. If $g^x = g^y$, then this would imply that $(pid', sid', role')$ considers g^x to be his own DH share. However, the only user honestly following the protocol who does so is $(pid_I, sid_I, initiator)$, who hasn't signed any messages yet; a contradiction. Thus the DH key created from g^x and g^y as well as all keys derived from it are marked unknown.

We proceed to show that $(pid', sid', role')$ is a responder, i.e., $role' = \mathbf{responder}$: Suppose by contradiction that this user was an initiator. As it has output a signature and a MAC, it must have already accepted the second protocol message before $(pid_I, sid_I, initiator)$ has received his second message. In particular, $(pid', sid', role')$ has accepted a MAC on the message $m' = pid''$. As the MAC is created from an unknown key which can only be derived by the users that own the exponents x or y , the MAC must have been created by $(pid_I, sid_I, initiator)$. However, that user does not create any MACs prior to receiving the second message. This implies that $(pid', sid', role')$ is not an initiator but rather a responder. Furthermore, $(pid', sid', role')$ is also not yet part of another session: By definition of \mathcal{S} , $(pid', sid', role')$ can only be in another session if an uncorrupted initiator has already accepted the signature on m before. However, the only uncorrupted initiator that accepts such a message is the single user that owns a pointer to x , i.e., $(pid_I, sid_I, initiator)$. Thus, the simulator can actually create a session from $(pid_I, sid_I, initiator)$ and $(pid', sid', role')$. Note that the session key in the realization is marked unknown and only the two users from the same session can potentially obtain a pointer to that key, just as required by the ideal protocol.

Case 2: Now consider an uncorrupted responder $(pid_R, sid_R, \mathbf{responder})$ that wants to exchange a key with party pid' . We have to show that if $(pid_R, sid_R, \mathbf{responder})$ outputs a pointer to a session key, the simulator can instruct $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ to also do so, i.e., $(pid_R, sid_R, \mathbf{responder})$ must already be in a global session with some user belonging to party pid' . Furthermore, both users must have obtained pointers to the same unknown session key in the realization/simulation. Suppose that $(pid_R, sid_R, \mathbf{responder})$ finishes the key exchange and outputs a pointer after accepting the third protocol message. Note that this implies that both pid_R and pid' are uncorrupted at this point. Therefore there must be some user belonging to pid' , say the user $(pid', sid', role')$, that has created the signature on $m = (g^y, g^x)$. Observe that this user cannot be explicitly corrupted as that would require pid' to also be corrupted. Thus we have that $(pid', sid', role')$ has generated x honestly and has only ever paired x with g^y . Furthermore,

as $y \neq x$ (recall that \mathbf{g}^x is blocked before \mathbf{g}^y is generated), this implies that all keys generated from \mathbf{g}^x and \mathbf{g}^y are also unknown.

We need to show that $(pid', sid', role')$ is an initiator, i.e., $role' = \text{initiator}$: Assume by contradiction that this user was a responder. The user must have received the first message and then created a signature on m before $(pid_R, sid_R, \text{responder})$ has received his first protocol message. This is because otherwise \mathbf{g}^x would have already been blocked via the `BlockDHShare` command, i.e., $(pid', sid', role')$ would have created an exponent $\neq x$ and therefore created a different signature. However, by the same reasoning it also follows that $(pid_R, sid_R, \text{responder})$ has received the first protocol message before $(pid', sid', role')$ has received his first protocol message, as otherwise $(pid_R, sid_R, \text{responder})$ would have created an exponent $\neq y$. Since this is a contradiction, we conclude that $(pid', sid', role')$ is an initiator.

We now argue that $(pid_R, sid_R, \text{responder})$ is indeed in a session with $(pid', sid', role')$; note in particular that we have to show that $(pid', sid', role')$ is uncorrupted as the simulator only pairs uncorrupted initiators. Let pid'' be the intended session partner of $(pid', sid', role')$. As $(pid', sid', role')$ has created a signature, it has already output a session key after accepting a signature on $m' = (\mathbf{g}^x, \mathbf{g}^y)$ and a MAC on pid'' . However, only $(pid_R, sid_R, \text{responder})$ and $(pid', sid', role')$ can create such a MAC (as the MAC key is unknown and no one else can derive the same key, see also the argument from Case 1), and $(pid', sid', role')$ does not create any MACs prior to accepting the second message. As $(pid_R, sid_R, \text{responder})$ will MAC pid_R only, we conclude $pid'' = pid_R$. As both pid' and pid_R are uncorrupted by assumption and $(pid', sid', role')$ was never explicitly corrupted, we conclude that $(pid', sid', role')$ is uncorrupted and thus assigned to a session by \mathcal{S} . Furthermore, that initiator was paired into a session with $(pid_R, sid_R, \text{responder})$ as no other user belonging to pid_R would create a signature on m' (he is the only user that has generated \mathbf{g}^y and all users belonging to pid_R act honestly). Thus, the simulator is able to instruct $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ to output a session key for $(pid_R, sid_R, \text{responder})$. Note that both $(pid_R, sid_R, \text{responder})$ and $(pid', sid', role')$ obtain pointers to the same unknown session key in the realization as that key was derived from the same unknown DH key, and no other instance can get a pointer to that key. This is just as required by $\mathcal{F}_{\text{key-use}}^{\text{MA}}$.

Case 3: The case of uncorrupted users that have finished a key exchange and are now using their session key is the same as in the proof of Theorem 5.2.

Case 4: Now consider a corrupted instance. The argument in this case is, again, almost identical to the one in the proof of Theorem 5.2, except for one detail: In the proof of Theorem 5.2, we showed that sessions are created only from pairs of honest users, which in turn implies that if an implicitly corrupted user is part of a session, then the corruption status changed after the session was created and thus the user blocks. However, as argued above for Case 1, in the SIGMA protocol it is entirely possible that the simulator combines an uncorrupted initiator $(pid_I, sid_I, \text{initiator})$ with an implicitly corrupted responder $(pid_R, sid_R, \text{responder})$ because the responder might have an expected partner $pid' \neq pid_I$ who is corrupted. To conclude that

the simulation also works for this case, we now argue that such a responder will never finish the key exchange and output a session key, i.e., the simulation works just as for blocked users.

Suppose by contradiction that such a responder $(pid_R, sid_R, \text{responder})$ *does* accept the third protocol message and outputs a session key. Recall from Case 1 that both x and y are unknown and thus all keys derived from the corresponding DH key are also unknown. So $(pid_R, sid_R, \text{responder})$ has accepted a MAC on pid' created with an unknown MAC key k , where pid' is different from pid_I and pid_R as those parties were not corrupted when the session was created (if pid' was one of those parties, then $(pid_R, sid_R, \text{responder})$ would have considered itself to be uncorrupted when the session was created, in contradiction to the assumption). However, the only users that have access to k and can therefore create ideal MACs from that key are $(pid_I, sid_I, \text{initiator})$ and $(pid_R, sid_R, \text{responder})$, both of which will only ever create MACs for the messages pid_R or pid_I , respectively. Thus we can conclude that $(pid_R, sid_R, \text{responder})$ will never receive a third protocol message that he accepts, i.e., if he receives a third message at all, he will just abort the key exchange and stop. The simulation of such users is trivially perfect.

Finally, it is easy to see that \mathcal{S} is a responsive simulator as it fulfills the runtime requirements and answers restricting messages immediately if the environment also does so. This concludes the proof. \square

We still have to deal with the fact that, in the above modeling, DH key shares from both uncorrupted and also implicitly corrupted users are generated to be unknown, even though the latter are (generally) not supposed to obtain any form of security guarantees via ideal session keys. In particular, this modeling can lead to the commitment problem for DH keys, therefore preventing us from replacing $\mathcal{F}_{\text{crypto}}$ with its realization $\mathcal{P}_{\text{crypto}}$. We now deal with this issue by adjusting our modeling of the SIGMA protocol in such a way that the above security proof still applies but the commitment problem for DH keys does not occur and implicitly corrupted users never output a pointer to an ideal session key.

Implicitly corrupted responders are changed to generate unknown DH keys if and only if g^x (contained in the first protocol message) was generated by an honest initiator.⁵⁴ Otherwise, responders manually mark their DH shares as known after they have been created. This modeling is slightly different from our modeling of the ISO protocol since an implicitly corrupted responder might end up in a session with an uncorrupted initiator, who needs to output an ideally generated DH key. Note that using this modeling our security proof still applies: In all cases where an unknown DH key share is needed, the DH key share is indeed still unknown. While we did change some DH shares to be known, this only affects users that are internally simulated by the simulator and that do not share a key with any honest users anyway. Also note that those implicitly corrupted responders that still generate an unknown DH key share will

⁵⁴On a technical level, this is implemented using an ideal subroutine database where (only) honest initiators store their DH key shares. This can then be used by responders to check whether their key share needs to be marked unknown.

never finish the key exchange phase and output a pointer to a session key (this follows by the same argument as in Case 4 of the security proof). Hence, if an implicitly corrupted responder outputs a pointer to a session key, then this key is considered to be known which is exactly as expected for implicitly corrupted users.

The situation of (both implicitly corrupted and honest) initiators is a bit more complicated. This is mainly because initiators have to derive a DH key already just to decrypt and then check the MAC in the second message. If the environment provides a bad DH key share in the second message, say, one that was not generated by $\mathcal{F}_{\text{crypto}}$, then this could cause a commitment problem: the exponent x would be marked known as soon as the DH key is derived by the initiator, but g^x might have already been used by other users to derive an ideal DH key. Observe, however, that in our security proof we have shown that honest initiators who output a key will generate that key from another honestly generated DH key share g^y , i.e., they do not run into the above commitment problem. Similarly, if an honest responder finishes a key exchange, then he is in a session with an initiator that has not caused the commitment problem. In all other cases, users either do not complete the key exchange at all or already consider themselves to be corrupted. Hence, this problem is just an artificial one: we try to give security guarantees for users that will not complete the key exchange or are implicitly corrupted to begin with. If we were to run into this situation using an approach other than universal composability, say, a game-based model for secure key exchanges, then we would simply abort in our proof or perhaps rewind to a previous step. After all, we only have to replace and idealize session keys for those honest users that actually finish the key exchange. However, these techniques are not available in the universal composability approach: the protocol/simulator already has to commit to using a known or unknown g^x for an initiator before learning (by receiving the second protocol message) whether that initiator is even supposed to obtain security guarantees, i.e., whether g^x should have been known or unknown.

We deal with this technical issue by requiring the environment to commit to either of those cases whenever a new initiator starts a key exchange. More specifically, (i) an honest initiator generates g^x to be unknown iff he will receive a DH key share $g^y \neq g^x$ that was generated by a **GenExp** command in $\mathcal{F}_{\text{crypto}}$, and (ii) an implicitly corrupted initiator generates g^x to be unknown iff he will receive a DH key share $g^y \neq g^x$ that was generated by an honest user. This is implemented by asking the environment via a restricting message on the network to commit to a known or unknown status when g^x is generated, and then rejecting incoming second protocol messages that do not meet the above criteria.⁵⁵ Let us discuss this modeling:

- Observe that the environment is still free to create both types of initiators: those that will end up requiring security guarantees for DH keys and those that do not. Furthermore, the environment is in full control over the run, even to the point of being able to choose the exact algorithms and exponents being used. Therefore, even though the environment has to

⁵⁵On a technical level, again, this can be checked via ideal subroutine databases that store DH key shares that were generated by a **GenExp** comment or were generated by an honest user, respectively.

commit to a known/unknown exponent when initiators start their run, such an environment can still recreate all possible message flows and by this all possible attacks that might exist on the protocol. Hence, our modification only removed the ability of the environment to cause the commitment problem by creating situations where initiators try to give security guarantees even though they are not supposed to as they will not finish a key exchange or are implicitly corrupted.

- It is easy to check that our security proof still applies for this modeling of the SIGMA protocol. In particular, every time we need that an exponent is marked unknown, this is still the case (this is mainly due to the signatures which are over both DH key shares and created by an uncorrupted party, i.e., the user that signed those messages was honestly following the protocol).
- Change (i) can be omitted if the SIGMA protocol is used without encryption. In this case, honest initiators can always generate unknown DH key shares: the signature in the second message, which can be checked without deriving a DH key if no encryption is used, implies that g^y was generated using $\mathcal{F}_{\text{crypto}}$ and is different from g^x . Hence, an initiator will either abort before deriving a DH key or derive a DH key from an honestly generated g^y to then check the MAC in the message. In both cases he does not cause the commitment problem for DH keys.
- We need (ii) due to two reasons. Firstly, an honest initiator might receive a second message with a signature created from another (potentially implicitly corrupted) initiator, instead of a responder (see Case 1). In this situation, we need security of the MAC to argue that such a signature would never be created in the first place as an implicitly corrupted initiator receiving g^x in the second message will just abort the protocol. Secondly, when an honest responder accepts the third protocol message, we have to rule out the possibility that he received g^x from an initiator that has a different intended partner and might hence also consider himself to be implicitly corrupted (see Case 2). Again, here we need the security of the MAC to argue that this situation cannot occur. Note that, by the same arguments as for Cases 1 and 2 in the security proof, we have that an implicitly corrupted initiator with an unknown DH key share will never receive a second message that he accepts and hence will never finish the key exchange. Therefore all pointers to session keys output by implicitly corrupted initiators are considered known, just as expected.

Using the above modeling of the SIGMA protocol, by the same argument as for the ISO protocol we can use Corollary 5.1 to replace $\mathcal{F}_{\text{crypto}}$ by its realization $\mathcal{P}_{\text{crypto}}$.

Corollary 5.3. *Let $\mathcal{P}_{\text{SIGMA}}$ be modeled as described above, and $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ and \mathcal{F}^* be as described previously, and let $\mathcal{F}_{\text{crypto}}$ and $\mathcal{P}_{\text{crypto}}$ as required by Theorem 5.1. In particular, we have that $\mathcal{P}_{\text{crypto}} \leq \mathcal{F}_{\text{crypto}}$ and \mathcal{F}^* enforces well-behaved environments during the key usage phase. Then the following holds true:*

$$(\mathcal{F}^* \mid \mathcal{P}_{SIGMA}, \mathcal{P}_{crypto}) \leq (\mathcal{F}^* \mid \mathcal{F}_{key-use}^{MA}, \mathcal{F}_{crypto})$$

Proof. This follows by an analogous argument as for Corollary 5.2. Note that in particular that \mathcal{P}_{SIGMA} does not cause the commitment problem for DH keys: For responders, this is never an issue as they use their DH key share only once right after it is created and they use it with a DH share that is different from their own. For honest initiators that finish the key exchange, we have argued as part of the proof of Theorem 5.3 that exponents are used only once with an unknown DH key share to create the session key. For all other initiators with an unknown DH key share, this is implied by the above modeling where the environment is required to commit to sending a honestly generated DH key share (that is different from the one of the initiator) in the second message. \square

5.3.3. OPTLS Key Exchange Protocol

The OPTLS protocol family [85] specifies several key exchange protocols with unilateral authentication. OPTLS was built to meet the specific requirements of TLS 1.3 for key exchange; a slightly modified version was included in draft-09 of TLS 1.3 [108]. The security of the OPTLS protocol family was proven in a (non-modular) game-based model in [85]. In Figure 5.5, we show the so-called non-static mode of OPTLS, which we analyze in this section. For simplicity of presentation, we will refer to this mode by saying “the OPTLS protocol” in what follows. Unlike the ISO and SIGMA protocols, OPTLS also specifies a specific key derivation procedure, which we depict in Figure 5.6.

We model the OPTLS protocol $\mathcal{P}_{OPTLS} = (\text{initiator}, \text{responder})$ in the same way as the ISO protocol. We use the optional bit string m' , which is part of the `InitKE` message expected by $\mathcal{F}_{key-use}^{UA}$, to provide initiators with the *chello* message and responders with the *shello* message that is to be used in the key exchange. Unlike \mathcal{P}_{ISO} and \mathcal{P}_{SIGMA} , responders do not specify an intended session partner at the beginning (as the protocol does not authenticate the initiator to the responder) and thus also do not consider themselves to be corrupted if the signature key of their session partner is corrupted. Responders also do not check whether their corruption status has changed during the key exchange as they essentially execute just a single atomic step (processing the first message and sending the second one), i.e., we can slightly strengthen the corruption model in this aspect. Instead, responders in \mathcal{P}_{OPTLS} additionally consider themselves to be corrupted (and by this manually mark their DH share g^y to be known) if they have received a DH share g^x that was not generated by an initiator who considers himself to be uncorrupted.⁵⁶ This models that, due to the lack of authentication of initiators, there are no security guarantees given for the session keys of responders unless they happen to receive an DH key share from an honest initiator. To put it in other words, an attacker on the protocol should not be able to use

⁵⁶On a technical level, we implement this by adding an ideal subroutine database where (only) initiators that consider themselves to be uncorrupted store their shares. This database can then be used by responders to determine their corruption status.

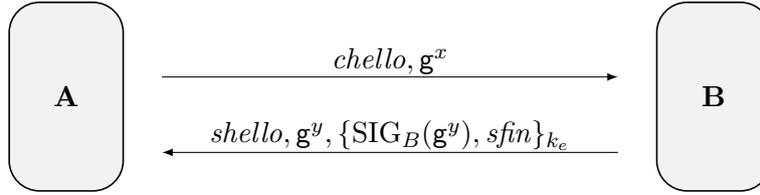


Figure 5.5.: The 1-RTT non-static mode of OPTLS. Both *chello* and *shello* are arbitrary bit strings that are exchanged during the protocol (they can be used to negotiate parameters for a higher-level protocol). The message *sfin* is a MAC on the whole key exchange, i.e., $sfin = \text{MAC}_{k_m}(\text{chello}, g^x, \text{shello}, g^y, \text{SIG}_B(g^y))$. The keys k_e (for encryption), k_m (for MACing) and the session key k_s are derived from the DH key g^{xy} as shown in Figure 5.6.

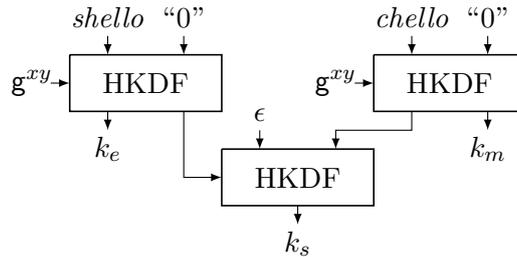


Figure 5.6.: Key derivation in the 1-RTT non-static mode of OPTLS. HKDF [83] is a key derivation function that takes as input a key (arrows on the left), context information (upper left arrows), and a seed (upper right arrows). It outputs a variable number of keys (bottom arrows).

trivial attacks on responders to distinguish real from ideal world; instead, an attacker must be able to break authentication or secrecy for honest initiators to distinguish both worlds. This is a natural modeling which closely resembles the situation in established game-based security definitions for unilaterally authenticated key exchanges, such as the one used for the analysis of TLS in [84], where the adversary may not use responders as test oracles for determining the security of the session key.

We model HKDF in the key derivation procedure (cf. Figure 5.6) via the `Derive` command of $\mathcal{F}_{\text{crypto}}$. As $\mathcal{F}_{\text{crypto}}$ provides only a single argument for key derivation, we concatenate both context information and seed and use the resulting string as seed for $\mathcal{F}_{\text{crypto}}$. This models that HKDF should provide independent keys if either seed or context information is changed. Another technical difference is that HKDF outputs a variable number of keys, while $\mathcal{F}_{\text{crypto}}$ outputs a single key for every seed. One can easily extend $\mathcal{F}_{\text{crypto}}$ to also support deriving multiple keys from a single seed and then realize it with a secure variable length output PRF, as has been done in [112]. Nevertheless, for simplicity of presentation, we use the current formulation of $\mathcal{F}_{\text{crypto}}$ and instead call the `Derive` command twice to obtain two keys. Formally, each of those calls uses a different seed which is obtained by prefixing the original seed with 0 or 1, depending on whether the first or second key is derived.

Perhaps surprisingly, $\mathcal{P}_{\text{OPTLS}}$ does *not* realize $\mathcal{F}_{\text{key-use}}^{\text{UA}}$, not even if we were to relax our

modeling by letting implicitly corrupted responders also use unknown DH key shares. To see this, consider the following setting: Assume that Alice as an initiator wants to exchange a key with Bob as a responder, where both parties are uncorrupted (i.e., their signature keys are uncorrupted and all of their key exchange instances follow the protocol honestly). Further assume that Alice has generated the share \mathbf{g}^x and receives Bob's DH share \mathbf{g}^y in the second protocol message. Observe that the responder belonging to Bob who signed \mathbf{g}^y might have received a different group element, say $h \neq \mathbf{g}^x$, in the first protocol message. If h was not honestly generated by $\mathcal{F}_{\text{crypto}}$, then y will be marked known after the calculation of h^y because the DDH assumption does not guarantee that an attacker learns nothing about y in this case (cf. Section 5.1.3). As y is marked known, the key \mathbf{g}^{xy} and all keys derived from it will also be marked known. Thus, we have no security guarantees for the MAC and an attacker can easily let Alice accept the second message even though there is no instance of Bob that can be paired with Alice (the instance that signed \mathbf{g}^y generates a different session key and thus cannot form a session with Alice). Furthermore, the session key obtained by Alice can also not be used ideally which, intuitively, is because an attacker might have obtained some information by letting Bob compute h^y which cannot be obtained in and hence reduced to the DDH setting.

We note that this is not a direct attack against the protocol and also not a technicality caused by the strong security requirements of the universal composability approach, but rather illustrates that assuming hardness of DDH, security of the PRF family, security of the digital signature scheme, and security of the MAC scheme is not sufficient to prove the security of this protocol mode. Indeed, we found that the original game-based security proof of this protocol from [85] is flawed: In the proof, which uses the same cryptographic assumptions, \mathbf{g}^{xy} is replaced by $\mathbf{g}^z, z \stackrel{\$}{\leftarrow} \{1, \dots, n\}$ during an early step of a hybrid argument (cf. Game 2 in the proof of Theorem 3 in [85]). The authors claim that this can be reduced to the DDH assumption. An adversary for the DDH game would have to simulate the protocol where \mathbf{g}^x of the initiator and \mathbf{g}^y of the responder are replaced with the challenges from the DDH game. In this situation, if the simulated responder received some group element $h \neq \mathbf{g}^x$ in his first message, then the adversary might have to calculate h^y and derive keys from it to compute the message that is output by the responder. This is impossible within the basic DDH game: as mentioned previously in Section 5.1.5, in groups where the DDH assumption holds true an attacker cannot compute h^y (for arbitrary group elements h). Furthermore, the adversary also has no other ways, such as an oracle, to compute the keys derived from h^y without computing h^y itself. Therefore the simulation of OPTLS fails in the reduction.

To fix this problem both in the original game-based setting and in our universal composability setting, one can use stronger assumptions than DDH. For example, one could use the PRF-ODH assumption [23, 79, 84], where the adversary additionally gets access to an oracle for calculating keys derived from h^y (where y is one of the secret exponents and h is provided by the adversary). This would then allow for properly simulating the responder in a reduction, even if he received $h \neq \mathbf{g}^x$ in his first message. As mentioned earlier, we leave a formulation of $\mathcal{F}_{\text{crypto}}$ based on

the PRF-ODH assumption for future work. An alternative fix for this problem (again for both settings) is to have \mathbf{g}^x signed as well, i.e., signing $(\mathbf{g}^x, \mathbf{g}^y)$ as in the SIGMA protocol. This allows for proving security using the DDH assumption as now the signature guarantees that the responder paired \mathbf{g}^y with \mathbf{g}^x only. In what follows, we denote this modified version of the OPTLS protocol by $\mathcal{P}'_{\text{OPTLS}}$ (cf. Appendix H for a formal specification using our template). The following theorem states that this variant is a secure universally composable unilaterally authenticated key exchange.

Theorem 5.4. *Let $\mathcal{P}'_{\text{OPTLS}}$ be the modified version of the OPTLS protocol, let $\mathcal{F}_{\text{crypto}}$ be the ideal crypto functionality with some fixed parameters, and let $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ be the ideal functionality for mutually authenticated key exchanges with the same parameter \mathbf{t}_{key} used by $\mathcal{P}'_{\text{OPTLS}}$. Then the following holds true:*

$$(\mathcal{P}'_{\text{OPTLS}} \mid \mathcal{F}_{\text{crypto}}) \leq (\mathcal{F}_{\text{key-use}}^{\text{UA}} \mid \mathcal{F}_{\text{crypto}}).$$

Proof. The proof is similar to the ones for the ISO and SIGMA protocols (cf. Theorems 5.2 and 5.3). Again, we reuse the same terminology, notation, and even the same simulator \mathcal{S} , except for the following changes: \mathcal{S} now internally simulates the modified OPTLS protocol $\mathcal{P}'_{\text{OPTLS}}$. If an honest initiator outputs a key in the simulation, \mathcal{S} creates a session from that instance and the instance of a responder that created the MAC in the second message. If an honest responder outputs a key, \mathcal{S} instructs $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ to also do so using the same key with the same known/unknown status (but without creating a session). In the following, we only have to reason about runs where the environment answers restricting messages immediately as this happens with overwhelming probability.

By the same reasoning as in the proof of Theorem 5.2, we have that key sets of $\mathcal{F}_{\text{crypto}}$ in the simulation and $\mathcal{F}'_{\text{crypto}}$ in the ideal world are synchronized and that honest users with closed sessions are trivially simulated perfectly. Therefore, we only have to argue that the simulation works for four cases, namely, honest initiators during key establishment, honest responders during key establishment, honest users that have already finished a key exchange, and corrupted users.

Case 1: Let $(pid_I, sid_I, \text{initiator})$ be an uncorrupted initiator that wants to establish a key with party pid' . It is easy to see that the simulation is perfect until a session key pointer is output. Note in particular that the simulator learns *chello* as part of the `InitKE` message from $\mathcal{F}_{\text{key-use}}^{\text{UA}}$.

Now assume that $(pid_I, sid_I, \text{initiator})$ finishes the key exchange and outputs a session key pointer in the realization/simulation. We have to show that \mathcal{S} can indeed group this initiator with a responder belonging to pid' in $\mathcal{F}_{\text{key-use}}^{\text{UA}}$. Furthermore, that responder must have output a pointer to the same (unknown) session key as the initiator in the realization/simulation. We start by arguing that the MAC key is unknown. As $(pid_I, sid_I, \text{initiator})$ is uncorrupted, we have that its intended session partner pid' is also uncorrupted. Thus, there must be an

entity belonging to pid' , say the user $(pid', sid', role')$, that created the signature on (g^x, g^y) in the second message. Furthermore, this user is a responder (as initiators do not sign any messages) and is uncorrupted (as pid' is uncorrupted and the DH share received in the first message was g^x , which was generated by an honest initiator). Thus, we have that g^y was created honestly, different from g^x , and g^y was used only with g^x to create a DH key. We conclude that g^y is still marked unknown, just as g^x , at the time when the second protocol message is received. This implies that all keys derived from g^{xy} are also marked unknown. Hence, we have that the MAC in the second protocol must have been created by $(pid', sid', role')$ as forgery is prevented, only $(pid', sid', role')$ and $(pid_I, sid_I, initiator)$ can get access to the MAC key, and $(pid_I, sid_I, initiator)$ does not create MACs. We conclude that both $(pid_I, sid_I, initiator)$ and $(pid', sid', role')$ use the same session parameters, namely, g^x , g^y , $shello$, and $chello$, to establish a session key. This implies that they output pointers to the same (unknown) session key. Finally, observe that $(pid', sid', role')$ is not yet part of another session: By the definition of \mathcal{S} , a responder gets paired only with an uncorrupted initiator that accepts the signature. However, no other uncorrupted initiator besides $(pid_I, sid_I, initiator)$ will accept the signature on g^x as all other honest initiators have different DH shares. Thus, \mathcal{S} can indeed pair both users in $\mathcal{F}_{key-use}^{UA}$.

Case 2: Now consider an uncorrupted responder $(pid_R, sid_R, responder)$ during session key establishment. Just as for Case 1, it is easy to see that the simulation is perfect until a session key pointer is output. Once that pointer is output in the realization/simulation, the simulator instructs $\mathcal{F}_{key-use}^{UA}$ to output the same key with the same known/unknown state (by optionally corrupting the key in \mathcal{F}'_{crypto}). As the key sets are synchronized, \mathcal{F}'_{crypto} accepts this key. So the output of session keys is simulated perfectly also for responders.

Case 3: Let $(pid, sid, role)$ be an uncorrupted user that has finished but not closed a key exchange. As argued above, if $(pid, sid, role)$ is an initiator, then he is in a session with a (honest) responder and both share a pointer to an ideal session key in \mathcal{F}'_{crypto} which is not accessible by any other users. This is precisely the same as in the realization and hence indistinguishable, also for all operations performed with the session key.

If $(pid, sid, role)$ is a responder, first observe that the simulation is perfect if the session key is known as then \mathcal{F}'_{crypto} uses the same real cryptographic algorithms as \mathcal{F}_{crypto} on the same inputs and keys, without any additional security checks and independent from any secret internal state. Hence, if someone, e.g., encrypts a message using the session key, even if it is a corrupted user handled in the internal simulation with access only to \mathcal{F}_{crypto} , then \mathcal{F}'_{crypto} will still behave just as \mathcal{F}_{crypto} during decryption of the message. Now suppose $(pid, sid, role)$ is a responder with an unknown session key k in \mathcal{F}'_{crypto} . This situation is more involved because Case 2, unlike in the proof for \mathcal{P}_{ISO} and \mathcal{P}_{SIGMA} , does not already imply that there is *at most one honest initiator* with access to the same session key k and which then gets paired with the responder such that both users are handled by \mathcal{F}'_{crypto} . We have to now show that this still holds true for \mathcal{P}'_{OPTLS} as

otherwise the simulation fails; in particular, a corrupted user with access to the same unknown key k could not be simulated correctly.

Since $(pid, sid, role)$ is honest, the DH key share \mathbf{g}^x received in the first message was generated by an initiator $(pid', sid', role')$ that considered itself to be honest. The user $(pid', sid', role')$ is the only one that might generate and output a pointer to the same session key k derived from \mathbf{g}^{xy} as \mathbf{g}^{xy} is an ideally generated unknown key. Note that the simulation is trivially perfect if $(pid', sid', role')$ does not output a pointer to that key, e.g., because it has received a share $\mathbf{g}^z \neq \mathbf{g}^y$ in its second protocol message or because it has blocked. Now suppose that $(pid', sid', role')$ does indeed output a pointer to a session key derived from \mathbf{g}^{xy} : in that case, the user must still be honest (he was honest at the start of the key exchange and would block if the attacker changed the corruption state inbetween) and has therefore accepted an ideally computed MAC for an unknown key that was ideally derived from \mathbf{g}^{xy} . Since $(pid, sid, role)$ is the only responder that can derive the corresponding MAC key, we have that the simulator will map $(pid, sid, role)$ and $(pid', sid', role')$ into the same session and both will derive the same unknown session key by Case 1. Hence, all session keys of honest users are handled internally only by $\mathcal{F}'_{\text{crypto}}$, i.e., the simulation is also perfect for this case.

Case 4: We now consider a corrupted user. This case is analogous to Case 4 from proof of Theorem 5.2. Note in particular that, by Cases 1 and 3, an *unknown* session key of an honest user is only ever shared by at most one honest user in the same global session. While, unlike in the proof of Theorem 5.2, a corrupted user might share the same *known* session key as an uncorrupted responder, this is also perfectly simulated as explained in Case 3, even though one of those users is handled by $\mathcal{F}_{\text{crypto}}$ and the other by $\mathcal{F}'_{\text{crypto}}$.

Finally, observe that \mathcal{S} is a responsive simulator as it fulfills the runtime requirements and guarantees immediate answers in those runs where the environment also answers immediately. \square

The OPTLS protocol, just as the SIGMA protocol, requires the initiator to already derive a DH key to decrypt a message before being able to check (via the signature contained in the message) that the key share \mathbf{g}^y was honestly generated. If it was not honestly generated, then we do not need to provide any security guarantees as the initiator will abort the protocol due to an incorrect signature anyways. Hence, we use the same technique as for the SIGMA protocol to prevent artificial commitment problems for DH keys: upon starting a key exchange, honest initiators ask the environment to commit to whether they will receive a key share \mathbf{g}^y that was generated in $\mathcal{F}_{\text{crypto}}$ and set their own key share \mathbf{g}^x to known if not. Just as for the SIGMA protocol, this does not affect the capabilities of an environment to perform an actual attack on the security of an exchanged key as an environment can still reproduce all message flows possible in reality. Furthermore, the security proof still carries over to this modified modeling. We note that, again just as for the SIGMA protocol, this change can be omitted if OPTLS is used without encryption; in that case, honest initiators can always create unknown DH keys.

Using this adjusted modeling of OPTLS, by Corollary 5.1 we can replace $\mathcal{F}_{\text{crypto}}$ by its realization $\mathcal{P}_{\text{crypto}}$.

Corollary 5.4. *Let $\mathcal{P}'_{\text{OPTLS}}$ be modeled as described above, $\mathcal{F}_{\text{key-use}}^{\text{UA}}$, and \mathcal{F}^* be as described previously, and let $\mathcal{F}_{\text{crypto}}$ and $\mathcal{P}_{\text{crypto}}$ as required by Theorem 5.1. In particular, we have that $\mathcal{P}_{\text{crypto}} \leq \mathcal{F}_{\text{crypto}}$ and \mathcal{F}^* enforces well-behaved environments during the key usage phase. Then the following holds true:*

$$(\mathcal{F}^* \mid \mathcal{P}'_{\text{OPTLS}}, \mathcal{P}_{\text{crypto}}) \leq (\mathcal{F}^* \mid \mathcal{F}_{\text{key-use}}^{\text{UA}}, \mathcal{F}_{\text{crypto}})$$

Proof. This follows by an analogous argument as for Corollary 5.3. □

5.4. Discussion

In the following, we highlight some of the key details of our case study where we are able to model reality very precisely. This illustrates the *flexibility* and *usability* of iUC, also compared to (conventions of) the UC and GNUC models, and therefore provides concrete examples for many of the statements from the discussion of iUC in Section 4.4.

Local SIDs: As already explained in detail in Section 4.4, many protocols, in particular real-world ones that were not built with universal composability models in mind, use so-called locally chosen SIDs as opposed to global pre-established SIDs. This is the case for all key exchange protocols from our case study: If an initiator starts a run, he does not know yet which exact responder instance will be part of his session and he also does not share any SID with his session partner; in fact, the protocol specifications do not include any SIDs at all. Instead, key exchange sessions are established dynamically throughout the protocol run depending on how the messages are routed, i.e., the responder instance that has received g^x and then generated g^y is the one that ends up in the session with the initiator. In particular, sessions are indeed created independently of how the initiating party and the responding party locally address their own runs of the key exchange protocol.

It is important to capture this setting precisely within a security analysis. If we were to instead model those key exchanges assuming global SIDs, then users of the same session already start with some shared information which could potentially be used to obtain security guarantees. In contrast, a key exchange with locally chosen SIDs, which is how the protocols from our case study are deployed in reality, uses a weaker setup assumption where no data is shared at the start of the protocol. Indeed, in [96] it was shown that there are natural key exchange protocols from practice that are insecure when they use locally chosen SIDs but become secure under the assumption that protocol participants of the same session already share a global SID.

Our case study illustrates that the iUC framework is fully capable of modeling locally chosen SIDs, both in terms of real protocols and of ideal protocols. A key ingredient for supporting local SIDs is the flexible addressing mechanism provided by the **CheckID** algorithm, which in turn is based on the general **CheckAddress** mode of the IITM model. This allows for defining ideal protocols which manage several entities, also with different SIDs, and then dynamically form sessions throughout the protocol execution. As already explained in detail in Section 4.4.3, most other UC-like models, including the UC and GNUC models, do not support local SIDs with their conventions. These models require workarounds to support local SIDs, if at all possible, and those workarounds come with serious drawbacks concerning both usability and flexibility.

Shared State: Many protocols share some state between multiple protocol runs and also across multiple protocol sessions. This is also the case for all of our key exchanges, where the same Diffie-Hellman group (G, n, g) and the same signing keys (one per party) are used throughout all protocol runs. The iUC framework allows for precisely modeling this setting: We are able to use

a single instance of $\mathcal{F}_{\text{crypto}}$ that manages all cryptographic operations and hence is able to use the same state to respond to requests from arbitrary users. Since iUC does not require protocols to have disjoint sessions, users from higher-level protocols are then free to use this single instance of $\mathcal{F}_{\text{crypto}}$ even if they do not share the same SID and are from entirely different protocol sessions. This makes it straightforward to reuse the same signing keys and also arbitrary other cryptographic values such as pre-shared keys. Of course, the same concepts can also be used to access arbitrary other (non-cryptographic) state across multiple protocol runs/sessions. For example, in $\mathcal{P}_{\text{OPTLS}}$ our corruption model uses a shared database to determine the corruption status of responders.

Altogether, the flexibility of iUC allows for a very precise modeling of protocols with state sharing. In particular, as already explained in detail in Section 4.4.3, this modeling is much more precise than so-called joint state realizations, which have to modify the protocols from our case study and therefore change their security properties.

Corruption Modeling: Our key exchanges illustrate that iUC supports a wide range of very different and potentially also very tailored corruption models with its four customizable corruption algorithms. For example, we are able to easily express a corruption model that lies between traditional static and dynamic corruption, i.e., the corruption model prevents corruption during the key exchange and key usage but allows for full corruption before starting and after closing the key exchange. Furthermore, the corruption status of users can be fine tuned to precisely capture the conditions under which the user retains security guarantees, such as uncorrupted long term keys of both parties.

Of particular interest is that the iUC framework allows for easily capturing also new interpretations of corruption that differ from what is typically considered in most UC-like models. Traditionally, conventions of UC-like models consider a user to be corrupted if the adversary has gained control over the user itself or one of his own local subroutines. This models that an attacker has gained access to and control over a certain program on the computer of that user in reality, which in turn typically invalidates all security guarantees. In comparison, iUC also fully supports a more general and more abstract interpretation of corruption, namely, one can interpret the corruption status as an indicator whether the user is still supposed to obtain security guarantees from the protocol. A breakdown in security guarantees resulting in a corrupted user can therefore be not just due to the attacker gaining direct control over (part of) the computer of the user, just as in traditional conventions of UC-like models, but also because underlying security assumptions have been violated, such as an honest majority taking part in the protocol.

Such a more abstract and general interpretation of corruption allows for capturing and analyzing a wider range of protocols and security settings. This is illustrated by our case study: Key exchange users consider themselves to be corrupted if the long-term secret (i.e., signature key) of their intended session partner is not secure. In this situation, while the attacker did not gain access to any of the programs on the user’s PC, the user still cannot provide any security

guarantees as security of long-term secrets is required to authenticate the session partner. To give another example, our ideal functionality $\mathcal{F}_{\text{crypto}}$ can properly capture the case where an attacker has gained access to the secret signing key of a party and can therefore forge signatures but did not necessarily gain control over any programs on the computer of the affected user. In particular, even if an attacker has corrupted a signature key, he still does not see the contents of the messages that are being signed by higher-level protocols and he also cannot control the outputs of signing/verification operations. This is unlike in other signature functionalities using the standard corruption mechanisms provided by the UC and GNUC models, or the formulation of $\mathcal{F}_{\text{sig-CA}}$ given in Section 4.3, which use a less fine-grained corruption model that over-approximates the capabilities of an attacker who has corrupted a signing key by giving that attacker full control over the I/O behavior of the affected user.

Responsive Environments: Responsive environments, which are fully supported by iUC, are an important factor for improving usability by simplifying specifications and proofs. In our case study, this can be seen, e.g., in the many situations where a key exchange user $(pid, sid, role)$ can send a message to $\mathcal{F}_{\text{crypto}}$ to perform a cryptographic operation, such as signing a message, and then simply wait for the response. We do not have to specify, or even think about, let alone argue in proofs what happens if the next activation of $(pid, sid, role)$ is not due to the expected signed message returned by $\mathcal{F}_{\text{crypto}}$, but rather due to, say, some input from the network or a new message from the environment. We also do not have to deal with any unintended state changes, such as signature keys being corrupted, while a local cryptographic operation is being processed. Due to responsive environments these cases occur only with negligible probability anyway and can therefore be ignored both in the protocol specifications and security proofs.

Furthermore, due to responsive environments we were able to drastically simplify key generation, in particular asymmetric keys, in $\mathcal{F}_{\text{crypto}}$. For example, we are now able to properly model the setup assumption that all parties of a protocol have already generated (and potentially distributed) their public keys before the start of the protocol. This is because, even if a key has not been provided by the attacker yet, we can simply ask him via a restricting message for the key once it is used for the first time. In contrast, the original formulation of $\mathcal{F}_{\text{crypto}}$, just as all other ideal functionalities for cryptographic primitives from the literature without responsive environments, had to manually handle cases where the attacker was not willing to provide keys. This in turn complicated higher-level protocols, which then also had to manually handle cases where $\mathcal{F}_{\text{crypto}}$ did not respond to a request for generating symmetric keys or using asymmetric keys. Considering that $\mathcal{F}_{\text{crypto}}$ models local operations only, there should not be any way for the attacker to interfere with key generation of honest users by not responding; this expectation is in fact already met by the realization $\mathcal{P}_{\text{crypto}}$ which generates keys locally. Responsive environments allow us to formalize and properly capture this expected property also for the ideal functionality.

5.5. Related Work

As already mentioned, our case study builds on and extends results from [96,97]. Both of these works have already been discussed and compared in Sections 5.1 and 5.2. We now discuss other closely related works.

In [43], Canetti and Gajek propose an abstraction of Diffie-Hellman key exchanges via an ideal key encapsulation functionality \mathcal{F}_{KEM} . There are two key differences to our results. Firstly, unlike $\mathcal{F}_{\text{crypto}}$ and our key exchange functionalities, the ideal key encapsulation functionality \mathcal{F}_{KEM} does not allow a user/higher-level protocol to use the exchanged key in an idealized way or to use it with other primitives, which entails reductions proofs. Secondly, a large class of protocols cannot be analyzed based on \mathcal{F}_{KEM} : in order to prove the realization via a Diffie-Hellman key exchange, the authors impose a very strong restriction on the environment/higher-level protocols, namely, an initiator may use his secret exponent x only with DH shares g^y that have previously been generated honestly via \mathcal{F}_{KEM} . This assumption is not met by many protocols, such as the key exchange protocols from our case study: for example, the intended session partner of an initiator might be corrupted and therefore sign a DH share h that was not generated by \mathcal{F}_{KEM} . This share would be accepted by the initiator and then used with the secret exponent x in \mathcal{F}_{KEM} , which violates the requirement.

None of the key exchanges from Section 5.3 has been faithfully analyzed in a universal composability setting before (see [47,48,85] for security analyses in game-based security models). Variants of the ISO 9798-3 and SIGMA protocols have been analyzed in the UC model in [43,48,49]. These variants assume that protocol participants of the same protocol session already share a globally unique SID and then either use different signing keys for every new session (which is an unrealistic assumption that does not match reality) or, in order to reuse the same key across different sessions, modify the original protocols via a joint state realization that additionally prefixes all signed messages with the global SID. As explained in Section 5.4, such modifications change the security properties of protocols. Hence a faithful analysis of the unmodified key exchange protocols using local SIDs and shared state is necessary and was missing so far.

6. Conclusion

The main goal of this thesis was to propose the first model for universal composability that combines *soundness* of the model with high *flexibility/expressiveness* and good *usability*.

As a major step towards this goal, we have identified and solved the non-responsiveness problem. This problem has led to ill-defined and underspecified protocols that are hard to reuse, flawed security proofs, a lack of expressiveness in modeling certain properties, as well as overly complicated protocol specifications and security proofs where the non-responsiveness problem was actually taken into account. Our model independent solution – the concept of responsive environments – gets rid of the non-responsiveness problem in its entirety and by this improves both *flexibility* and *usability* of the universal composability approach.

We then proposed the *iUC framework* for universal composability. This framework is an instantiation of the IITM model with responsive environments, which is a *sound* and highly *flexible* model that, however, lacks *usability* due to missing modeling tools and conventions. A major challenge in creating iUC was to define conventions in such a way that they are still easy to use yet also do not restrict the flexibility of the IITM model by being tailored towards a specific setting. We have solved this tension by providing a protocol template with many optional and customizable algorithms that come with sensible defaults, allowing protocol designers to focus on the core logic of their protocols by specifying only those parts of the template that are relevant for the protocol at hand. Hence, the iUC framework achieves the overarching goal of this thesis by combining *soundness*, *flexibility*, and *usability* in a currently unmatched way.

We have illustrated both usability and flexibility of the iUC framework via a case study where we model and analyze several real world key exchange protocols. Importantly, we are able to model these protocols exactly as they are deployed in practice; previous universally composable analyses, not based on the iUC or the underlying IITM model, had to resort to analyzing modified versions of the protocols with different security properties due to a lack of flexibility. As part of our case study, we have also provided ideal functionalities that allow for reasoning about cryptographic primitives on an intuitive information theoretic level, instead of having to resort to tedious and error-prone reduction proofs. This independent contribution further improves *usability* of the universal composability approach.

Altogether, we have achieved the main goal of our thesis. Our contributions drastically improve both applicability and accessibility of the universal composability approach in general, thereby providing a solid foundation for the precise design and faithful analysis of a wide variety of different security protocols in various adversarial settings.

A. Dealing With Non-Responsiveness: Queuing of Intermediate Requests with Notifications to the Adversary

In the situation described in Section 3.1.4 and in particular for the queuing approach described there, instead of simply ending the activation of the ideal functionality after queuing an input, one could also choose to send a notification to the adversary for each message that is stored in this way. The adversary would then be expected to send responses to each notification such that the ideal functionality is activated sufficiently often to process every message. To illustrate the complexity and see the disadvantages of this approach, let us first be more precise about the exact implementation. If the ideal functionality, upon receiving some input, wants to send an urgent request, the input is first stored (in some set). Then the ideal functionality not only sends the urgent request itself, but also a notification consisting of the ID of the sender of the original input message and a token associated with this message. Now, every time the ideal functionality receives another input before receiving a response to its urgent request, it will again store this input and send another notification (ID and token) to the adversary. As soon as the ideal functionality receives a response to its urgent request, it stores the data from this response and returns control to the adversary. Now, the adversary is allowed to send responses to the notifications she received, which can be uniquely identified by the token in the notification. As soon as the ideal functionality receives such a response, it will process and then delete the input message associated with the token in the response. Note that it should now be possible to process these inputs because the ideal functionality received the answer to its urgent request before. However, while processing one of these stored messages, the ideal functionality might have to send a second urgent request, leading to yet another set of queued input messages, and so on.

Observe that this gives a lot of additional power to the adversary by allowing him to see the original sender of an input, blocking certain requests by never responding to a notification, and being able to change the order in which messages are processed. These abilities are necessary to prevent the same kind of artificial distinguishing attack presented for the other approaches mentioned in Section 3.1.4. In fact, to prevent the specific attack in our running example, the simulator must know whether it has to simulate an instance of A or B upon input (which is why we have to include the ID), and it has to be able to tell the ideal functionality whether it should first process the stored message for A or the stored message for B . If the ideal

functionality does not allow the latter and instead processes the messages in the order they arrived, an environment can still distinguish the real and the ideal world: if in the above attack the environment first answers the urgent request of B , then the simulator must be able to tell \mathcal{F} to process this request first, although A sent the first request.

Although this approach prevents the simple distinguishing attack mentioned above, it still has three severe drawbacks: First, it is very complex and not very intuitive, especially if ideal functionalities send more than one urgent request. Second, it prevents some artificial distinguishing attacks by giving a lot of additional power to the adversary. This weakens the overall security guarantees provided by an ideal functionality. The adversary gains more information and is now able to use new attacks to potentially distinguish the real and ideal worlds. Third, it is still not generally applicable to any ideal functionality. This is the case, for example, if the ideal functionality has to preserve the order of execution to model its intended task.

The complexity and the weakened security guarantees are probably the reasons this approach does not seem to have been followed in the literature so far.

B. Single Session Security Analysis in iUC

This chapter provides the formal definition of the unbounded self-composition theorem of iUC and then illustrates how this theorem can be used to perform a single-session security analysis, which was explained on a high-level in Section 4.4.2. As part of this chapter, we provide one concrete specification for protocols with disjoint sessions (others are possible), which, thus, can be analyzed by considering just a single-session.

We start by defining (disjoint) protocol sessions of a protocol \mathcal{P} in iUC. For this purpose, we introduce a function σ , called a *protocol session ID (PSID) function*, that groups entities of \mathcal{P} into protocol sessions.⁵⁷ On a high level, the function σ takes as input an entity and assigns it a PSID. A session is then defined via a single PSID $psid$ and encompasses all entities with that $psid$. Intuitively, the sessions of a protocol \mathcal{P} are disjoint (according to σ) if instances accept entities only for a single PSID and send messages only to other entities with the same PSID. Thus, entities cannot share state directly or indirectly with entities from other protocol sessions. We note that the concepts of SIDs and PSIDs are independent of each other: an SID is used to denote multiple runs of some party pid in some role $role$, whereas a PSID denotes the set of all entities that form a global protocol session. While it is possible to define a protocol session $psid$ to contain exactly those entities that share some fixed SID sid (which is the fixed way to deal with sessions in most other model, including in UC and GNUC), a protocol session can also contain entities with multiple different SIDs, say, all SIDs that share a specific prefix or all SIDs belonging to the same party.

More formally, protocol session functions and protocols with disjoint sessions are defined as follows. These notions are derived from the protocol session identifier functions and σ -session versions in the IITM model (cf. Section 3.2.8).

Definition B.1 (PSID function). *A function $\sigma : (\{0, 1\}^*)^3 \rightarrow \{0, 1\}^* \cup \{\perp\}$ is called PSID function if it is computable in polynomial time (in the length of its input).*

Definition B.2 (Protocols with disjoint sessions). *Let σ be a PSID function and let \mathcal{P} be a complete protocol. We say that \mathcal{P} has disjoint sessions (according to σ), or \mathcal{P} is a σ -session protocol, if in all runs of the system $\{\mathcal{E}, \mathcal{P}\}$ (for an arbitrary responsive environment $\mathcal{E} \in \text{Env}(\mathcal{P})$ that interacts with the I/O interfaces of public roles of \mathcal{P} and all network interfaces) the following holds true for every machine M in \mathcal{P} :*

⁵⁷PSID functions in iUC are very similar to PSID functions in the IITM model with responsive environments (cf. Definition 3.21). The main difference is that PSID functions in iUC are defined for entities, instead of tapes and messages.

1. M will never accept (via the **CheckID** algorithm) an entity $(pid, sid, role)$ such that $\sigma(pid, sid, role) = \perp$.
2. If $(pid, sid, role)$ is the first entity that an instance of M accepted, then this instance rejects all following entities $(pid', sid', role')$ where $\sigma(pid, sid, role) \neq \sigma(pid', sid', role')$.
3. Let $(pid, sid, role)$ be the first entity that an instance of M accepted. If this instance sends a message m , then the message is sent in the name of an entity $(pid', sid', role')$ such that $\sigma(pid, sid, role) = \sigma(pid', sid', role')$. Furthermore, if m is sent on a connection to some role $role''$ in \mathcal{P} , then this message is sent to an entity $(pid'', sid'', role'')$ such that $\sigma(pid, sid, role) = \sigma(pid'', sid'', role'')$.

We can analyze a single session of a σ -session protocol \mathcal{P} in isolation to obtain security for an unbounded number of sessions of \mathcal{P} . We use a special type of environment to define such a single session security analysis which, intuitively, is allowed to call at most a single session of \mathcal{P} . Again, the concept of such single-session environments is derived from a similar concept in the IITM model. More formally:

Definition B.3. Let $\mathcal{E} \in \text{Env}(\mathcal{P})$ and let σ be a PSID function. We say that \mathcal{E} is a σ -single-session environment if the following holds true for all systems \mathcal{Q} that can connect to \mathcal{E} and in all runs of the combined system $\{\mathcal{E}, \mathcal{Q}\}$:

Let m be the first message that \mathcal{E} sends on one of its external connections (i.e., that can be connected to \mathcal{Q}). Then m is sent to an entity $(pid, sid, role)$ such that $\sigma(pid, sid, role) \neq \perp$. Furthermore, every following message m' on an external connection of \mathcal{E} is addressed to an entity $(pid_1, sid_1, role_1)$ such that $\sigma(pid, sid, role) = \sigma(pid_1, sid_1, role_1)$.

We denote the set of all σ -single-session (responsive and universally bounded) environments for a protocol \mathcal{P} by $\text{Env}_{\sigma\text{-single}}(\mathcal{P})$.

We can now define the single session realization relation and state the unbounded self-composition theorem in iUC (an informal version of this theorem was given as Corollary 4.3 in Section 4.4.2). Both the realization relation and the composition theorem are natural translations of the corresponding statements in the IITM model.

Definition B.4 (Single session realization relation). Let σ be a PSID function, and let \mathcal{P} and \mathcal{F} be two environmentally bounded complete σ -session protocols with identical sets of public roles. We say that \mathcal{P} single-session realizes \mathcal{F} ($\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$) if there exists a simulator $\mathcal{S} \in \text{Adv}(\mathcal{F})$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ for all $\mathcal{E} \in \text{Env}_{\sigma\text{-single}}(\mathcal{P})$.⁵⁸

Corollary B.1 (Unbounded self-composition theorem). Let σ be a PSID function, and let \mathcal{P} and \mathcal{F} be two protocols such that $\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$. Then $\mathcal{P} \leq \mathcal{F}$.

⁵⁸Note that both \mathcal{F} and \mathcal{P} use the same PSID function σ to define disjoint sessions. Thus, they agree in their “behavior” for the shared public roles, i.e., entities of those roles are grouped into protocol sessions in the same way.

Proof. This follows from the unbounded self-composition theorem in the IITM model (cf. Theorem 3.4). However, unlike Corollary 4.1, this corollary is not a trivial instantiation but rather requires a short argument. This is because the definitions of PSID functions and σ -session protocols in iUC are slightly different from how they are defined in the IITM model, so we have to relate the notions from iUC to those in the IITM model. We do so in the following.

Let σ be an PSID function as defined for iUC in Definition B.1. Suppose we have two (iUC) protocols \mathcal{P} and \mathcal{F} that are σ -session protocols such that $\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$. In the following, we want to use the unbounded self-composition theorem of the IITM model (cf. Theorem 3.4) to conclude that $\mathcal{P} \leq \mathcal{F}$. For this purpose, we have to define a PSID function $\tilde{\sigma}$ in the sense of the IITM model (cf. Definition 3.21), show that \mathcal{P} and \mathcal{F} are $\tilde{\sigma}$ -session protocols in the sense of the IITM model (cf. Definition 3.22), and show that $\mathcal{P} \leq_{\tilde{\sigma}\text{-single}} \mathcal{F}$ (again, in the sense of the IITM model). An important difficulty here is that a “ σ -session protocol” is a property defined for runs of the whole protocol (in some arbitrary responsive environment), whereas “ $\tilde{\sigma}$ -session protocol” is a property defined for runs of individual machines of the protocol (in some arbitrary context that might not even be responsive or runtime bounded). Thus, formally, some properties that hold true in the context of the whole protocol, might no longer be true if the protocol is broken apart. Dealing with this issue is the main obstacle of this proof.

Let us begin by summarizing the major differences between σ and $\tilde{\sigma}$ as well as σ -session protocols and $\tilde{\sigma}$ -session protocols in iUC and the IITM model, respectively. Firstly, σ is defined on entities, whereas $\tilde{\sigma}$ takes as input a message and a named tape and then determines the PSID of the machine instance that sent/received this message. Thus, we have to define $\tilde{\sigma}$ such that it uses the header information contained in messages in iUC (cf. Appendix D.2.3) and the tape to determine the entity that receives some message and then evaluate σ for that entity. Note that importantly, for internal tapes connecting two roles of \mathcal{P} , the output of $\tilde{\sigma}$ is not only required to match the PSID of the receiving entity but it must also match the PSID of the sending entity (as otherwise the sender would send a message to another session, i.e., \mathcal{P} would not be a $\tilde{\sigma}$ -session protocol).

We define $\tilde{\sigma}(m, t)$ as follows:

- If t is an external output tape of \mathcal{P}/\mathcal{F} (i.e., connecting to the environment or adversary), then compute the sending entity $(pid_{snd}, sid_{snd}, role_{snd})$. That is, parse m to obtain $pid[snd]$ and $sid[snd]$ of the sending entity and compute $role[snd]$ (if parsing m does not work, e.g., due to an invalid header format, set $\tilde{\sigma}(m, t) := \perp$). Output the PSID of the sending entity, i.e., $\tilde{\sigma} := \sigma(entity_{snd})$.
- If t is an external input tape of \mathcal{P}/\mathcal{F} (i.e., connecting from the environment or adversary) or an internal tape of \mathcal{P}/\mathcal{F} (i.e., connecting two machines of \mathcal{P}/\mathcal{F}), then compute the receiving entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$. That is, parse m to obtain $pid[rcv]$ and $sid[rcv]$ of the receiving entity and compute $role[rcv]$ (if parsing m does not work, e.g., due to an invalid header format, set $\tilde{\sigma}(m, t) := \perp$). Output the PSID of the receiving entity, i.e., $\tilde{\sigma} := \sigma(entity_{rcv})$.

- For tapes t that are not part of \mathcal{P}/\mathcal{F} , set $\tilde{\sigma}(m, t) := \perp$.

Observe that $\tilde{\sigma}$ is indeed a session function as it is efficiently computable. The protocols \mathcal{P} and \mathcal{F} are *almost* $\tilde{\sigma}$ -session protocols for the above definition. Consider the behavior of individual machines in runs of the whole protocol with an arbitrary (not necessarily single-session) responsive environment. Firstly, observe that no machine in \mathcal{P}/\mathcal{F} accepts messages where $\tilde{\sigma}$ is \perp as then either the header is malformed or σ is also \perp . Secondly, if an instance has accepted a message with some PSID according to $\tilde{\sigma}$, then it will not accept messages for any other PSIDs, as this would imply that it accepts two entities with different PSIDs according to σ . Thirdly, messages sent by an instance have the same PSID according to $\tilde{\sigma}$ as those that were previously accepted. For external tapes, this directly follows from the fact that the sending entity must have the correct PSID according to σ . For internal tapes, this is implied by the additional requirement that the receiving entity also has the correct PSID according to σ .

So overall, the properties of machines of $\tilde{\sigma}$ -session protocols hold true but only for the specific context of the whole protocol running with a responsive environment. However, we need those properties to also hold true when running individual machines in an arbitrary context, which is not the case in general.⁵⁹ Thus, we have to modify \mathcal{P} and \mathcal{F} slightly. The new protocols $\tilde{\mathcal{P}}$ and $\tilde{\mathcal{F}}$ are the same as before, except for the following changes made to each machine: Before accepting a message in mode `CheckAddress`, the machine first checks that the properties of $\tilde{\sigma}$ session protocols are not violated and rejects the message otherwise. Furthermore, before sending a message, the machine again checks that the properties of $\tilde{\sigma}$ -session protocols are fulfilled and aborts the activation without sending the message otherwise. Thus, we have that $\tilde{\mathcal{P}}$ and $\tilde{\mathcal{F}}$ are $\tilde{\sigma}$ -session protocols and, by the above observations, they behave identical to \mathcal{P} and \mathcal{F} when running the whole protocol in a responsive environment. Note that both $\tilde{\mathcal{P}}$ and $\tilde{\mathcal{F}}$ are still environmentally bounded as in particular $\tilde{\sigma}$ is efficiently computable.

Now let $\mathcal{E} \in \text{Env}_{\tilde{\sigma}\text{-single}}(\mathcal{P})$ be a single-session environment according to $\tilde{\sigma}$. We have that \mathcal{E} sends only messages m on tape t such that $\tilde{\sigma}(m, t)$ always outputs the same PSID, say, $psid$. By definition of $\tilde{\sigma}$, those messages are always sent to entities who have PSID $psid$ according to σ . Thus we have $\mathcal{E} \in \text{Env}_{\sigma\text{-single}}(\mathcal{P})$. As $\mathcal{P} \leq_{\sigma\text{-single}} \mathcal{F}$ by assumption, we have that there exists a simulator \mathcal{S}_{single} such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}_{single}, \mathcal{F}\}$. As \mathcal{P} and $\tilde{\mathcal{P}}$ as well as \mathcal{F} and $\tilde{\mathcal{F}}$ behave identical in runs with arbitrary responsive environments, we have that $\{\mathcal{E}, \tilde{\mathcal{P}}\} \equiv \{\mathcal{E}, \mathcal{S}_{single}, \tilde{\mathcal{F}}\}$. In summary, this implies $\tilde{\mathcal{P}} \leq_{\tilde{\sigma}\text{-single}} \tilde{\mathcal{F}}$.

We can now apply the unbounded self-composition theorem of the IITM model (cf. Theorem 3.4) to conclude that $\tilde{\mathcal{P}} \leq \tilde{\mathcal{F}}$. By the same argument as above, using that $\tilde{\mathcal{P}}$ and \mathcal{P} as well as $\tilde{\mathcal{F}}$ and \mathcal{F} behave identical for arbitrary responsive environments, this implies $\mathcal{P} \leq \mathcal{F}$. \square

⁵⁹For example, a machine M that is used only as an internal subroutine within a protocol might, upon being activated by a sender entity from a higher-level protocol/role of the same protocol, compute some response and return this response to the sending entity. In the context of the whole protocol, the sender entity is always of the same session as the receiver entity in M , and hence M would also return the message to an entity in the same session. However, when running in an arbitrary context that can claim arbitrary sender entities, the context might choose a sender entity that is from a different session than the receiver entity of M . Thus, M would return/send a message to an entity in a different session, violating property 3 of $\tilde{\sigma}$ -session protocols.

We now illustrate how protocols with disjoint sessions can be modeled in iUC by giving an example. More specifically, we model the standard case that is considered in the UC and GNUC models where a single session of a single highest-level protocol is analyzed in isolation. This single session can use potentially several instances of arbitrary subroutines, as long as no instance is accessible by two or more different sessions. In our framework, we model this setting by considering a combined protocol $\mathcal{R} := \mathcal{P} \parallel \mathcal{S}_1 \parallel \dots \parallel \mathcal{S}_n$ consisting of a highest-level protocol \mathcal{P} with several subroutine protocols \mathcal{S}_i .

We define a protocol session of \mathcal{R} via the SID used by entities in the highest-level protocol \mathcal{P} . That is, an entity $(pid, sid, role)$ of \mathcal{P} runs in the protocol session $psid := sid$. The SIDs of entities in subroutines consist of two parts, a prefix and a suffix, where the prefix is the actual protocol session that they run in and the suffix allows for arbitrarily many copies of a subroutine within the same session. That is, an entity of \mathcal{S}_i has the form $(pid, (sid_{pre}, sid_{suf}), role)$ and runs in session $psid := sid_{pre}$. This directly implies a definition of a PSID function σ which, in particular, is computable in polynomial time: $\sigma(pid, sid, role)$ checks whether $role$ is in \mathcal{P} or \mathcal{S}_i and then either outputs sid or the prefix of sid . In all other cases (e.g., if there is no prefix in the SID in case of a subroutine) σ outputs \perp .

Now, (instances of) machines in \mathcal{R} have to meet the three properties of Definition B.2 in order to have disjoint sessions: (i) they may not accept entities that belong to no protocol session (i.e., where σ outputs \perp), (ii) they never accept entities from two different protocol sessions, and (iii) senders of messages are in the correct session and receivers, if they are part of \mathcal{R} , are in the same session. We ensure these properties as follows:

- (i) The **CheckID** algorithm is used to ensure that the SIDs of entities in subroutines have the expected format. That is, subroutine machines accept only entities that have a prefix in their SID. Thus, no machine in \mathcal{R} accepts an entity that does not have a PSID.
- (ii) This can also be enforced via the **CheckID** algorithm. More specifically, a machine saves the first entity that it has accepted and then accepts following entities only if they have the same PSID as the first one. That is, they either have the same SID (in the case of entities in \mathcal{P}) or the same SID prefix (in the case of entities in \mathcal{S}_i).
- (iii) This property is straightforward to ensure via suitable definitions of the various algorithms in our template. More specifically, every **send** command in each of those algorithms must be defined such that senders and receivers of this message meet this condition (note that this includes the macros from Section 4.2.6, which internally send messages). Furthermore, we use the **AllowAdvMessage** algorithm to prevent the adversary from breaking this condition for corrupted entities.

We provide formal definitions of the components of \mathcal{R} following these guidelines in Figures B.1 and B.2.

We directly obtain that a protocol \mathcal{R} that is constructed as described above (cf. Figures B.1, B.2) is a σ -session protocol. Thus we can use Corollary B.1 to obtain the following:

Structure of a highest-level protocol \mathcal{P} (used in a combined protocol $\mathcal{P} || \mathcal{S}_1 || \dots || \mathcal{S}_n$) with disjoint sessions:

<p>Participating roles: <i>arbitrary</i> Corruption model: <i>arbitrary</i></p>
<p>For each of the machines M of \mathcal{P}:</p>
<p>Implemented role(s): <i>arbitrary</i> CheckID($pid, sid, role$): Perform <i>arbitrary</i> checks and, potentially, output reject based on these checks. If no other entity has been accepted yet, output accept. Otherwise, let $sid^{accepted}$ be the (full) SID of the first entity that was accepted. Output accept if and only if $sid^{accepted} = sid$. Corruption behavior: – AllowAdvMessage($pid, sid, role, pid_{receiver}, sid_{receiver}, role_{receiver}, m$): If $role_{receiver}$ is part of \mathcal{P} or a higher-level protocol/the environment, then check that $sid = sid_{receiver}$. Otherwise, try to parse as $sid_{receiver}$ as (sid_{prefix}, sid') and check that $sid = sid_{prefix}$. <div style="text-align: right; font-size: small;"><i>{i.e., the role is specified in \mathcal{S}_i.</i></div> If any of the previous steps/checks fails, output false. <div style="text-align: center; font-size: small;"><i>{Ensure that messages are sent only to the same “session”.</i></div> Perform <i>arbitrary</i> other checks and output true or false based on these checks. Other Corruption behavior, initialization, and core logic algorithms: These algorithms are <i>arbitrary</i>, but subject to the restriction that they may only send messages from entities managed by the current instance^a to entities that are part of the same “session”. In particular, messages must have a correct header (cf. Appendix D.2.3; note that protocol designers using our syntax from Section 4.2.6 need not care about headers as they are automatically added). Furthermore, if a message is sent to a higher-level protocol or a role in \mathcal{P}, then $sid_{sender} = sid_{receiver}$, and if it is sent to a subroutine role in one of the subroutine protocols \mathcal{S}_i, then $sid_{receiver} = (sid_{sender}, sid')$ (for messages sent to the network there is not restriction imposed on the receiver). <hr style="width: 20%; margin-left: 0;"/> ^ai.e., from entities that get accepted by CheckID.</p>

Figure B.1.: Example structure of a highest-level protocol with disjoint-sessions. Fields/algorithms that are marked as *arbitrary* or that are omitted can be specified freely by the protocol designer without breaking disjoint sessions. See Figure B.2 for how subroutine protocols \mathcal{S}_i are defined.

Let \mathcal{R} and \mathcal{I} be two σ -session protocols that are constructed as described above (for the same σ). If $\mathcal{R} \leq_{\sigma\text{-single}} \mathcal{I}$ then $\mathcal{R} \leq \mathcal{I}$. That is, it is sufficient to analyze and compare a single protocol session of \mathcal{R} with a single protocol session of \mathcal{I} (and a simulator) to obtain security for arbitrarily many sessions running concurrently.

We note that, in the above theorem, we did not fix which roles are public and private in \mathcal{R}/\mathcal{I} , i.e., this argument also works even if some of the subroutine protocols have public roles that the environment can access. For example, consider a protocol \mathcal{P} that uses a subroutine \mathcal{S}_1 that provides common reference strings (CRSs), where a different independent CRS is provided for different sessions of \mathcal{P} . We can make the subroutine \mathcal{S}_1 public in the combined protocol \mathcal{R} , modeling globally available CRSs that can also be used by other protocols independently of \mathcal{P} . Even in this setting with global state, we can still apply Corollary B.1 to be able to analyze just a single session of \mathcal{R} , i.e., one session of \mathcal{P} using a single global CRS provided by \mathcal{S}_1 .

Structure of subroutine protocols \mathcal{S}_i (used in a combined protocol $\mathcal{P} \parallel \mathcal{S}_1 \parallel \dots \parallel \mathcal{S}_n$) with disjoint sessions:

Participating roles: *arbitrary*
Corruption model: *arbitrary*

For each of the machines M of \mathcal{S}_i :

Implemented role(s): *arbitrary*

CheckID($pid, sid, role$):

Check that $sid = (sid_{prefix}, sid')$; otherwise output **reject**.

Perform *arbitrary* additional checks and, potentially, output **reject** based on these checks.

If not other entity has been accepted yet, output **accept**.

Otherwise, let $sid_{prefix}^{accepted}$ be the prefix of the SID of the first entity that was accepted.

Output **accept** if and only if $sid_{prefix}^{accepted} = sid_{prefix}$.

Corruption behavior:

– **AllowAdvMessage**($pid, sid, role, pid_{receiver}, sid_{receiver}, role_{receiver}, m$):

Parse sid as (sid_{prefix}, sid') .

If $role_{receiver}$ is part of \mathcal{P} , then check that $sid_{prefix} = sid_{receiver}$.

Otherwise, try to parse $sid_{receiver}$ as (sid_{prefix}, sid'') .

{i.e., the role is specified in some \mathcal{S}_j .

If any of the previous steps/checks fails, output **false**.

{Ensure that messages are sent only to the same “session”.

Perform *arbitrary* other checks and output **true** or **false** based on these checks.

Other Corruption behavior, initialization, and core logic algorithms:

These algorithms are *arbitrary*, but subject to the restriction that they may only send messages from entities managed by the current instance^a to entities that are part of the same “session”.

In particular, messages must have a correct header (cf. Appendix D.2.3; note that protocol designers using our syntax from Section 4.2.6 need not care about headers as they are automatically added).

Furthermore, if a message is sent to a role in \mathcal{P} , then $sid_{sender} = (sid_{receiver}, sid')$, and if it is sent to a subroutine role in one of the subroutine protocols \mathcal{S}_i , then the prefixes of sid_{sender} and $sid_{receiver}$ are identical (for messages sent to the network there is not restriction imposed in the receiver).

^ai.e., from entities that get accepted by **CheckID**.

Figure B.2.: Example structure of subroutine protocols with disjoint-sessions. Fields/algorithms that are marked as *arbitrary* or that are omitted can be specified freely by the protocol designer without breaking disjoint sessions. See Figure B.1 for how the highest-level protocol \mathcal{P} is defined.

C. Example: Joint State Realization in iUC

This chapter illustrates how one can use iUC to model and analyze joint state realizations (see Section 4.4.2 for a general explanation of joint state composition). We do so by means of an example: We consider an ideal signature functionality \mathcal{F}_{sig} with disjoint sessions, i.e., where each session (defined via the SID of entities) uses an independent key pair. We want to realize this functionality with a joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ where the same key pair is reused across multiple sessions.

The ideal signature functionality \mathcal{F}_{sig} is given in Figure C.2. It is analogous to existing signature functionalities from the literature (e.g., [94]) and almost identical to the functionality $\mathcal{F}_{\text{sig-CA}}$ from Section 4.3, except that it does not register verification keys in an ideal \mathcal{F}_{CA} subroutine. SIDs in \mathcal{F}_{sig} are of the form $(pid_{\text{owner}}, sid')$, where pid_{owner} is the owner of a key and sid' denotes a specific key pair of pid_{owner} . In particular, for each sid' there is a separate independent key pair.

In our joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ of \mathcal{F}_{sig} , we want to reuse the same signing key for all sessions of a key owner, i.e., every party pid_{owner} only has a single signing key for all sid' . We achieve this as follows: $\mathcal{P}_{\text{sig}}^{\text{js}}$ (given in Figures C.3 and C.4 at the end of this chapter) implements the **signer** and **verifier** roles, which act as multiplexers for a subroutine \mathcal{F}_{sig} . For each party pid , there is one session of the subroutine \mathcal{F}_{sig} with fixed SID (pid, ϵ) . We define the **CheckID** algorithm of the **signer** and **verifier** machines to accept all entities belonging to the same party, i.e., an instance of the **signer/verifier** machines models one party in all sessions (as is common for typical joint state realizations). If a request for signing a message m in session/with key (pid, sid') is received, then $\mathcal{P}_{\text{sig}}^{\text{js}}$ instead prefixes m with sid' and uses the subroutine \mathcal{F}_{sig} in session (pid, ϵ) to sign (sid', m) ; the response is then returned. Prefixing messages with sid' ensures that, even though the same signing key is used, signatures on messages for different SIDs sid' are still independent, just as in the ideal world for \mathcal{F}_{sig} . This is a standard technique often employed for obtaining joint state realizations; in fact, our realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ is directly based on the joint state realization for digital signatures proposed in [94]. We obtain the following result for $\mathcal{P}_{\text{sig}}^{\text{js}}$:

Lemma C.1. *Let \mathcal{F}_{sig} and $\mathcal{P}_{\text{sig}}^{\text{js}}$ be as above. Then:*

$$(\mathcal{P}_{\text{sig}}^{\text{js}} \mid \mathcal{F}_{\text{sig}}) \leq (\mathcal{F}_{\text{sig}}).$$

Proof. Analogous to [94]. □

Hence, we can replace \mathcal{F}_{sig} used by some protocol \mathcal{Q} (which might have disjoint sessions, each

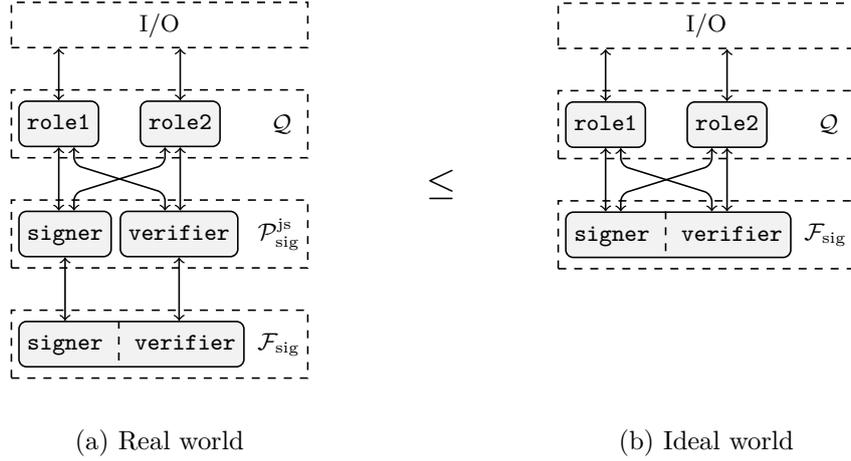


Figure C.1.: An example of combined protocols that are covered by Corollary C.1: a higher-level protocol \mathcal{Q} consisting of two roles using the ideal signature functionality with public key infrastructure \mathcal{F}_{sig} (right side) respectively its joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ (left side). Note that this is just one specific example of a protocol that is covered by Corollary C.1. One can also, e.g., consider a combined protocol where the $\mathcal{P}_{\text{sig}}^{\text{js}} : \text{verifier}$ subroutine role (respectively $\mathcal{F}_{\text{sig}} : \text{verifier}$ in the ideal world) is public such that other protocols can also verify signatures.

of them using separate signature key pairs) with its joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ via our general composition theorem:

Corollary C.1. *Let \mathcal{Q} be a protocol such that \mathcal{Q} and \mathcal{F}_{sig} are connectable. Furthermore, let $\mathcal{R} \in \text{Comb}(\mathcal{Q}, (\mathcal{P}_{\text{sig}}^{\text{js}} \mid \mathcal{F}_{\text{sig}}))$ and let $\mathcal{I} \in \text{Comb}(\mathcal{Q}, \mathcal{F}_{\text{sig}})$ such that \mathcal{R} and \mathcal{I} have the same sets of public roles. If \mathcal{R} is environmentally bounded, then*

$$\mathcal{R} \leq \mathcal{I} \quad .$$

Proof. This is an application of the main composition theorem (cf. Corollary 4.1; see also Corollary 4.4 which formulates the special case of the main composition theorem for joint state protocols). \square

See Figure C.1 for an example application of Corollary C.1. Let us highlight a few crucial features of iUC that we used above to model and compose the joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$:

- The flexible addressing mechanism **CheckID**, which allows for dynamically mapping entities to machine instances, allows us to easily create machine instances that manage/model a party in all sessions, as done for $\mathcal{P}_{\text{sig}}^{\text{js}}$. In particular, one can then share the same state for that party throughout all sessions.
- Our framework does not impose any restrictions on how different machines are connected and how entities of different machines are allowed to communicate. Hence, entities in $\mathcal{P}_{\text{sig}}^{\text{js}}$ that have different SIDs (but belong to the same party pid) can directly access the same signer entity (with fixed SID (pid, ϵ)) in the subroutine \mathcal{F}_{sig} . Similar to the previous point, this feature allows for easily sharing state between multiple sessions.

- The main composition theorem is independent of (the interpretation of) SIDs and in particular does not require protocols to have disjoint session. Instead, the theorem supports a general protocol definition, where both protocols with and without joint state (such as $\mathcal{P}_{\text{sig}}^{\text{js}}$ and \mathcal{F}_{sig}) are mere special case. Hence, we can directly apply the theorem also to such a joint state realization.
- Our corruption model allows the simulator in the ideal world to corrupt fresh entities as soon as they are created, while non-existing entities always output `false` when asked for their corruption status. As already explained in detail in Section 4.4.2, this is crucial to be able to prove Lemma C.1: Once the single key of a party pid in the real world has been corrupted, this corresponds to infinitely many corrupted sessions of \mathcal{F}_{sig} in the ideal world. Hence, the simulator needs a way to perform those corruptions even though he has only polynomial runtime. iUC addresses this tension by providing a reactive mechanism, i.e., corruption can occur as soon as an entity is first triggered and does not have to be performed pro-actively for non-existing entities.

Description of the protocol $\mathcal{F}_{\text{sig}} = (\text{signer}, \text{verifier})$:

Participating roles: $\{\text{signer}, \text{verifier}\}$	
Corruption model: dynamic with secure erasures	
Protocol parameters:	$\left\{ \begin{array}{l} \text{Polynomial that bounds the runtime of the algo-} \\ \text{rithms provided by the adversary.} \end{array} \right.$
- $p \in \mathbb{Z}[x]$.	

Description of $M_{\text{signer}, \text{verifier}}$:

Implemented role(s): $\{\text{signer}, \text{verifier}\}$	
Internal state:	
- $(\text{sig}, \text{ver}, \text{pk}, \text{sk}) \in (\{0, 1\}^* \cup \{\perp\})^4 = (\perp, \perp, \perp, \perp)$.	$\{ \text{Algorithms and key pair.} \}$
- $\text{pidowner} \in \{0, 1\}^* \cup \{\perp\} = \perp$.	$\{ \text{Party ID of the key owner.} \}$
- $\text{msglist} \subseteq \{0, 1\}^* = \emptyset$.	$\{ \text{Set of recorded messages.} \}$
- $\text{KeysGenerated} \in \{\text{ready}, \perp\} = \perp$.	$\{ \text{Has signer initialized his key?} \}$
CheckID ($\text{pid}, \text{sid}, \text{role}$):	
Check that $\text{sid} = (\text{pid}', \text{sid}')$.	
If this check fails, output reject .	$\left\{ \begin{array}{l} \text{A single instance manages all} \\ \text{parties and roles in a single} \\ \text{session.} \end{array} \right.$
Otherwise, accept all entities with the same SID.	
Corruption behavior:	
- LeakedData ($\text{pid}, \text{sid}, \text{role}$): If $(\text{pid}, \text{sid}, \text{role})$ determines its initial corruption status, use the default behavior of LeakedData .	
Otherwise, if $\text{role} = \text{signer}$ and $\text{pid} = \text{pidowner}$, return KeysGenerated . In all other cases return \perp .	
Initialization:	
send responsively InitMe to NET;	
wait for (Init , ($\text{sig}, \text{ver}, \text{pk}, \text{sk}$)).	
$(\text{sig}, \text{ver}, \text{pk}, \text{sk}) \leftarrow (\text{sig}, \text{ver}, \text{pk}, \text{sk})$.	
Parse sid_{cur} as (pid, sid) .	
$\text{pidowner} \leftarrow \text{pid}$.	
Main:	
rcv InitSign from I/O to $(\text{pidowner}, _, \text{signer})$:	$\left\{ \begin{array}{l} \text{Successful initialization. Note that} \\ \text{signer can submit InitSign multiple} \\ \text{times, always with the same effect.} \end{array} \right.$
KeysGenerated \leftarrow ready. reply (InitSign , success, pk).	
rcv (Sign , m) from I/O to $(\text{pidowner}, _, \text{signer})$ s.t. KeysGenerated = ready:	
$\sigma \leftarrow \text{sig}^{(p)}(m, \text{sk})$.	
$b \leftarrow \text{ver}^{(p)}(m, \sigma, \text{pk})$.	$\{ \text{Sign and check that verification succeeds.} \}$
if $\sigma = \perp \vee b \neq \text{true}$:	
reply (Signature , \perp).	$\{ \text{Signing or verification test failed.} \}$
else:	
add m to msglist.	
reply (Signature , σ).	$\{ \text{Record } m \text{ for verification and return signature.} \}$
rcv (Verify , m, σ, pk) from I/O to $(_, _, \text{verifier})$:	
$b \leftarrow \text{ver}^{(p)}(m, \sigma, \text{pk})$.	$\{ \text{Verify signature.} \}$
if $\text{pk} = \text{pk} \wedge b = \text{true} \wedge m \notin \text{msglist} \wedge (\text{pidowner}, \text{sid}_{\text{cur}}, \text{signer}) \notin \text{CorruptionSet}$:	
reply (VerResult , false).	$\{ \text{Prevent forgery.} \}$
else:	
reply (VerResult , b).	$\{ \text{Return verification result.} \}$

Figure C.2.: The ideal signature protocol \mathcal{F}_{sig} .

Description of the protocol $\mathcal{P}_{\text{sig}}^{\text{js}} = (\text{signer}, \text{verifier})$:

Participating roles: signer, verifier	
Corruption model: dynamic with secure erasures	
Protocol parameters:	
– $p \in \mathbb{Z}[x]$.	$\left\{ \begin{array}{l} \text{Polynomial that bounds the runtime of the algo-} \\ \text{rithms provided by the adversary.} \end{array} \right.$

Description of M_{signer} :

Implemented role(s): signer	
Subroutines: $\mathcal{F}_{\text{sig}} : \text{signer}$	
Internal state:	
– $\text{initedSids} \subseteq \{0, 1\}^* \cup \{\perp\} = \emptyset$.	$\left\{ \begin{array}{l} \text{Which sessions have already sent an} \\ \text{InitSign request?} \end{array} \right.$
CheckID ($pid, sid, role$):	
Check that $sid = (pid', sid')$; otherwise output reject .	
Accept all entities with the same PID.	$\left\{ \text{Accept one party in all sessions.} \right.$
Corruption behavior:	
– DetermineCorrStatus ($pid, sid, role$): Output corr ($pid, (pid, \epsilon), \mathcal{F}_{\text{sig}} : \text{signer}$).	
– AllowCorruption ($pid, sid, role$): Output corr ($pid, (pid, \epsilon), \mathcal{F}_{\text{sig}} : \text{signer}$).	$\left\{ \begin{array}{l} \text{Allow corruption only if the corresponding subrou-} \\ \text{tine entity of } \mathcal{F}_{\text{sig}} \text{ has been corrupted.} \end{array} \right.$
MessagePreprocessing:	
recv m from I/O:	
Try to parse sid_{cur} as $(\text{pid}_{\text{cur}}, \text{sid}')$.	$\left\{ \begin{array}{l} \text{Signers can perform actions only in sessions where} \\ \text{they own the key, i.e., where their PID is a prefix} \\ \text{of the SID. Otherwise, do nothing.} \end{array} \right.$
if parsing fails:	
abort .	
Main:	$\left\{ \begin{array}{l} \text{Due to MessagePreprocessing we have that, for} \\ \text{I/O requests, the current entity is of the form} \\ \text{(pid, (pid, sid'), signer).} \end{array} \right.$
recv InitSign from I/O:	
send InitSign to $(\text{pid}_{\text{cur}}, (\text{pid}_{\text{cur}}, \epsilon), \mathcal{F}_{\text{sig}} : \text{signer})$;	
wait for (InitSign , success , pk).	
Add sid_{cur} to initedSids .	
reply (InitSign , success , pk).	
recv (Sign , m) from I/O s.t. $\text{sid}_{\text{cur}} \in \text{initedSids}$:	
send (Sign , $(\text{sid}_{\text{cur}}, m.sg)$) to $(\text{pid}_{\text{cur}}, (\text{pid}_{\text{cur}}, \epsilon), \mathcal{F}_{\text{sig}} : \text{signer})$;	$\left\{ \begin{array}{l} \text{Prefix message with the current SID to obtain dis-} \\ \text{joint message spaces for different sessions.} \end{array} \right.$
wait for (Signature , σ).	
reply (Signature , σ).	

Figure C.3.: The joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ of \mathcal{F}_{sig} (part I). It uses a single instance of \mathcal{F}_{sig} as a subroutine to realize multiple instances (in different sessions) of \mathcal{F}_{sig} .

Description of M_{verifier} :

Implemented role(s): verifier
Subroutines: \mathcal{F}_{sig}
CheckID($pid, sid, role$):
 Check that $sid = (pid', sid')$; otherwise output **reject**.
 Accept all entities with the same PID. { Accept one party in all sessions. }

Corruption behavior:

- **DetermineCorrStatus**($pid, sid, role$): Parse sid as $(pid_{\text{signer}}, sid')$.
 Output $\text{corr}(pid_{\text{signer}}, (pid_{\text{signer}}, \epsilon), \mathcal{F}_{\text{sig}} : \text{signer})$
 $\vee \text{corr}(pid, (pid_{\text{signer}}, \epsilon), \mathcal{F}_{\text{sig}} : \text{verifier})$. { If either of these two entities
has been corrupted, then the
adversary can forge messages. }
- **AllowAdvMessage**($pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{\text{receiver}}, m$):
 Check that $pid = pid_{\text{receiver}}$ and $role_{\text{receiver}} \neq \mathcal{F}_{\text{sig}} : \text{signer}$.
 If all checks succeed, output **true**, otherwise output **false**

Main:

recv (Verify, m, σ, pk) **from** I/O:
 Parse sid_{cur} as $(pid_{\text{signer}}, sid')$.
send (Verify, $(sid_{\text{cur}}, m), \sigma, pk$) **to** $(pid_{\text{cur}}, (pid_{\text{signer}}, \epsilon), \mathcal{F}_{\text{sig}} : \text{verifier})$;
{ Prefix message with the current SID to obtain disjoint message
spaces for different sessions. }

wait for (VerResult, b).
reply (VerResult, b).

Figure C.4.: The joint state realization $\mathcal{P}_{\text{sig}}^{\text{js}}$ of \mathcal{F}_{sig} (part II). It uses a single instance of \mathcal{F}_{sig} as a subroutine to realize multiple instances (in different sessions) of \mathcal{F}_{sig} .

D. Mapping iUC Protocols to the IITM Model

In this chapter, we explain how the protocols specified in Section 4.2 are mapped to actual systems in the sense of the IITM model with responsive environments. The resulting systems are instantiations of the protocol systems in the IITM model with responsive environment (see Section 3.2). Hence, all theorems of the responsive IITM model, including composition theorems, hold true for these systems. This, in particular, shows that iUC inherits the soundness properties of the IITM model.

This chapter is structured as follows: First, we introduce a low-level syntax for describing IITMs in Appendix D.1. Then, in Appendix D.2, we explain how a single instance of our template from Section 4.2.3 is mapped to a system of IITMs, including the exact specification of those IITMs. Finally, in Appendix D.3 we explain how a complete protocol, which is defined by one or more instances of our template, is interpreted as a system of IITMs (this mainly entails connecting the individual systems created from each template instance).

D.1. Notation for the Formal Specification of IITMs

Before being able to formally specify how protocol systems in the sense of the IITM model are obtained from the specifications/templates, we need to introduce some notation for specifying IITMs. This notation is used to formally define the IITMs that are obtained from our template, where by IITMs we mean the (plain) IITMs introduced in Section 3.2. We note that this is a rather low-level syntax that need not be used by a protocol designer but is rather only used for presenting the mapping of our template to IITMs. For specifying algorithms in our template, we provide a convenient high-level syntax in Section 4.2.6. Note that the following low-level syntax borrows and adjusts several elements from the high-level syntax such as message patterns.

Message patterns: A message pattern mp is used to describe the format of a message $m \in \{0, 1\}^*$. It is built from *local variables* (denoted in italic font) which only exist for a single activation of an algorithm, **global variables** (denoted in sans-serif font) which are part of the internal state of an instance of a machine and can be accessed across multiple activations of different algorithms of the same instance, **strings** (denoted in typewriter font), and special characters such as “(”, “)”, “,” and “ \perp ”.

Message patterns can be used to describe outgoing messages, in the following denoted by mp_{out} , and incoming messages, in the following denoted by mp_{in} . If a message pattern is used for sending, the current values of global and local variables are inserted, while the remainder of

the pattern stays as is (in particular, strings and special signs are not altered). The resulting message is then sent. If a message pattern is used for receiving, a message m upon receipt is matched against the pattern: After inserting the values of global variables and, if already defined, those of local variables into mp_{in} , the resulting message must be the same as m except for undefined local variables, which match an arbitrary text. After a successful match, all local variables contain the value that they matched on. The special symbol $_$ can be used in mp_{in} instead of an undefined local variable if the value that is matched on is not needed afterwards, i.e., $_$ matches everything but does not store the result.

To illustrate message patterns, consider the case where an instance of an IITM has a global variable id storing the ID of the instance. Such an instance might at some point send a request on the network to the adversary which contains a unique request ID qid , which is stored in a local variable (as it is no longer needed once the algorithm has terminated). Such an ID is useful to match responses from the adversary to specific requests. Now, the message pattern $m_{\text{in}} = (id, (\text{Response}, qid, m'))$ can be used to wait for a response from the adversary. This pattern will match any message m which contains the ID id of the instance, the fixed bit string **Response**, the value contained in the local variable qid , and an arbitrary bit string which will be stored in a new local variable m' after a successful match.

Sending raw messages: When we write **send** mp_{out} **on** t , we mean that the message m that is created from mp_{out} at runtime is sent on tape t .

Restricting messages: As introduced in Section 3.2, we consider a restriction relation R and responsive environments such that if a real or ideal protocol outputs a restricting message $x \in R[0]$ on a network tape, the environment/adversary/simulator has to send a reply y on the corresponding tape with $(x, y) \in R$ immediately, i.e., without sending an incorrect message (wrong message according to R or wrong tape) to the protocol before. To be precise, we use the following definition of R :

$$\begin{aligned}
R := & \{(m, m') \mid m = (id, \text{CorruptMe?}) \wedge \\
& \quad m' = (id, (\text{SetCorruptionStatus}, b)) \wedge \\
& \quad id \in \{0, 1\}^* \wedge b \in \{\text{false}, \text{true}\}\} \\
\cup & \{(m, m') \mid m = (id, (\text{CorrStatusRestrict}, b, m'')) \wedge \\
& \quad m' = (id, \text{OK}) \wedge \\
& \quad id \in \{0, 1\}^* \wedge b \in \{\text{false}, \text{true}\} \wedge m'' \in \{0, 1\}^* \cup \{\perp\}\} \\
\cup & \{(m, m') \mid m = (id, (\text{Respond}, m'')) \wedge \\
& \quad m' = (id, m''') \wedge \\
& \quad id \in \{0, 1\}^* \wedge m'', m''' \in \{0, 1\}^* \cup \{\perp\}\}
\end{aligned}$$

Note that according to R defined above, a restricting message $m \in R[0]$ and a possible response m' with $(m, m') \in R$ always start with the same ID id . In our framework, instances of machines will use id to store the sending entity of a message. Thus, by the definition of

the restriction the response will be sent back to the same entity and thus to the corresponding instance. In particular, the adversary/environment may not interact with any other protocol instances before sending this response (except for negligible probability, which can be ignored in security proofs).

We note that, as also discussed in Section 3.2.3, it would be sufficient to consider a restriction R which contains only the last type of message pairs. That is, one simply indicates a restricting message by adding **Respond** to the message and does not make any restrictions about the response. If one wants an answer that satisfies certain conditions, one can inspect the answer and if it does not satisfy the condition, one can send the restricting message again until one obtains a message that satisfies the condition; see also the command introduced next. However, we chose to consider the above version of R as it makes explicit which responses are permitted for framework specific messages and thus slightly simplifies runs (i.e., we do not have to re-send messages in those cases).

We write **send responsively** mp_{out} **on** τ_{NET} ; **wait for** mp_{in} **on** τ_{NET} **s.t.** $\langle condition \rangle$ to emphasize that the machine sends a *restricting* message on a network tape τ_{NET} and then waits to receive a response on the same network tape that matches with mp_{in} and satisfies $\langle condition \rangle$. This command will only be used if a message $m \in R[0]$ is sent. We note that the message pattern mp_{in} is usually defined in such a way that it accepts (some of the) possible answers to the restricting message. If an incoming message m' (with $(m, m') \in R$) is not accepted by the command, then the machine repeats the command **send responsively** mp_{out} **on** τ_{NET} ; **wait for** mp_{in} **on** τ_{NET} **s.t.** $\langle condition \rangle$, i.e., it automatically sends the first message m on τ_{NET} again and waits for an answer that matches mp_{in} and satisfies *condition*. This is repeated until the answer matches mp_{in} and satisfies *condition*. Because of the responsiveness requirement, it is guaranteed that the environment/adversary/simulator has to provide the expected answer to the correct instance if it wants the run to continue.

Abort: We use the special keyword **abort** to say that a machine stops its current activation at some point. More specifically, as soon as a machine in **Compute** mode reaches the **abort** command, it will produce empty output and thus stop its computation. Then, by definition, the master IITM is activated with empty input on the **start** tape.

D.2. Mapping Templates/Machines

To explain how protocols in our framework can be interpreted as (protocol) systems in the sense of the IITM model, we first have to explain how a single instance of the template in Figure 4.3 is mapped to a system of IITMs. We start by describing how such a system and machines therein are structured and then detail the **CheckAddress** and **Compute** modes of the IITMs, including the behavior in case of corruption. Based on the mapping of templates, we then explain in Appendix app:iuc:mapping:protocols how a full protocol, potentially defined via several different instances of the template interacting with each other, as well as public and

private roles are interpreted as a system in the IITM model. While some of the following has already been sketched Section 4.2, this section provides full details.

D.2.1. System of machines

In the following, let \mathcal{P} be a protocol defined by a single instance of the template in Figure 4.3. Recall from Section 4.2 that protocols consist of several machines (i.e., IITMs) that implement the roles in this protocol. To be more precise, for each (set of) role(s) defined in the **Participating roles** field, there is one machine that implements this role (set). Thus, if there are n (sets of) roles in \mathcal{P} , then \mathcal{P} is the system $\{M_1, \dots, M_n\}$. Next, we explain how a machine $M \in \{M_1, \dots, M_n\}$ given in the template is defined, where M_i is defined by one part of the template. Let $\{role_1, \dots, role_m\}$ be the set of roles that M implements, as specified in the **Implemented role(s)** field.

D.2.2. Tapes

Recall from Sections 3.2.2 and 3.2.3 that a machine M has named pairs of input and output tapes for (bidirectional) communication with other machines. In the following, for simplicity of presentation, we will call such a pair of named input and output tapes simply a (named) tape. An instance of a machine M_1 can write a message on a tape t named n ; if there is another machine M_2 with a tape t' also named n , then the message is delivered to (an instance of) M_2 on tape t' . In this case, we say that M_1 and M_2 (respectively their tapes t and t') are connected. Within a system of IITMs, it is required that tapes connect uniquely, i.e., for each name there are at most two machines with tapes of that name. Also recall that tapes are grouped into network and I/O tapes.

On a high level, we have to use tapes to represent the abstract connections from the iUC framework. That is, we have to represent network connections each role has to the adversary, (internal) I/O connections between various roles, including subroutine roles, in a protocol, and a set of connections from each (public) role to the environment that can be used to simulate higher-level protocols. To give an intuition right away, let us look at an example.

Example D.1. In Figure 4.8 on Page 138 each of the arrows between the box labeled “I/O” (representing an environment) and the protocol will be represented via a parameterized set of I/O tapes. Each of the internal arrows between roles of the protocol will be represented by a single unique (bidirectional) I/O tape. In addition (not shown in the Figure) each role will have one (bidirectional) network tape for connecting to the adversary.

In the following, we formalize the tapes that a single machine M offers for others to connect to. Later, in Appendix D.3, we detail how multiple machines are connected in the context of a protocol, where some (roles of a) machine are public and others are private.

Let M be the machine of \mathcal{P} from above. For each role $role_i$ implemented by M , there is one network tape that is used to connect the adversary, allowing $role_i$ to send and receive messages

from the network. Furthermore, for each role $role_i$ and for each subroutine role $subrole_j$ of M , there is a I/O tape that is used to connect $role_i$ to $subrole_j$, allowing them to directly send and receive messages to each other. Finally, each role $role_i$ also has a parameterized number of I/O tapes that allow other (unknown) higher-level protocols as well as the environment to connect and send direct messages to $role_i$. All tapes of M are uniquely named (the exact names can be chosen arbitrarily and do not matter for the purpose of this mapping. Later on, in Appendix D.3, the names of I/O tapes will be chosen such that tapes connect to subroutine roles and higher-level protocols/the environment as expected).

Jumping ahead, the exact parameterized number of I/O tapes for a role will be fixed depending on whether the role is public in the context of a protocol \mathcal{Q} (cf. Appendix D.3): If $role_i$ is private, then it offers exactly as many I/O tapes as are used by other roles of \mathcal{Q} (that are not specified as subroutines by $role_i$) that want to connect to $role_i$ as a subroutine. For public roles, the number of I/O tapes is arbitrarily large but fixed, allowing the environment to connect to an arbitrary number of I/O tapes and use them for simulating higher-level protocols.

Example D.2 (Example D.1 continued). In Figure 4.8 the **signer** role is private (in the context of the combined protocol) and hence offers exactly two I/O tapes for the (unknown higher-level) **initiator** and **responder** roles to connect to (represented by the two arrows between those roles). In contrast, the **initiator** role is public and hence offers an arbitrary number of I/O tapes for the environment to connect to (these are represented by the single arrow pointing towards the box labeled “I/O”).

D.2.3. Message format

In order to uniquely determine both the sending entity and the intended receiving entity of a message, machines in our framework expect incoming messages m on some tape t to have a specific format (and likewise will encode all outgoing messages in this format). More specifically, a message received on an I/O tape must be of the form $m = ((pid_{snd}, sid_{snd}), (pid_{rcv}, sid_{rcv}), m')$ where m' is the message payload, pid_{snd} is the PID of the sender, sid_{snd} is the SID of the sender, pid_{rcv} is the PID of the intended receiver, and sid_{rcv} is the SID of the intended receiver. The expected format for messages m received and sent on network tapes is shortened to be $m = ((pid_{rcv}, sid_{rcv}), m')$ and $m = ((pid_{snd}, sid_{snd}), m')$ respectively, as one of the communication partners is always the network, i.e., there is no second entity involved in the communication. Messages that do not adhere to this format are automatically dropped by the CheckAddress mode (cf. Appendix D.2.4).

Now, suppose M receives a message $m = ((pid_{snd}, sid_{snd}), (pid_{rcv}, sid_{rcv}), m')$ on one of its I/O tapes t belonging to role $role_i$. The sending and receiving entities are computed from the message m and the tape t as follows. The receiving entity is set to $(pid_{rcv}, sid_{rcv}, role_i)$. For the sending entity, there are two cases to distinguish: if t is a tape connecting $role_i$ to one of its known subroutine roles $subrole_j$, then the sending entity is set to be $(pid_{snd}, sid_{snd}, subrole_j)$. Otherwise, t is one of the parameterized many I/O tapes of $role_i$. In that case, the sending entity

is set to be $(pid_{snd}, sid_{snd}, l)$ where $l \in \mathbb{N}$ is a number that uniquely identifies the tape t (i.e., all of the parameterized many tapes are numbered consecutively). Note that, while technically speaking l is not the name of a role, it still identifies a unique name of a role that is connected to tape t (cf. paragraph “exchanging messages” in Section 4.2.1).⁶⁰

Conversely, suppose an entity $(pid_{snd}, sid_{snd}, role_i)$ managed by an instance of M wants to send a message body m' to some other entity $(pid_{rcv}, sid_{rcv}, subrole_j)$ or $(pid_{rcv}, sid_{rcv}, l)$, $l \in \mathbb{N}$. The resulting message m , including the header, and the tape t that is used for sending are computed as follows. The message m is set to be $((pid_{snd}, sid_{snd}), (pid_{rcv}, sid_{rcv}), m')$. The tape t is either the I/O tape connecting $role_i$ to its subroutine role $subrole_j$, or the parameterized I/O tape connecting to a higher-level protocol which is identified by the number l .

Sending and receiving entities are computed analogously for messages that are sent/received on a network tape. We note that protocol designers using our syntax from Section 4.2.6 only have to deal with the actual message payload m' in their protocol specification; the syntax automatically takes care of adding the correct headers to the message payload and parsing headers of incoming messages.

D.2.4. Check address mode of protocol machines

As mentioned, every instance of a machine in our framework manages one or more entities of the form $(pid, sid, role)$ (where $role$ is one of the roles of M), and every entity is managed by a unique instance, i.e., there are no two instances that manage the same entity. On a high level, the `CheckAddress` mode is used to decide which instance manages which entity and route incoming messages accordingly.

First, recall that in the IITM model, whenever a message m is received on a tape of M , then all existing instances of M (in the order of their creation) are invoked in mode `CheckAddress` to check which instance accepts m . The first instance to accept m gets to process m in mode `Compute`. If no such instance exists, then a new one is created and run in mode `CheckAddress`. If this new instance accepts, it gets to process m , and otherwise, m is dropped and the new instance is removed from the run.

Now, upon being activated with a message m on tape t , an instance of M does the following in the `CheckAddress` mode, as specified in detail in Figure D.1: the instance first checks that m contains a header that specifies the sender and intended receiver as described in the paragraph “message format”. Note that the expected header format depends on whether the tape t is an I/O or network tape. If m does not contain a correct header, then `reject` is output; this ensures that instances accept a message and enter `Compute` only if they can determine both the sender and/or intended receiver of a message. Otherwise, the instance determines

⁶⁰In other words, M is not aware of the precise name of the role as the connecting higher-level protocol is unknown/only simulated by the environment. In particular, M cannot depend on this name in their code. Note that having the tuple $(pid_{snd}, sid_{snd}, l)$ is typically sufficient as it allows for uniquely addressing the sending entity, e.g., to return a message. A machine can simply send a message with sender (pid_{snd}, sid_{snd}) on the tape with number l .

Upon receiving a message m on tape t in mode **CheckAddress**, do the following.

if t is an I/O tape: {Check that m contains the expected header, cf. Appendix D.2.3}
 Check that $m = ((pid_{snd}, sid_{snd}), (pid_{rcv}, sid_{rcv}), m')$.

else:
 Check that $m = ((pid_{rcv}, sid_{rcv}), m')$.

if the above check fails:
 output **reject**.

Compute the receiving role $role_{rcv}$ from t . {cf. Appendix D.2.3}
 $decision \leftarrow \mathbf{CheckID}(pid_{rcv}, sid_{rcv}, role_{rcv})$.
 Output $decision$.

Figure D.1.: The **CheckAddress** mode of protocol machines in our framework.

the intended receiving entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$ as described in Appendix D.2.3 and runs $\mathbf{CheckID}(pid_{rcv}, sid_{rcv}, role_{rcv})$ to determine whether that entity is accepted or rejected. Thus, the protocol designer can freely define which receiving entities are accepted and thus managed by an instance of a machine.

Note that, since we require **CheckID** to produce consistent outputs (i.e., never output both **accept** and **reject** for the same entity during any run), the above definition of the **CheckAddress** mode implies that every receiving entity that has been accepted at some point will be accepted by the same instance again; in particular, no other instance gets to process messages for that entity during any point in the run. In other words, in every run for every (accepted) entity there is a uniquely determined instance that implements/manages that entity during mode **Compute**.

D.2.5. Compute mode of protocol machines

Recall that the **Compute** mode of an IITM specifies the actual computation performed by (an instance of) the IITM. Our framework fixes parts of the behavior of protocol machines in a specific way to guarantee the desired behavior in terms of corruption and addressing other machines. All other aspects can be customized by a protocol designer via specification of the various algorithms in the template from Figure 4.3.

We provide the formal specification of the **Compute** mode of protocol machines in Figures D.2, D.3, and D.4. On a high level, when an instance is activated with some message that includes the message body m from a sender $sender$ (either some entity connected via the I/O interface or the network) for some receiving entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$, then the instance performs the following steps in order:

1. If $m = \mathbf{CorruptionStatus?}$ on an I/O tape, then the corruption status of the entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$ is determined and returned to the sender immediately. In particular, none of the following steps are performed but instead a response is returned directly to the sending entity $sender$.
2. If this is the first time that this instance reaches this step, then it runs **Initialization**.

3. If this is the first time that this instance reaches this step when receiving a message for the entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$, then it runs **EntityInitialization**.
4. If this is the first time that this instance reaches this step when receiving a message for the entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$, then it asks the adversary to determine the initial corruption status of that entity. This is done via a restricting message, i.e., the adversary is forced to respond such that the computation can continue from this point forward (except for negligible probability).
5. Corruption requests received from the network are processed. We note that already in Step 4 the message to be processed might be a corruption request. In this case, in 4. the adversary is not asked whether or not he wants to corrupt the entity.
6. If the instance was activated by the initial message $m = \text{InitEntity}$ on an I/O tape, then the instance reports a successful initialization by sending a message to the sender *sender*. This ends the activation and none of the following steps are performed. We note that, by default, a sender of a **InitEntity** request receives a response even if the adversary decides to corrupt $(pid_{rcv}, sid_{rcv}, role_{rcv})$ in Step 4.
7. If $(pid_{rcv}, sid_{rcv}, role_{rcv})$ is explicitly corrupted by the adversary, then messages are forwarded to/from the network.
8. Otherwise, if the receiver entity $(pid_{rcv}, sid_{rcv}, role_{rcv})$ is honest, then first the algorithm **MessagePreprocessing** is run which might edit the message body m' . If the algorithm **MessagePreprocessing** does not end the current activation, e.g., by sending a message, then **Main** is run afterwards on the modified message body m' .

Let us highlight and discuss a few aspects of the implementation in the following.

User-defined algorithms: The Compute mode makes calls to all user-defined algorithms from the template given in Figure 4.3, except for **CheckID** which is used in the CheckAddress mode. We do not fix or restrict how these algorithms should be defined and we allow them full access not only to the internal state but also to all framework-specific variables such as **transcript**, which is a log of all sent and received messages, and $(pid_{cur}, sid_{cur}, role_{cur})$, which stores the entity that has received the current message.⁶¹ While this gives great flexibility for protocol designers, it also means that they must be careful in their definitions such that they do not accidentally disrupt the intended protocol execution. In general, all algorithms should ensure that, when sending a message, that message has the expected format (cf. Appendix D.2.3) including a header that specifies sender and receiver. Note that this is automatically taken care of if our convenient syntax from Section 4.2.6 is used, i.e., a protocol designer using this syntax only has to worry

⁶¹This access is kept implicit in Figures D.2 to D.4 as we do not want to clutter the calls to the algorithms with several additional parameters.

State variable $\text{acceptedEntities} \subseteq (\{0, 1\}^*)^3 = \emptyset.$	$\left\{ \begin{array}{l} \text{List of entities that have been accepted so far. Mainly for use in the CheckID algorithm.} \end{array} \right.$
State variable $\text{initDone} \in \{\text{true}, \text{false}\} = \text{false}.$	$\left\{ \begin{array}{l} \text{Has the instance been initialized?} \end{array} \right.$
State variable $\text{entityInitDone} \subseteq (\{0, 1\}^*)^3 = \emptyset.$	$\left\{ \begin{array}{l} \text{Set of entities that have been initialized.} \end{array} \right.$
State variable $\text{explicitCorr} : (\{0, 1\}^*)^3 \rightarrow \{\text{true}, \text{false}, \perp\}.$	$\left\{ \begin{array}{l} \text{Has an entity been explicitly corrupted? Initially } \perp. \end{array} \right.$
State variable $\text{corrStatus} : (\{0, 1\}^*)^3 \rightarrow \{\text{true}, \text{false}\}.$	$\left\{ \begin{array}{l} \text{Consider entity to be corrupted? Initially false.} \end{array} \right.$
State variable $\text{internalState}.$	$\left\{ \begin{array}{l} \text{The internal state of the machine as defined in the template from Figure 4.3.} \end{array} \right.$
State variable $\text{transcript}.$	$\left\{ \begin{array}{l} \text{Log of all messages that were sent and received.} \end{array} \right.$
State variable $\text{entity}_{\text{cur}} \in (\{0, 1\}^* \cup \{\perp\})^3 = (\perp, \perp, \perp).$	$\left\{ \begin{array}{l} \text{Currently active entity (which was activated by receiving a message).} \end{array} \right.$
State variable $\text{entity}_{\text{call}} \in (\{0, 1\}^* \cup \{\perp\})^3 = (\perp, \perp, \perp).$	$\left\{ \begin{array}{l} \text{Sender of the last message that was received on the I/O interface.} \end{array} \right.$
<p>Upon receiving a message $m = ((\text{pid}_{\text{snd}}, \text{sid}_{\text{snd}}), (\text{pid}_{\text{rcv}}, \text{sid}_{\text{rcv}}), m')$ on an I/O tape t, or a message $m = ((\text{pid}_{\text{rcv}}, \text{sid}_{\text{rcv}}), m')$ from a network tape t do:</p> <p>Determine the receiver entity, append it to the list acceptedEntities if it is not yet included, and store it in $\text{entity}_{\text{cur}} = (\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}})$. If t is an I/O tape, also determine the sender entity and store it in $\text{entity}_{\text{call}} = (\text{pid}_{\text{call}}, \text{sid}_{\text{call}}, \text{role}_{\text{call}})$. Let t_{NET} be the network tape corresponding to role_{cur}. $\left\{ \begin{array}{l} \text{cf. Appendix D.2.2 and Appendix D.2.3.} \end{array} \right.$</p>	
<p>if $\text{Corruption model} \neq \text{custom} \wedge m' = \text{CorruptionStatus?} \wedge t$ is an I/O tape: $\left\{ \begin{array}{l} \text{Step 1: handle corruption status requests.} \end{array} \right.$</p>	
<p> if $\text{explicitCorr}[\text{entity}_{\text{cur}}] = \perp$: $\left\{ \begin{array}{l} \text{Initial corruption status has not been determined yet.} \end{array} \right.$</p>	
<p> send $((\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}), (\text{pid}_{\text{call}}, \text{sid}_{\text{call}}), (\text{CorruptionStatus}, \text{false}))$ on t.</p>	
<p> else:</p>	
<p> $\text{corrStatus}[\text{entity}_{\text{cur}}] \leftarrow \text{corrStatus}[\text{entity}_{\text{cur}}] \vee \text{explicitCorr}[\text{entity}_{\text{cur}}]$.</p>	
<p> if $\text{corrStatus}[\text{entity}_{\text{cur}}] = \text{false}$:</p>	
<p> $\text{corrStatus}[\text{entity}_{\text{cur}}] \leftarrow \text{DetermineCorrStatus}(\text{entity}_{\text{cur}})$.</p>	
<p> send $((\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}), (\text{pid}_{\text{call}}, \text{sid}_{\text{call}}), (\text{CorruptionStatus}, \text{corrStatus}[\text{entity}_{\text{cur}}]))$ on t.</p>	
<p>if $\text{initDone} = \text{false}$: $\left\{ \begin{array}{l} \text{Steps 2 and 3: Initialization of internal state.} \end{array} \right.$</p>	
<p> $\text{initDone} \leftarrow \text{true}$.</p>	
<p> Run Initialization.</p>	
<p>if $\text{entity}_{\text{cur}} \notin \text{entityInitDone}$:</p>	
<p> Add $\text{entity}_{\text{cur}}$ to entityInitDone.</p>	
<p> Run EntityInitialization($\text{entity}_{\text{cur}}$).</p>	
<p>Continue in Figure D.3.</p>	

Figure D.2.: The Compute mode of protocol machines (part I).

```

if Corruption model  $\neq$  custom:
  if explicitCorr[entitycur] =  $\perp$ :
    {Step 4: Initialize corruption status of new entity.
    if  $t = \tau_{\text{NET}} \wedge m' = (\text{SetCorruptionStatus}, b) \wedge b \in \{\text{true}, \text{false}\}$ :
      {First message al-
      ready sets corrup-
      tion status.
      if  $b = \text{true}$  and Corruption model allows corruption of the entity at this point:
        {See Section 4.2.2 for when corruption of an entity is allowed.
        explicitCorr[entitycur]  $\leftarrow$  AllowCorruption(entitycur).
        {Use AllowCorruption to decide whether corruption succeeds.
      else:
        explicitCorr[entitycur]  $\leftarrow$  false.
      if explicitCorr[entitycur] = true:
        {Return a response to the adversary. Note that
        this stops the activation, none of the following
        steps are performed.
        leakage  $\leftarrow$  LeakedData().
        send ((pidcur, sidcur), (CorruptionStatus, true, leakage)) on  $\tau_{\text{NET}}$ .
      else:
        send ((pidcur, sidcur), (CorruptionStatus, false,  $\perp$ )) on  $\tau_{\text{NET}}$ .
    else:
      {For all other first messages.
      send responsively ((pidcur, sidcur), CorruptMe?) on  $\tau_{\text{NET}}$ ;
      wait for ((pidcur, sidcur), (SetCorruptionStatus, b)) on  $\tau_{\text{NET}}$  s.t.  $b \in \{\text{true}, \text{false}\}$ .
      if  $b = \text{true}$  and Corruption model allows corruption of the entity at this point:
        {See Section 4.2.2 for when corruption of an entity is allowed.
        explicitCorr[entitycur]  $\leftarrow$  AllowCorruption(entitycur).
        {Use AllowCorruption to decide whether corruption succeeds.
      else:
        explicitCorr[entitycur]  $\leftarrow$  false.
      if explicitCorr[entitycur] = true:
        {Leak information and give control to either the ad-
        versary or, if this instance was triggered by an
        InitEntity command, to entitycall.
        leakage  $\leftarrow$  LeakedData().
        if  $t$  is an I/O tape and  $m' = \text{InitEntity}$ :
          send responsively ((pidcur, sidcur), (CorrStatusRestrict, true, leakage)) on  $\tau_{\text{NET}}$ ;
          wait for ((pidcur, sidcur), OK) on  $\tau_{\text{NET}}$ .
          send ((pidcur, sidcur), (pidcall, sidcall), InitEntityDone) on  $t$ .
        else:
          send ((pidcur, sidcur), (CorruptionStatus, true, leakage)) on  $\tau_{\text{NET}}$ .
    else if explicitCorr[entitycur] = false:
      {Step 5: Process corruption requests for al-
      ready existing entities.
      if  $t = \tau_{\text{NET}} \wedge m' = (\text{SetCorruptionStatus}, b) \wedge b \in \{\text{true}, \text{false}\}$ :
        if  $b = \text{true}$  and Corruption model allows corruption of entities at this point:
          explicitCorr[entitycur]  $\leftarrow$  AllowCorruption(entitycur).
        else:
          explicitCorr[entitycur]  $\leftarrow$  false.
        if explicitCorr[entitycur] = true:
          {Return a response to the adversary. Note that
          this stops the activation, none of the following
          steps are performed.
          leakage  $\leftarrow$  LeakedData().
          send ((pidcur, sidcur), (CorruptionStatus, true, leakage)) on  $\tau_{\text{NET}}$ .
        else:
          send ((pidcur, sidcur), (CorruptionStatus, false,  $\perp$ )) on  $\tau_{\text{NET}}$ .

```

Continue in Figure D.4.

Figure D.3.: The Compute mode of protocol machines (part II).

<pre> if t is an I/O tape and $m' = \text{InitEntity}$: send $((\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}), (\text{pid}_{\text{call}}, \text{sid}_{\text{call}}), \text{InitEntityDone})$ on t. if $\text{Corruption model} \neq \text{custom} \wedge \text{explicitCorr}[\text{entity}_{\text{cur}}] = \text{true}$: if t is an I/O tape: send $((\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}), (\text{CorrMsgForward}, \text{entity}_{\text{call}}, m'))$ on τ_{NET}. else if $m' = (\text{CorrMsgForward}, (\text{pid}_{\text{target}}, \text{sid}_{\text{target}}, \text{role}_{\text{target}}), m'')$: Let t' be the tape connecting to $\text{role}_{\text{target}}$ (if there is no such tape, abort). if $\text{AllowAdvMessage}(\text{entity}_{\text{cur}}, \text{entity}_{\text{target}}, m'')$: send $((\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}), (\text{pid}_{\text{target}}, \text{sid}_{\text{target}}, m''))$ on t'. else: send $((\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}), (\text{CorrMsgForward}, \text{failed}))$ on τ_{NET}. else: Append $(\text{recv}, (m, t))$ to transcript. Run $\text{MessagePreprocessing}(\text{entity}_{\text{call}}, \text{entity}_{\text{cur}}, m')$. Run $\text{Main}(\text{entity}_{\text{call}}, \text{entity}_{\text{cur}}, m')$. abort. </pre>	<p><i>{ Step 6: Respond to InitEntity requests as initialization of $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}})$ is finished now. }</i></p> <p><i>{ Step 7: corrupted instances act as multiplexers for the adversary. }</i></p> <p><i>{ Handle message forwarding to $\text{entity}_{\text{target}}$. }</i></p> <p><i>{ Step 8: Honest behavior. }</i></p> <p><i>{ Update message log with non-framework specific message. This is also done for all incoming and outgoing messages in MessagePreprocessing and Main. }</i></p> <p><i>{ Note that this algorithm might modify m'. }</i></p> <p><i>{ In the case that no message was sent. }</i></p>
--	---

Figure D.4.: The Compute mode of protocol machines (part III).

about the actual message payloads. The following messages payloads should be used with care as they are also used internally by our framework:

- InitEntity
- InitEntityDone
- CorruptionStatus?
- CorruptionStatus
- CorruptMe?
- SetCorruptionStatus
- CorrStatusRestrict
- CorrMsgForward

In principle, every algorithm can end the current activation either by sending a message or using the **abort** keyword. This has to be used with some care as it is quite easy to disrupt the intended protocol execution. In particular, a protocol generally should ensure that the initialization phase can run without interruption.

Core logic of (honest) entities: Recall that the core logic of a protocol machine is defined via the four algorithms **Initialization**, **EntityInitialization**, **MessagePreprocessing**, and **Main**. One important property of both initialization algorithms is that they are run before the initial corruption status of the current entity is determined and thus before the adversary

can take control of the current entity. This means that initialization is performed even if the adversary intends to corrupt the current entity right at the start, allowing for performing a protected setup without interference of the adversary. In particular, one can first honestly create some internal state, and then later on decided based on this internal state whether the adversary may actually corrupt the current entity. In contrast, both **MessagePreprocessing** and **Main** are executed only for honest entities, so if an adversary corrupts an entity right at the beginning, they will never be run for that entity. Instead, all messages would always be forwarded to the adversary in this case.

Corruption handling: While the general corruption related behavior and the corresponding algorithms have already been explained in Section 4.2.2 and Section 4.2.3, we now give more details about the technical aspects of corruption in this paragraph.

First, note that all corruption related behavior (Steps 1, 4, 5, and 7) will be disabled and skipped entirely if **Corruption model** is set to **custom**. Thus, all framework-specific corruption related messages, such as **CorruptionStatus?**, are no longer handled automatically in such a case. Instead, a protocol designer is able to receive those messages in **MessagePreprocessing** and **Main** and manually specify how they are handled. For example, one can define an entirely different mechanism where corruption is not handled per entity but rather per machine instance, while **CorruptionStatus?** requests are still answered in an appropriate way to ensure interoperability with other protocols from our framework.

If **Corruption model** is not set to **custom**, then Step 1 takes care of all **CorruptionStatus?** requests from other protocols/the environment. Observe that this is done at the very start of **Compute** mode and the current activation is ended directly after, without even performing any type of initialization. This ensures that **CorruptionStatus?** requests, by default and depending on the definition of the **DetermineCorrStatus** algorithm, are not visible to the network and do not affect the behavior of the protocol. As mentioned previously, this is because intuitively these requests are meta messages that are supposed to allow other protocols/the environment to obtain a snapshot of the current corruption state at any point in time, i.e., it should not matter for the protocol execution whether/when such a snapshot was retrieved. In particular, if the behavior of the protocol were to depend on whether a **CorruptionStatus?** request was received previously, then this can create additional artificial attack vectors for the environment.

Except for **CorruptionStatus?** requests, all corruption related behavior is performed during Steps 4, 5, and 7, i.e., after (honest) initialization has been completed but before the algorithms **MessagePreprocessing** and **Main**. The implementation mostly follows the behavior that has been described before and which we do not repeat here. We want to highlight some details though: firstly, observe that, when an entity asks for its initial corruption status in Step 4, it does so using a restricting message which the adversary must answer immediately. This feature ensures that the adversary must decide on the corruption status without interrupting the protocol execution by, e.g., first interacting with and changing the state of other instances. In particular, if the adversary decides not to corrupt the entity, then the protocol execution

continues without interference. Secondly, there is a technical special case that needs to be handled differently, namely, receiving an `InitEntity` request from the I/O interface (cf. Step 6). This special message can be used by other protocols to initialize a subroutine entity, including its corruption status, but without losing control to the adversary in case the entity gets corrupted (cf. `init` macro defined in Section 4.2.6). Thus, if the adversary corrupts an entity that initializes itself due to `InitEntity` request, he is notified via a restricting message (instead of a regular message) which includes the leakage of the entity. The adversary then has to return control to the corrupted entity immediately, which in turn can send the expected response to the initial sender of the `InitEntity` request. Furthermore, if such a request is received, then the instance responds after Step 5 as then the entity has been initialized, i.e., Step 7 in particular is not executed. Overall, this yields the desired behavior.

D.2.6. Mapping the syntax from Section 4.2.6

Interpreting the send and receive commands presented in Section 4.2.6 as commands in the sense of the IITM model is mostly straightforward. In particular, the tapes that are used and the headers that are added to the message payloads can be directly computed from the sending and receiving entities that are specified in the send and receive commands (as described in Appendix D.2.3). However, the `send responsively; wait for` and `send; wait for` commands need some additional explanation.

Recall that the `send responsively; wait for` command allows for sending messages to the network that will be answered immediately with a specific message. In particular, the adversary is not allowed to interact with other parts of the protocol or other entities before providing the expected response. Formally, this command sends the restricting message $(header, (\text{Respond}, m'))$ on a network tape, where *header* is the message header as defined in Appendix D.2.3 and m' is the message payload as specified in the `send responsively` command. By definition of the restriction (cf. Appendix D.1), the adversary has to respond to such a message with a message of the format $(header, m'')$, where *header* is the same as in the restricting message and m'' is some arbitrary bit string. Because the headers are identical, the same instance of a machine will receive the response. That instance then continues where it left off, i.e., without repeating all previous steps and with all local variables still set to the same values, and tries to match m'' to the conditions and the message format imposed by the `wait for` part. If the match fails, it repeats the whole process by re-sending the original message $(header, (\text{Respond}, m'))$. Thus, an adversary must provide an expected answer if he wants the run to continue. We note that, formally, the adversary might not provide an immediate response (or a wrong response, e.g., with an incorrect header) with negligible probability, however, this negligible set of runs can be ignored in security proofs.

While the purpose of the `send; wait for` command is similar to the `send responsively; wait for` command, its implementation is actually more complex. Recall that this command is also supposed to allow a protocol designer to send some message and then wait for a response such

that the run continues where it left off, including keeping all local variables. However, unlike the **send responsively; wait for** command, the **send; wait for** command can be used for sending messages to both arbitrary protocols and the network; furthermore, it is not guaranteed that such a request will indeed be answered immediately. For example, if an instance I queries a subroutine via the **send; wait for** command, then this subroutine might be corrupted such that the request is forwarded to the adversary who can decide to, e.g., not answer but rather activate the instance I on a network tape or a different I/O tape again. Thus, we need to specify the behavior of I while it is waiting for a response but receives another message. In particular, it should still allow for certain meta actions, such as obtaining the current corruption status or sending a message in the name of a corrupted entity, to be processed.

We use the following definition: If an instance I sends a message via the **send; wait for** command, then it pushes the current values of all local variables and the current position in the program code on an internal stack. Now, if I receives some message m while this stack is non-empty, it proceeds as follows depending on the type of message:

- Check whether m is one of the following four types of meta messages: `CorruptionStatus?`, `InitEntity` on the I/O interface, or `SetCorruptionStatus`, `CorrMsgForward` on the network interface.

If so, then proceed with the standard program logic while ignoring the state stored on the internal stack. In other words, these messages are processed separately even when waiting for a response to another request. Note that these messages might also include calls to a **send; wait for** command and thus push new states to the stack.

- For all other messages m , I takes the top most state stored on the stack and continue from the stored program position with the stored local variables. That is, the instance checks whether m matches all criteria of the **wait for** command based on the stored values of local variables and the current values of global variables (which might have changed since the state was stored on the stack due to meta messages that have been processed in the mean time!). If it does, then the computation continues where it left of; otherwise, the state is pushed back to the stack and the instance stops the current activation without output.

As should be obvious from this definition, it is quite easy to produce unintended behavior with the **send; wait for** command. We thus emphasize again that this construct must be used with special care and only sparsely, e.g., to obtain a value from an incorruptible subroutine that responds immediately by its definition.

Besides the send and receive commands, we have also introduced two macros in Section 4.2.6: the `corr(pid, sid, role)` and the `init(pid, sid, role)` macros for retrieving the corruption status of and initializing an entity, respectively. Using the notation from Section 4.2.6, `corr(pid, sid, role)` is just a shorthand notation for the command **send CorruptionStatus? to (pid, sid, role); wait for (CorruptionStatus, b) s.t. $b \in \{\text{true}, \text{false}\}$** . Analogously, `init(pid, sid, role)` is just a shorthand notation for **send InitEntity to (pid, sid, role); wait for InitEntityDone**. Both of these are fully defined by the above explanations.

D.3. Mapping Protocols

Now that we have explained how our template can be mapped to individual machines in the sense of the IITM model, we can explain how a complete protocol $\mathcal{P} = (role_1^{pub}, \dots, role_n^{pub} \mid role_1^{priv}, \dots, role_m^{priv})$ is mapped to a system of machines, where \mathcal{P} might be built from several (sub-)protocols which are each specified using the template. More specifically, we mainly need to explain how the tapes of individual machines are connected and how public and private roles differ.

Let \mathcal{P} be an arbitrary complete protocol as above. Let M_1, \dots, M_l be the machines of \mathcal{P} that are specified using one or more templates and which implement one or more roles each. Now, the tapes corresponding to a role $role_i$ implemented by a machine M_j are connected as follows (in the context of \mathcal{P}) via suitable tape name choices (again, the exact names are arbitrary and not important for this mapping):

- Recall that $role_i$ has an I/O tape t for each of the subroutine roles $role_{sub}$ of M_j . Since \mathcal{P} is complete, we have that $role_{sub}$ is implemented by one of the machines M_k of \mathcal{P} . Note that $k \neq j$ as M_j does not specify its own roles as subroutines, i.e., M_k and M_j are different machines and can hence in principle be connected via a tape. If (the machine implementing) $role_{sub}$ also has an I/O tape t' to connect to $role_i$ (because $role_{sub}$ specifies $role_i$ as a subroutine), then t and t' are connected and we require the (bidirectional) tapes t and t' to be identical since we need only one (bidirectional) tape for $role_i$ and $role_{sub}$ to communicate. Otherwise, the tape t is connected to one of the parameterized many I/O tapes of $role_{sub}$.
- Recall that $role_i$ also has parameterized many I/O tapes for arbitrary higher-level protocols and the environment to connect to.
 - If $role_i$ is a private role in \mathcal{P} , then the parameter is chosen such that there are exactly as many tapes as are needed for roles of \mathcal{P} that want to use $role_i$ as a subroutine (and that are not also specified as subroutine by $role_i$ itself. Those roles are already connected as described above). In other words, all roles of \mathcal{P} that use $role_i$ as a subroutine can actually connect to $role_i$; however, there are no additional (unconnected) tapes that would allow other protocols or the environment to connect to $role_i$.
 - If $role_i$ is a public role in \mathcal{P} , then the parameter is still arbitrary but sufficiently large such that all roles of \mathcal{P} that use $role_i$ as a subroutine can connect to it. In other words, not only can roles of \mathcal{P} connect to $role_i$ (if they use this role as subroutine), but there are also arbitrarily many unconnected tapes that can be used by other protocols or the environment to connect to $role_i$.
- The single network tape of $role_i$ is left unconnected such that the environment/adversary/simulator in the definition of the realization relation can connect to it.

The protocol \mathcal{P} is then implemented by the system of machines $\{M_1, \dots, M_l\}$ that is connected as described above. This was already illustrated by Example D.1.

E. Security Definitions for Cryptographic Primitives

In this chapter, we present the security notions which we use to realize $\mathcal{F}_{\text{crypto}}$. All these notions are standard. We note that traditionally game based security notions consider uniform adversaries, i.e., adversaries that do not obtain an additional external input. In contrast, universal composability models usually consider non-uniform environments that *do* obtain some external input. In order to reduce security both settings have to be compatible. Hence, we use adapted versions of the security notions where non-uniform adversaries are considered. Of course, all results of this paper also carry over to the uniform setting.

E.1. Symmetric Encryption

Here we recall the definition of symmetric encryption schemes and the IND-CPA, IND-CCA2, and INT-CTXT security notions.

Definition E.1. *A symmetric encryption scheme $\Sigma = (\text{gen}, \text{enc}, \text{dec})$ with plaintext domain $\text{dom}(\Sigma) \subseteq \{0, 1\}^*$ consists of three polynomial-time algorithms. The probabilistic key generation algorithm gen expects a security parameter η and returns a key $\text{gen}(1^\eta)$. The probabilistic encryption algorithm enc expects a key k and a plaintext m and returns a ciphertext $\text{enc}(k, m) \in \{0, 1\}^*$ or $\text{enc}(k, m) = \perp$ (where $\perp \notin \{0, 1\}^*$ is a special error symbol) if encryption fails. The deterministic decryption algorithm dec expects a key k and a ciphertext $c \in \{0, 1\}^*$ and returns the plaintext $\text{dec}(k, c) \in \{0, 1\}^*$ or $\text{dec}(k, c) = \perp$ if decryption fails.*

It is required that for every security parameter η and key k generated by $\text{gen}(1^\eta)$ it holds that i) $\text{enc}(k, m) = \perp$ if and only if $m \notin \text{dom}(\Sigma)$ and ii) $\text{dec}(k, \text{enc}(k, m)) = m$ for every plaintext $m \in \text{dom}(\Sigma)$.

We assume that every encryption scheme is associated with a polynomial q that bounds the runtime of the algorithms and the length of their description in some standard encoding. We say that Σ is q -bounded. For all symmetric encryption schemes considered in this paper, we assume that the key generation algorithm chooses keys uniformly at random from $\{0, 1\}^\eta$.

We define $\text{LR}(m_0, m_1, b) = m_b$ for every $b \in \{0, 1\}$ and $m_0, m_1 \in \{0, 1\}^*$ of the same length. If m_0 and m_1 are not of the same length, we define $\text{LR}(m_0, m_1, b) = \perp$.

Definition E.2 (IND-CPA security). *A symmetric encryption scheme Σ is called IND-CPA secure if for every probabilistic polynomial-time algorithm $A^{O(\cdot)}$ with access to an oracle O , the*

IND-CPA advantage of A with respect to Σ

$$\text{Adv}_{A,\Sigma}^{\text{IND-CPA}}(1^\eta, a) := \left| \Pr \left[\text{Exp}_{A,\Sigma}^{\text{IND-CPA}-1}(1^\eta, a) = 1 \right] - \Pr \left[\text{Exp}_{A,\Sigma}^{\text{IND-CPA}-0}(1^\eta, a) = 1 \right] \right|$$

is negligible as a function in η and a , where the experiment $\text{Exp}_{A,\Sigma}^{\text{IND-CPA}-b}$ ($b \in \{0, 1\}$) is defined as follows:

function $\text{Exp}_{A,\Sigma}^{\text{IND-CPA}-b}(1^\eta, a)$
 $k := \text{gen}(1^\eta)$
return $A^{\text{enc}(k, \text{LR}(\cdot, b))}(1^\eta)$

Definition E.3 (IND-CCA2 security). A symmetric encryption scheme Σ is called IND-CCA2 secure if for every probabilistic polynomial-time algorithm $A^{O_1(\cdot), O_2(\cdot)}$ with access to two oracles O_1, O_2 which never queries O_2 with a bit string returned by O_1 , the IND-CCA2 advantage of A with respect to Σ

$$\text{Adv}_{A,\Sigma}^{\text{IND-CCA2}}(1^\eta, a) := \left| \Pr \left[\text{Exp}_{A,\Sigma}^{\text{IND-CCA2}-1}(1^\eta, a) = 1 \right] - \Pr \left[\text{Exp}_{A,\Sigma}^{\text{IND-CCA2}-0}(1^\eta, a) = 1 \right] \right|$$

is negligible as a function in η and a , where the experiment $\text{Exp}_{A,\Sigma}^{\text{IND-CCA2}-b}$ ($b \in \{0, 1\}$) is defined as follows:

function $\text{Exp}_{A,\Sigma}^{\text{IND-CCA2}-b}(1^\eta, a)$
 $k := \text{gen}(1^\eta)$
return $A^{\text{enc}(k, \text{LR}(\cdot, b)), \text{dec}(k, \cdot)}(1^\eta)$

Definition E.4 (INT-CTXT security). A symmetric encryption scheme Σ is called INT-CTXT secure if for every probabilistic polynomial-time algorithm $A^{O_1(\cdot), O_2(\cdot)}$ with access to two oracles O_1, O_2 , the INT-CTXT advantage of A with respect to Σ

$$\text{Adv}_{A,\Sigma}^{\text{INT-CTXT}}(1^\eta, a) := \Pr \left[\text{Exp}_{A,\Sigma}^{\text{INT-CTXT}}(1^\eta, a) = 1 \right]$$

is negligible as a function in η and a , where the experiment $\text{Exp}_{A,\Sigma}^{\text{INT-CTXT}}$ is defined as follows:

function $\text{Exp}_{A,\Sigma}^{\text{INT-CTXT}}(1^\eta, a)$
 $k := \text{gen}(1^\eta)$
Run $A^{\text{enc}(k, \cdot), \text{dec}(k, \cdot)}(1^\eta)$
return 1 if A makes a query c to $\text{dec}(k, \cdot)$ such that $\text{dec}(k, c) \neq \perp$ and c was not previously returned by $\text{enc}(k, \cdot)$.
return 0, otherwise

E.2. Public-Key Encryption

Here we recall the definition of public-key encryption schemes and IND-CCA2 security notion.

Definition E.5. A public-key encryption scheme $\Sigma = (\text{gen}, \text{enc}, \text{dec})$ with plaintext domain $\text{dom}(\Sigma) \subseteq \{0, 1\}^*$ consists of three polynomial-time algorithms. The probabilistic key generation algorithm gen expects a security parameter η and returns a pair of keys (k_d, k_e) , the secret (or decryption) key k_d and the public (or encryption) key k_e . The probabilistic encryption algorithm enc expects a public key k_e and a plaintext m and returns a ciphertext $\text{enc}(k_e, m) \in \{0, 1\}^*$ or $\text{enc}(k_e, m) = \perp$ (where $\perp \notin \{0, 1\}^*$ is a special error symbol) if encryption fails. The deterministic decryption algorithm dec expects a private key k_d and a ciphertext $c \in \{0, 1\}^*$ and returns the plaintext $\text{dec}(k_d, c) \in \{0, 1\}^*$ or $\text{dec}(k_d, c) = \perp$ if decryption fails.

It is required that for every security parameter η and key pair (K_d, k_e) generated by $\text{gen}(1^\eta)$ it holds that i) $\text{enc}(k_e, m) = \perp$ if and only if $m \notin \text{dom}(\Sigma)$ and ii) $\text{dec}(k_d, \text{enc}(k_e, m)) = m$ for every plaintext $m \in \text{dom}(\Sigma)$.

We assume that every encryption scheme is associated with a polynomial q that bounds the runtime of the algorithms and the length of their description in some standard encoding. We say that Σ is q -bounded.

We define $\text{LR}(m_0, m_1, b) = m_b$ for every $b \in \{0, 1\}$ and $m_0, m_1 \in \{0, 1\}^*$ of the same length. If m_0 and m_1 are not of the same length, we define $\text{LR}(m_0, m_1, b) = \perp$.

Definition E.6 (IND-CCA2 security). A public-key encryption scheme Σ is called IND-CCA2 secure if for every probabilistic polynomial-time algorithm $A^{O_1(\cdot), O_2(\cdot)}$ with access to two oracles O_1, O_2 which never queries O_2 with a bit string returned by O_1 , the IND-CCA2 advantage of A with respect to Σ

$$\text{Adv}_{A, \Sigma}^{\text{IND-CCA2}}(1^\eta, a) := \left| \Pr \left[\text{Exp}_{A, \Sigma}^{\text{IND-CCA2-1}}(1^\eta, a) = 1 \right] - \Pr \left[\text{Exp}_{A, \Sigma}^{\text{IND-CCA2-0}}(1^\eta, a) = 1 \right] \right|$$

is negligible as a function in η and a , where the experiment $\text{Exp}_{A, \Sigma}^{\text{IND-CCA2-}b}$ ($b \in \{0, 1\}$) is defined as follows:

```

function  $\text{Exp}_{A, \Sigma}^{\text{IND-CCA2-}b}(1^\eta, a)$ 
   $(k_e, k_d) := \text{gen}(1^\eta)$ 
  return  $A^{\text{enc}(k_e, \text{LR}(\cdot, \cdot, b)), \text{dec}(k_d, \cdot)}(1^\eta)$ 

```

E.3. Message Authentication Codes (MACs)

In this section, we recall the definition of MACS and the UF-CMA security notion.

Definition E.7. A message authentication code (MAC) scheme $\Sigma = (\text{gen}, \text{mac}, \text{verify})$ consists of three poly-time algorithms. The probabilistic key generation algorithm gen expects a security parameter η and returns a key $\text{gen}(1^\eta)$. The (possibly) probabilistic MAC algorithm mac expects a key k and a message m and returns a message authentication code $\text{mac}(k, m)$. The deterministic verification algorithm verify expects a key k , a message m , and a message authentication code σ and returns $\text{verify}(k, m, \sigma) \in \{\text{true}, \text{false}\}$.

It is required that for every security parameter $\eta \in \mathbb{N}$, key k generated by $\text{gen}(1^\eta)$, and message $m \in \{0, 1\}^*$ it holds that $\text{verify}(k, m, \text{mac}(k, m)) = \text{true}$.

For all MAC schemes considered in this paper, we assume that the key generation algorithm chooses keys uniformly at random from $\{0, 1\}^\eta$

Definition E.8 (UF-CMA security). A MAC scheme Σ is called UF-CMA secure if for every probabilistic polynomial-time algorithm $A^{O_1(\cdot), O_2(\cdot)}$ with access to two oracles O_1, O_2 , the UF-CMA advantage of A with respect to Σ

$$\text{Adv}_{A, \Sigma}^{\text{UF-CMA}}(1^\eta, a) := \Pr \left[\text{Exp}_{A, \Sigma}^{\text{UF-CMA}}(1^\eta, a) = 1 \right]$$

is negligible as a function in η and a , where the experiment $\text{Exp}_{A, \Sigma}^{\text{UF-CMA}}$ is defined as follows:

function $\text{Exp}_{A, \Sigma}^{\text{UF-CMA}}(1^\eta, a)$

$k := \text{gen}(1^\eta)$

$(m, \sigma) = A^{\text{mac}(k, \cdot), \text{verify}(k, \cdot)}(1^\eta)$

return 1 if $\text{verify}(k, m, \sigma) = \text{true}$ and A has not previously called $\text{mac}(k, m)$.

return 0, otherwise.

E.4. Digital Signature Schemes

In this section, we recall the definition of digital signature schemes and the UF-CMA security notion.

Definition E.9. A (digital) signature scheme $\Sigma = (\text{gen}, \text{sig}, \text{verify})$ consists of three polynomial-time algorithms. The probabilistic key generation algorithm gen expects a security parameter η and returns a pair of keys (k_s, k_v) , the secret (or signing) key k_s and the public (or verification) key k_v . The (possibly) probabilistic signing algorithm sig expects a private key k_s and a message $m \in \{0, 1\}^*$ and returns a signature $\text{sig}(k_s, m)$. The deterministic verification algorithm verify expects a public key k_v , a message $m \in \{0, 1\}^*$, and a message authentication code σ and returns $\text{verify}(k_v, m, \sigma) \in \{\text{true}, \text{false}\}$.

It is required that for every security parameter $\eta \in \mathbb{N}$, key pair (k_s, k_v) generated by $\text{gen}(1^\eta)$, and message $m \in \{0, 1\}^*$ it holds that $\text{verify}(k_v, m, \text{sig}(k_s, m)) = \text{true}$.

Definition E.10 (UF-CMA security). A digital signature scheme Σ is called UF-CMA secure if for every probabilistic polynomial-time algorithm A^O with access to a signing oracle O , the UF-CMA advantage of A with respect to Σ

$$\text{Adv}_{A, \Sigma}^{\text{UF-CMA}}(1^\eta, a) := \Pr \left[\text{Exp}_{A, \Sigma}^{\text{UF-CMA}}(1^\eta, a) = 1 \right]$$

is negligible as a function in η and a , where the experiment $\text{Exp}_{A, \Sigma}^{\text{UF-CMA}}$ is defined as follows:

function $\text{Exp}_{A, \Sigma}^{\text{UF-CMA}}(1^\eta, a)$

$(k_s, k_v) := \text{gen}(1^\eta)$

$(m, \sigma) = A^{\text{sig}(k_s, \cdot)}(1^\eta, a, k_v)$

return 1 if $\text{verify}(k_v, m, \sigma) = \text{true}$ and A has not previously called $\text{sig}(k_s, m)$.

return 0, otherwise.

E.5. Decisional Diffie-Hellman

In this section, we recall the Decisional Diffie-Hellman (DDH) assumption. The DDH assumption is defined with respect to an algorithm $\text{GroupGen}(1^\eta)$ that runs in polynomial time in η (except for a negligible probability) and outputs a tuple of the form (G, n, g) of polynomial length (in η), where G is a group description, $n = |G|$, and g is a generator of G .

Definition E.11 (DDH assumption). *Let GroupGen be an algorithm as above. The DDH assumption holds for GroupGen if for every polynomial time algorithm A (in η and $|a|$) the DDH advantage of A*

$$\text{Adv}_{A, \text{GroupGen}}^{\text{DDH}}(1^\eta, a) := \left| \Pr \left[\text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-1}(1^\eta, a) = 1 \right] - \Pr \left[\text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-0}(1^\eta, a) = 1 \right] \right|$$

is negligible as a function in η and a , where the experiment $\text{Exp}_{A, \Sigma}^{\text{DDH}-b}$ ($b \in \{0, 1\}$) is defined as follows:

function $\text{Exp}_{A, \text{GroupGen}}^{\text{DDH}-b'}(1^\eta, a')$
 $(G, n, g) := \text{GroupGen}(1^\eta)$
 $a \xleftarrow{\$} \{1, \dots, n\}, b \xleftarrow{\$} \{1, \dots, n\}, c \xleftarrow{\$} \{1, \dots, n\}$
if $b' = 1$:
 return $A(1^\eta, a', (G, n, g, g^a, g^b, g^{ab}))$
else:
 return $A(1^\eta, a', (G, n, g, g^a, g^b, g^c))$

E.6. Pseudo-Random Functions

In this section, we recall the definition of secure PRFs, and define secure PRFs keyed with Diffie-Hellman group elements.

Let $h : \{0, 1\}^* \rightarrow \{0, 1\}^\eta$ be the following probabilistic, stateful algorithm. It maintains a set H which is initially empty. Upon input $s \in \{0, 1\}^*$, h returns x if there exists an x such that $(x, s) \in H$. Otherwise, h chooses x uniformly at random from $\{0, 1\}^\eta$, adds (x, s) to H , and returns x . Furthermore, let $\text{GroupGen}(1^\eta)$ be an algorithm that runs in polynomial time in η (except for a negligible probability) and outputs a tuple of the form (G, n, g) of polynomial length (in η), where G is a group description, $n = |G|$, and g is a generator of G .

Definition E.12 (PRF security). *Let $F = \{F_\eta\}_{\eta \in \mathbb{N}}$ with $F_\eta : \{0, 1\}^\eta \times \{0, 1\}^* \rightarrow \{0, 1\}^\eta$ be a family of efficiently computable functions. F is called a pseudo-random function family if for*

every polynomial time algorithm A (in η and $|a|$) the PRF advantage of A

$$\text{Adv}_{A,F,\text{GroupGen}}^{\text{PRF}}(1^\eta, a) := \left| \Pr \left[\text{Exp}_{A,F,\text{GroupGen}}^{\text{PRF}-1}(1^\eta, a) = 1 \right] \right. \\ \left. - \Pr \left[\text{Exp}_{A,F,\text{GroupGen}}^{\text{PRF}-0}(1^\eta, a) = 1 \right] \right|$$

is negligible as a function in η and a , where the experiment $\text{Exp}_{A,F,\text{GroupGen}}^{\text{PRF}-b}$ ($b \in \{0, 1\}$) is defined as follows:

function $\text{Exp}_{A,F,\text{GroupGen}}^{\text{PRF}-b}(1^\eta, a)$
 $k \xleftarrow{\$} \{0, 1\}^\eta$
if $b = 1$:
 $O(\cdot) := F_\eta(k, \cdot)$
else:
 $O(\cdot) := h(\cdot)$
return $A^{O(\cdot)}(1^\eta, a)$

Definition E.13 (PRF security on groups). Let $F = \{F_\eta\}_{\eta \in \mathbb{N}}$ with $F_\eta : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^\eta$ be a family of efficiently computable functions and let GroupGen be an algorithm as defined above. F is called a pseudo-random function family for GroupGen if for every polynomial time algorithm A (in η and $|a|$) the group PRF advantage of A

$$\text{Adv}_{A,F,\text{GroupGen}}^{\text{G-PRF}}(1^\eta, a) := \left| \Pr \left[\text{Exp}_{A,F,\text{GroupGen}}^{\text{G-PRF}-1}(1^\eta, a) = 1 \right] \right. \\ \left. - \Pr \left[\text{Exp}_{A,F,\text{GroupGen}}^{\text{G-PRF}-0}(1^\eta, a) = 1 \right] \right|$$

is negligible as a function in η and a , where the experiment $\text{Exp}_{A,F,\text{GroupGen}}^{\text{G-PRF}-b}$ ($b \in \{0, 1\}$) is defined as follows:

function $\text{Exp}_{A,F,\text{GroupGen}}^{\text{G-PRF}-b}(1^\eta, a)$
 $(G, n, g) := \text{GroupGen}(1^\eta)$
 $k \xleftarrow{\$} G$
if $b = 1$:
 $O(\cdot) := F_\eta(k, \cdot)$
else:
 $O(\cdot) := h(\cdot)$
return $A^{O(\cdot)}(1^\eta, a, (G, n, g))$

F. Formal Specifications of $\mathcal{F}_{\text{crypto}}$ and $\mathcal{P}_{\text{crypto}}$

Here we give the formal specifications of $\mathcal{F}_{\text{crypto}}$ and $\mathcal{P}_{\text{crypto}}$ using our template from the iUC framework. These specifications have already been explained in Section 5.1 and mainly serve for reference purposes.

Description of the protocol $\mathcal{F}_{\text{crypto}} = (\text{crypto})$:

Participating roles: crypto

Corruption model: custom ^a

Protocol parameters:

- $p \in \mathbb{Z}[x]$. *{Polynomial that bounds the runtime of algorithms provided by the adversary.}*
- $\mathcal{L} \in \{0, 1\}^*$. *{Algorithm for computing the leakage of an ideally encrypted message.}*
- **GroupGen**. *{Algorithm that generates a cyclical group for DH key exchanges.}*

^aAs explained in Section 5.1.1, $\mathcal{F}_{\text{crypto}}$ does not use the default corruption mechanism of the iUC framework where the adversary gains full control (in terms of message forwarding) over a corrupted entity. Instead, $\mathcal{F}_{\text{crypto}}$ implements a different type of corruption where the adversary can corrupt keys in order to disable security guarantees. This models that the adversary might get access to secret keys even without gaining full control over the user's computer.

Description of M_{crypto} :

Implemented role(s): crypto

Internal state:

- $(\text{enc}_{\text{auth}}, \text{dec}_{\text{auth}}, \text{enc}_{\text{unauth}}, \text{dec}_{\text{unauth}}, \text{enc}_{\text{pke}}, \text{dec}_{\text{pke}}, \text{mac}, \text{verify}_{\text{mac}}, \text{sig}, \text{verify}_{\text{sig}})$
 $\in (\{0, 1\}^* \cup \{\perp\})^{10} = (\perp, \perp, \perp, \perp)$. *{(Stateless) algorithms provided by adversary.}*
- $(G, n, g) \in \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^* = (\epsilon, 0, \epsilon)$. *{DH group.}*
- $\text{keys} \subseteq \{\text{authenc-key}, \text{unauthenc-key}, \text{mac-key}, \text{pre-key}, \text{dh-key}\} \times \{0, 1\}^* = \emptyset$.^a
{Set of all symmetric keys.}
- $\text{keys}_{\text{known}} \subseteq \text{keys} = \emptyset$. *{Set of known symmetric keys; we define $\text{keys}_{\text{unknown}} := \text{keys} \setminus \text{keys}_{\text{known}}$.}*
- $\text{keymap} : (\{0, 1\}^*)^3 \times \mathbb{N} \rightarrow \text{keys} \cup \{\perp\}$
{Stores keys corresponding to users and pointers. Initially \perp for all entries.}
- $\text{exp} \subseteq \mathbb{N} = \emptyset$. *{Set of all DH exponents.}*
- $\text{exp}_{\text{known}} \subseteq \text{exp} = \emptyset$. *{Set of known DH exponents; we define $\text{exp}_{\text{unknown}} := \text{exp} \setminus \text{exp}_{\text{known}}$.}*
- $\text{expmap} : (\{0, 1\}^*)^3 \times \mathbb{N} \rightarrow \{1, \dots, n\} \cup \{\perp\}$.
{Stores exponents corresponding to users and pointers. Initially \perp for all entries.}
- $\text{blockedDHShares} \subseteq G$. *{DH group elements that may not be generated by a GenExp request.}*
- $\text{pk}_{\text{pke}}, \text{sk}_{\text{pke}}, \text{pk}_{\text{sig}}, \text{sk}_{\text{sig}} : \{0, 1\}^* \rightarrow \{0, 1\}^* \cup \{\perp\}$. *{Mappings from PIDs to their public/private keys.}*
- $\text{nonceSet} \subseteq \{0, 1\}^* = \emptyset$. *{Set of existing nonces.}*
- $\text{corruptedKeys} \subseteq \{\text{ptr}, \text{psk}, \text{pke}, \text{sig}\} \times \{0, 1\}^* = \emptyset$. *{Set of corrupted keys.}*

In addition to the above, $\mathcal{F}_{\text{crypto}}$ also keeps track of past operations, which we keep implicit for better readability.

CheckID(*pid, sid, role*):

Accept all entities.

Initialization:

```

(G, n, g) ← GroupGen(1η).
send responsively (Init, (G, n, g)) to NET;
wait for (Init, (encauth, decauth, encunauth, decunauth, encpke, decpke, mac, verifymac, sig, verifysig)).
(encauth, decauth, encunauth, decunauth, encpke, decpke, mac, verifymac, sig, verifysig)
  ← (encauth, decauth, encunauth, decunauth, encpke, decpke, mac, verifymac, sig, verifysig).
if initialization was triggered by a message received from the network:
  send (Init, done) to NET.
    
```

Continue with Figure F.2.

^aDH keys of type **dh-key** from the group G are saved in this set using a representation as bit string. As explained in Section 5.1.2, this representation has a fixed length that depends on the group G (and potentially η) but otherwise is the same for all group elements of that group. We keep this implicit in what follows. In particular, when we check some input for group membership and when we check for well-tagged messages during decryption, then these checks implicitly include checking that group elements use the correct encoding.

Figure F.1.: The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part I).

Main:

recv (CorruptSignatureKey, pid) **from** NET: $\left\{ \begin{array}{l} \text{NET: Dynamic corruption of signature keys.} \\ \text{Note that corruption of PSKs and PKE keys is} \\ \text{static and handled during their generation.} \end{array} \right.$
 Add (sig, pid) to corruptedKeys.
reply (CorruptSignatureKey, ok).

recv (AddKey, k , knownStatus) **from** NET **s.t.** $k \notin \text{keys} \wedge \text{knownStatus} \in \{\text{known}, \text{unknown}\}$:
 Add k to keys $\{ \text{NET: Inform } \mathcal{F}_{\text{crypto}} \text{ of a new key.} \}$
if knownStatus = known:
 Add k to keys_{known}.

recv (New, t) **from** I/O **s.t.** $t \in \{\text{authenc-key}, \text{unauthenc-key}, \text{mac-key}, \text{pre-key}\}$:
send responsively (New, entity_{call}, t) **to** NET; $\{1. \text{ Symmetric key generation.}\}$
wait for (New, k) **s.t.** $(t, k) \notin \text{keys}$.
 Add (t, k) to keys. Create a new pointer ptr for entity_{call} to this key.^a
reply (New, ptr).

recv (GetPSK, t , name) **from** I/O **s.t.** $t \in \{\text{authenc-key}, \text{unauthenc-key}, \text{mac-key}, \text{pre-key}\}$:
send responsively (GetPSK, entity_{call}, t , name) **to** NET; $\{2. \text{ Establishing pre-shared keys.}\}$
wait for (GetPSK, corrupted, k) **s.t.** corrupted $\in \{\text{true}, \text{false}\}$.
 Check that (i) if a key (t, k') is recorded for $(t, name)$, then $k = k'$ and corrupted = false,
 (ii) if (psk, $(t, name)$) \in corruptedKeys, then corrupted = true,
 (iii) if corrupted = true, then $(t, k) \notin \text{keys}_{\text{unknown}}$,
 (iv) if no key (t, k') is recorded for $(t, name)$ and corrupted = false, then $(t, k) \notin \text{keys}$.
 If any of the above checks fails, go back to the start of this block (and therefore repeat the request).
 Add (t, k) to keys. Create a new pointer ptr for entity_{call} to (t, k) .
if corrupted = true:
 Add (psk, $(t, name)$) and (ptr, (entity_{call}, ptr)) to corruptedKeys. Add (t, k) to keys_{known}.^b
else:
 Record the key (t, k) as the PSK $(t, name)$.
reply (GetPSK, ptr).

recv (Store, t, k) **from** I/O **s.t.** t is a key type: $\{3. \text{ Storing symmetric keys.}\}$
 If $(t, k) \in \text{keys}_{\text{unknown}}$, then **reply** (Store, \perp). Otherwise, continue.
 Add (t, k) to keys and keys_{known}. Create a new pointer ptr for entity_{call} to (t, k) .
reply (Store, ptr).

recv (Retrieve, ptr) **from** I/O **s.t.** keymap[entity_{call}, ptr] = (t, k) : $\{4. \text{ Retrieving symmetric keys.}\}$
 Add (t, k) to keys_{known}.
reply (Retrieve, k).

recv (Equal?, ptr, ptr') **from** I/O **s.t.** keymap[entity_{call}, ptr] = (t, k) , keymap[entity_{call}, ptr'] = (t', k') :
if $t = t'$ and $k = k'$: $\{5. \text{ Testing equality of symmetric keys.}\}$
reply (Equal?, true).
else:
reply (Equal?, false).

recv GetPubKeyPKE, pid') **from** I/O: $\{6. \text{ Public encryption key request.}\}$
 InitAsymKey(pke, pid'). $\{ \text{This function is defined in Figure F.6} \}$
reply (GetPubKeyPKE, pk_{pke}[pid']).

The Main algorithm is continued in Figure F.3.

^aFormally, by the statement “create a pointer for a user to a key” we mean that $\mathcal{F}_{\text{crypto}}$ sets ptr to the next natural number that has not been used for a symmetric key pointer of that user yet (starting at 0). Then $\mathcal{F}_{\text{crypto}}$ sets keymap[pid_{call}, sid_{call}, role_{call}, ptr] $\leftarrow (t, k)$. Note that this statement is also defined analogously for Diffie-Hellman exponents, except that those use a separate counter for pointers and a separate map expmap for storing the exponents.

^bAs explained in Sections 5.1.2, the adversary is notified when a key (t, k) is added to keys_{known} or an exponent e is added to exp_{known}. Formally, this is done by sending a restricting message (AddedKnownKey, (t, k)) or (AddedKnownExponent, e), respectively, and waiting for an arbitrary response before continuing. We keep this implicit in what follows and just say “Add a key/exponent to keys_{known}/exp_{known}” to avoid clutter.

Figure F.2.: The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part II).

Main (continued):

recv GetPubKeySig, pid') from I/O: { 7. Public verification key request.
 InitAsymKey(sig, pid'). { cf. Figure F.6
reply (GetPubKeySig, $pk_{\text{sig}}[pid']$).

recv (Derive, ptr, t', s) from I/O s.t. $t' \in \{\text{authenc-key, unauthenc-key, mac-key, pre-key}\}$,
 $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k)$, $t \in \{\text{pre-key, dh-key}\}$:
 Set $\text{knownStatus} \leftarrow \text{unknown}$ if $(t, k) \in \text{keys}_{\text{unknown}}$; { 8. Key derivation.
 set $\text{knownStatus} \leftarrow \text{known}$ otherwise.
send responsively (Derive, $\text{entity}_{\text{call}}, t, k, t', s, \text{knownstatus}$) to NET;
wait for (Derive, k').
 Check that **if** exists k'' such that (t', k'') has been derived from (t, k) with seed s **then** $k' = k''$
 else if $(t, k) \in \text{keys}_{\text{unknown}}$ **then** $(t', k') \notin \text{keys}$
 else $(t', k') \notin \text{keys}_{\text{unknown}}$.
 If the above check fails, go back to the start of this block (and therefore repeat the request).
 Add (t', k') to keys . If $(t, k) \in \text{keys}_{\text{known}}$, then also add (t', k') to $\text{keys}_{\text{known}}$.
 Record (t', k') as derived from (t, k) with seed s . Create a new pointer ptr for $\text{entity}_{\text{call}}$ to (t', k') .
reply (Derive, ptr).

recv (Enc, ptr, x) from I/O s.t. $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k)$, $t \in \{\text{authenc-key, unauthenc-key}\}$,
 and x is a well-tagged plaintext:
 Set $\text{ideal} \leftarrow \text{false}$ if $(t, k) \in \text{keys}_{\text{known}}$; { 9. Symmetric encryption.
 otherwise set $\text{ideal} \leftarrow \text{true}$.
 Obtain x' from x by replacing every pointer $\text{Tag}_{t'}(ptr')$ in x (where t' is a key type) by $\text{Tag}_{t'}(k')$
 where $(t', k') = \text{keymap}[\text{entity}_{\text{call}}, ptr']$. If this is not possible because there is a pointer ptr' that is
 not defined for the user (i.e., $\text{keymap}[\text{entity}_{\text{call}}, ptr'] = \perp$), then **reply** (Enc, \perp).
if $\text{ideal} = \text{true}$:
 $\bar{x} \leftarrow \mathcal{L}(1^n, x')$, $y \leftarrow \text{enc}_t^{(p)}(k, \bar{x})$, $\bar{x}' \leftarrow \text{dec}_t^{(p)}(k, y)$.^a
 If $\bar{x} = \bar{x}'$, record (x', y) for key (t, k) . Otherwise, set $y \leftarrow \perp$.
else:
 $y \leftarrow \text{enc}_t^{(p)}(k, x')$.
 If $y \neq \perp$ and $\text{ideal} = \text{false}$, then: For every key $\text{Tag}_{t'}(k')$ in x' (i.e., where t' is a key type) add
 (t', k') to $\text{keys}_{\text{known}}$.
reply (Enc, y).

recv (Dec, ptr, y) from I/O s.t. $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k)$, $t \in \{\text{authenc-key, unauthenc-key}\}$:
 Set $\text{ideal} \leftarrow \text{false}$ if $(t, k) \in \text{keys}_{\text{known}}$ or { 10. Symmetric decryption.
 $(t = \text{unauthenc-key}$ and there exists no x' such that (x', y) has been recorded (during encryption)
 for (t, k)); otherwise set $\text{ideal} \leftarrow \text{true}$.
if $\text{ideal} = \text{true}$:
 if $\exists x_1, x_2 : x_1 \neq x_2$ and (x_1, y) and (x_2, y) have been recorded (during encryption) for (t, k) :
 $x' \leftarrow \perp$.
 else if $\exists x'' : (x'', y)$ has been recorded (during encryption) for (t, k) :
 $x' \leftarrow x''$.
 else:
 $x' \leftarrow \perp$. { Note that this case can only occur for $t = \text{authenc-key}$
else:
 $x' \leftarrow \text{dec}_t^{(p)}(k, y)$.
 If x' is not well-tagged, then $x' \leftarrow \perp$.
 If $x' \neq \perp$, $\text{ideal} = \text{false}$, and there exists a key $\text{Tag}_{t'}(k')$ in x' such that $(t', k') \in \text{keys}_{\text{unknown}}$, then
 set $x' \leftarrow \perp$.
 If $x' \neq \perp$ and $\text{ideal} = \text{false}$, then for every key $\text{Tag}_{t'}(k')$ in x' add (t', k') to keys and $\text{keys}_{\text{known}}$.
 Let x be the bit string obtained from x' by doing the following for every key $\text{Tag}_{t'}(k')$ in x' : Create
 a new pointer ptr' for $\text{entity}_{\text{call}}$ to (t', k') and replace $\text{Tag}_{t'}(k')$ by $\text{Tag}_{t'}(ptr)$ in x' .
reply (Dec, x).

The **Main** algorithm is continued in Figure F.4.

^aI.e., depending on t use either enc_{auth} or $\text{enc}_{\text{unauth}}$ respectively dec_{auth} or $\text{dec}_{\text{unauth}}$ for encryption/decryption.

Figure F.3.: The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part III).

Main (continued):

recv (PKEnc, pid' , pk , x) **from** I/O **s.t.** x is a well-tagged plaintext: {11. Encryption under public keys.}
{cf. Figure F.6}

InitAsymKey(pke , pid').
Set $ideal \leftarrow \text{false}$ if $(pke, pid') \in \text{corruptedKeys}$ or $pk \neq pk_{pke}[pid']$;
otherwise set $ideal \leftarrow \text{true}$.
Obtain x' from x by replacing every pointer $\text{Tag}_{t'}(ptr')$ in x (where t' is a key type) by $\text{Tag}_{t'}(k')$ where $(t', k') = \text{keymap}[\text{entity}_{\text{call}}, ptr']$. If this is not possible because there is a pointer ptr' that is not defined for the user (i.e., $\text{keymap}[\text{entity}_{\text{call}}, ptr'] = \perp$), then **reply** (Enc, \perp).
if $ideal = \text{true}$:
 $\bar{x} \leftarrow \mathcal{L}(1^n, x')$, $y \leftarrow \text{enc}_{pke}^{(p)}(pk, \bar{x})$, $\bar{x}' \leftarrow \text{dec}_{pke}^{(p)}(\text{sk}_{pke}[pid'], y)$.
If $\bar{x} = \bar{x}'$, record (x', y) for party pid' . Otherwise, set $y \leftarrow \perp$.
else:
 $y \leftarrow \text{enc}_{pke}^{(p)}(pk, x')$.
If $y \neq \perp$ and $ideal = \text{false}$, then: For every key $\text{Tag}_{t'}(k')$ in x' (i.e., where t' is a key type) add (t', k') to $\text{keys}_{\text{known}}$.
reply (PKEnc, y).

recv (PKDec, y) **from** I/O: {12. Decryption with private keys.}
{cf. Figure F.6}

InitAsymKey(pke , pid_{call}).
Set $ideal \leftarrow \text{false}$ if $(pke, pid_{\text{call}}) \in \text{corruptedKeys}$ or there exists no x' such that (x', y) has been recorded (during encryption) for party pid_{call} ; otherwise set $ideal \leftarrow \text{true}$.
if $ideal = \text{true}$:
if $\exists x_1, x_2 : x_1 \neq x_2$ and (x_1, y) and (x_2, y) have been recorded (during encryption) for pid_{call} :
 $x' \leftarrow \perp$.
else:
Let x'' be the unique value such that (x'', y) has been recorded (during encryption) for pid_{call} .
 $x' \leftarrow x''$.
else:
 $x' \leftarrow \text{dec}_{pke}^{(p)}(\text{sk}_{pke}[pid_{\text{call}}], y)$.
If x' is not well-tagged, then $x' \leftarrow \perp$.
If $x' \neq \perp$, $ideal = \text{false}$, and there exists a key $\text{Tag}_{t'}(k')$ in x' such that $(t', k') \in \text{keys}_{\text{unknown}}$, then set $x' \leftarrow \perp$.
If $x' \neq \perp$ and $ideal = \text{false}$, then for every key $\text{Tag}_{t'}(k')$ in x' add (t', k') to keys and $\text{keys}_{\text{known}}$.
Let x be the bit string obtained from x' by doing the following for every key $\text{Tag}_{t'}(k')$ in x' : Create a new pointer ptr' for $\text{entity}_{\text{call}}$ to (t', k') and replace $\text{Tag}_{t'}(k')$ by $\text{Tag}_{t'}(ptr)$ in x' .
reply (PKDec, x).

recv (MAC, ptr , x) **from** I/O **s.t.** $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k)$, $t = \text{mac-key}$: {13. Creating MACs.}
 $\sigma \leftarrow \text{mac}^{(p)}(k, x)$ and $b \leftarrow \text{verify}_{\text{mac}}^{(p)}(k, x, \sigma)$. If $b \neq \text{true}$, then $\sigma \leftarrow \perp$.
If $(t, k) \in \text{keys}_{\text{unknown}}$ and $\sigma \neq \perp$, then record x for key (t, k) .
reply (MAC, σ).

recv (MACVerify, ptr , x , σ) **from** I/O **s.t.** $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k)$, $t = \text{mac-key}$: {14. Verifying MACs.}
 $b \leftarrow \text{verify}_{\text{mac}}^{(p)}(k, x, \sigma)$.
If $b = \text{true}$, $(t, k) \in \text{keys}_{\text{unknown}}$, and (upon MACing) x has not been recorded for (t, k) , then $b \leftarrow \perp$.
reply (MACVerify, b).

recv (Sign, x) **from** I/O: {15. Creating Signatures.}
{cf. Figure F.6}

InitAsymKey(sig , pid_{call}).
 $\sigma \leftarrow \text{sig}^{(p)}(\text{sk}_{\text{sig}}[pid_{\text{call}}], x)$ and $b \leftarrow \text{verify}_{\text{sig}}^{(p)}(\text{pk}_{\text{sig}}[pid_{\text{call}}], x, \sigma)$. If $b \neq \text{true}$, then $\sigma \leftarrow \perp$.
If $(\text{sig}, pid_{\text{call}}) \notin \text{corruptedKeys}$ and $\sigma \neq \perp$, then record x for party pid_{call} .
reply (Sign, σ).

The Main algorithm is continued in Figure F.5.

Figure F.4.: The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part IV).

Main (continued):

rcv (SigVerify, pid' , pk , x , σ) **from** I/O: {16. Verifying Signatures.
 InitAsymKey(sig, pid'). {cf. Figure F.6
 $b \leftarrow \text{verify}_{\text{sig}}^{(p)}(pk, x, \sigma)$.
 If $b = \text{true}$, $pk = \text{pk}_{\text{sig}}[pid']$, $(\text{sig}, pid') \notin \text{corruptedKeys}$, and (upon signing) x has not been recorded for party pid' , then $b \leftarrow \perp$.
reply (SigVerify, b).

rcv NewNonce **from** I/O: {17. Generating fresh Nonces.
send responsively (NewNonce, $\text{entity}_{\text{call}}$) **to** NET;
wait for (NewNonce, nonce) **s.t.** $\text{nonce} \notin \text{nonceSet}$.
 Add nonce to nonceSet .
reply (NewNonce, nonce).

rcv (IsSymKeyCorrupt?, ptr) **from** I/O: {18. Corruption status of symmetric keys.
 Set $b \leftarrow \text{true}$ if $(ptr, (\text{entity}_{\text{call}}, ptr)) \in \text{corruptedKeys}$;
 otherwise set $b \leftarrow \text{false}$.
reply (IsSymKeyCorrupt?, b).

rcv (IsPKEncKeyCorrupt?, pid') **from** I/O: {18. Corruption status of public encryption keys.
 InitAsymKey(pke, pid'). {cf. Figure F.6
 Set $b \leftarrow \text{true}$ if $(\text{pke}, pid') \in \text{corruptedKeys}$;
 otherwise set $b \leftarrow \text{false}$.
reply (IsPKEncKeyCorrupt?, b).

rcv (IsSigKeyCorrupt?, pid') **from** I/O: {19. Corruption status of public signing keys.
 InitAsymKey(sig, pid'). {cf. Figure F.6
 Set $b \leftarrow \text{true}$ if $(\text{sig}, pid') \in \text{corruptedKeys}$;
 otherwise set $b \leftarrow \text{false}$.
reply (IsSigKeyCorrupt?, b).

rcv GetDHGroup **from** I/O: {20. Getting generated group.
reply (GetDHGroup, (G, n, g)).

rcv GenExp **from** I/O: {21. Generating DH exponents.
send responsively (GenExp, $\text{entity}_{\text{call}}, t$) **to** NET;
wait for (GenExp, e) **s.t.** $e \in \{1, \dots, n\}$, $e \notin \text{exp}$, and $g^e \notin \text{blockedDHShares}$.
 Add e to exp . Create a new pointer ptr for $\text{entity}_{\text{call}}$ to this exponent.
reply (GenExp, ptr, g^e).

rcv (BlockDHShare, h) **from** I/O **s.t.** $h \in G$: {22. Blocking DH group elements.
 Add h to blockedDHShares .
reply (BlockDHShare, ok).

rcv (StoreExp, e) **from** I/O **s.t.** $e \in \{1, \dots, n\}$: {23. Storing DH exponents.
 If $e \in \text{exp}_{\text{unknown}}$, then **reply** (Store, \perp). Otherwise, continue.
 Add e to exp and $\text{exp}_{\text{known}}$. Create a new pointer ptr for $\text{entity}_{\text{call}}$ to this exponent.
reply (StoreExp, ptr).

rcv (RetrieveExp, ptr) **from** I/O **s.t.** $\text{expmap}[\text{entity}_{\text{call}}, ptr] = e (\neq \perp)$:
 Add e to $\text{exp}_{\text{known}}$. {24. Retrieving DH exponents.
reply (RetrieveExp, e).

The **Main** algorithm is continued in Figure F.6.

Figure F.5.: The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part V).

Description of M_{crypto} (continued):

Main (continued):

```

recv (GenDHKey, ptr, h) from I/O s.t.  $h \in G$  and  $\text{expmap}[\text{entity}_{\text{call}}, \text{ptr}] = e$  ( $\neq \perp$ ):
  Add  $h$  to blockedDHShares.
  if  $h = g^e$ :
    Add  $e$  to  $\text{exp}_{\text{known}}$ .
  if some key (dh-key,  $k$ ) has already been recorded for  $(g^e, h)$  or  $(h, g^e)$ :
    Create a new pointer  $\text{ptr}'$  for  $\text{entity}_{\text{call}}$  to key (dh-key,  $k$ ).
    reply (GenDHKey,  $\text{ptr}'$ ).
  else:
    if  $e \in \text{exp}_{\text{unknown}}$  and  $\exists d \in \text{exp}_{\text{unknown}}$  such that  $g^d = h$ :
      send responsively (GenDHKey,  $\text{entity}_{\text{call}}, e, d, \text{unknown}$ ) to NET;
      wait for (GenDHKey,  $k$ ) s.t.  $k \in G \wedge (\text{dh-key}, k) \notin \text{keys}$ .
    else:
      send responsively (GenDHKey,  $\text{entity}_{\text{call}}, e, h, \text{known}$ ) to NET;
      wait for (GenDHKey,  $k$ ) s.t.  $k \in G \wedge (\text{dh-key}, k) \notin \text{keys}_{\text{unknown}}$ .
      if  $\nexists d \in \text{exp}$  such that  $g^d = h$ :
        Add  $e$  to  $\text{exp}_{\text{known}}$ .
  Record the key (dh-key,  $k$ ) for the DH key shares  $(g^e, h)$ .
  Create a new pointer  $\text{ptr}'$  for  $\text{entity}_{\text{call}}$  to key (dh-key,  $k$ ).
  reply (GenDHKey,  $\text{ptr}'$ ).

```

{25. Generating DH keys.

{ DH key generation is deterministic.

{ Ideal DH key generation.

{ DH key generation without security guarantees.

Helperfunction:

```

function InitAsymKey(type, pid)
  if type = pke and  $\text{pk}_{\text{pke}}[\text{pid}] = \perp$ :
    send responsively (InitAsymKey, pke, pid) to NET;
    wait for (InitAsymKey, corrupted, pk, sk) s.t. corrupted  $\in \{\text{true}, \text{false}\}$ ,  $\text{pk} \neq \perp$ ,  $\text{sk} \neq \perp$ .
     $\text{pk}_{\text{pke}}[\text{pid}] \leftarrow \text{pk}$ ,  $\text{sk}_{\text{pke}}[\text{pid}] \leftarrow \text{sk}$ .
    if corrupted = true:
      Add (pke, pid) to corruptedKeys.
  else if type = sig and  $\text{pk}_{\text{sig}}[\text{pid}] = \perp$ :
    send responsively (InitAsymKey, sig, pid) to NET;
    wait for (InitAsymKey, corrupted, pk, sk) s.t. corrupted  $\in \{\text{true}, \text{false}\}$ ,  $\text{pk} \neq \perp$ ,  $\text{sk} \neq \perp$ .
     $\text{pk}_{\text{sig}}[\text{pid}] \leftarrow \text{pk}$ ,  $\text{sk}_{\text{sig}}[\text{pid}] \leftarrow \text{sk}$ .
    if corrupted = true:
      Add (sig, pid) to corruptedKeys.

```

Figure F.6.: The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part VI).

Description of the protocol $\mathcal{P}_{\text{crypto}} = (\text{crypto})$:

Participating roles: `crypto`

Corruption model: `custom`

Protocol parameters:

- $\Sigma_{\text{auth-enc}} = (\text{enc}_{\text{auth}}, \text{dec}_{\text{auth}})$, $\Sigma_{\text{unauth-enc}} = (\text{enc}_{\text{unauth}}, \text{dec}_{\text{unauth}})$, $\Sigma_{\text{pub}} = (\text{gen}_{\text{pke}}, \text{enc}_{\text{pke}}, \text{dec}_{\text{pke}})$.
- $\Sigma_{\text{mac}} = (\text{mac}, \text{verify}_{\text{mac}})$, $\Sigma_{\text{sig}} = (\text{gen}_{\text{sig}}, \text{sig}, \text{verify}_{\text{sig}})$.
- $F = \{F_{\eta}\}_{\eta \in \mathbb{N}}$, $F' = \{F'_{\eta}\}_{\eta \in \mathbb{N}}$.
- `GroupGen`.

Description of M_{crypto} :

Implemented role(s): `crypto`

Internal state:

- $(G, n, g) \in \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^* = (\epsilon, 0, \epsilon)$. *{DH group.}*
- $\text{keymap} : (\{0, 1\}^*)^3 \times \mathbb{N} \rightarrow (\{\text{authenc-key}, \text{unauthenc-key}, \text{mac-key}, \text{pre-key}, \text{dh-key}\} \times \{0, 1\}^*) \cup \{\perp\}$
{Stores keys corresponding to users and pointers. Initially \perp for all entries.}
- $\text{expmap} : (\{0, 1\}^*)^3 \times \mathbb{N} \rightarrow \{1, \dots, n\} \cup \{\perp\}$.
{Stores exponents corresponding to users and pointers. Initially \perp for all entries.}
- $\text{pk}_{\text{pke}}, \text{sk}_{\text{pke}}, \text{pk}_{\text{sig}}, \text{sk}_{\text{sig}} : \{0, 1\}^* \rightarrow \{0, 1\}^* \cup \{\perp\}$. *{Mappings from PIDs to their public/private keys.}*
- $\text{corruptedKeys} \subseteq \{\text{ptr}, \text{psk}, \text{pke}, \text{sig}\} \times \{0, 1\}^* = \emptyset$. *{Set of corrupted keys.}*

In addition to the above, just as $\mathcal{F}_{\text{crypto}}$ the realization $\mathcal{P}_{\text{crypto}}$ also keeps track of some of the past operations, in particular for the purpose of generating new pointers. We keep this implicit in what follows.

CheckID(*pid, sid, role*):

Accept all entities.

Initialization:

$(G, n, g) \leftarrow \text{GroupGen}(1^n)$.

Main:

recv (`CorruptSignatureKey, pid`) **from** `NET`: *{NET: Dynamic corruption of signature keys.}*
Add (`sig, pid`) to `corruptedKeys`.
reply (`CorruptSignatureKey, ok`).

recv (`New, t`) **from** `I/O` **s.t.** $t \in \{\text{authenc-key}, \text{unauthenc-key}, \text{mac-key}, \text{pre-key}\}$:

Sample $k \xleftarrow{\$} \{0, 1\}^n$. *{1. Symmetric key generation.}*

Create a new pointer *ptr* for `entitycall` to the key (*t, k*).

reply (`New, ptr`).

Continue with Figure F.8.

Figure F.7.: The ideal functionality for cryptographic primitives $\mathcal{P}_{\text{crypto}}$ (Part I).

Description of M_{crypto} (continued):

Main:

recv (GetPSK, $t, name$) **from** I/O **s.t.** $t \in \{\text{authenc-key, unauthenc-key, mac-key, pre-key}\}$:
if a key (t, k) is recorded for $(t, name)$: {2. Establishing pre-shared keys.}
 Create a new pointer ptr for $\text{entity}_{\text{call}}$ to the key (t, k) .
 reply (New, ptr).
else:
 if $(psk, name) \in \text{corruptedKeys}$:
 send responsively (GetPSK, $\text{entity}_{\text{call}}, t, name$) **to** NET;
 wait for (GetPSK, k).
 Create a new pointer ptr for $\text{entity}_{\text{call}}$ to the key (t, k) .
 Add $(ptr, (\text{entity}_{\text{call}}, ptr))$ to corruptedKeys
 reply (GetPSK, ptr).
 else:
 send responsively (GetPSK, $\text{entity}_{\text{call}}, t, name$) **to** NET;
 wait for (GetPSK, $corrupted, k$) **s.t.** $corrupted \in \{\text{true, false}\}$.
 Create a new pointer ptr for $\text{entity}_{\text{call}}$ to the key (t, k) .
 if $corrupted = \text{true}$:
 Add $(psk, (t, name))$ and $(ptr, (\text{entity}_{\text{call}}, ptr))$ to corruptedKeys .
 else:
 Record the key (t, k) as the PSK $(t, name)$.
 reply (GetPSK, ptr).

recv (Store, t, k) **from** I/O **s.t.** t is a key type: {3. Storing symmetric keys.}
 Create a new pointer ptr for $\text{entity}_{\text{call}}$ to (t, k) .
 reply (Store, ptr).

recv (Retrieve, ptr) **from** I/O **s.t.** $\left[\begin{smallmatrix} \text{keymap}[\text{entity}_{\text{call}}, ptr] \\ = (t, k) \end{smallmatrix} \right]$: {4. Retrieving symmetric keys.}
 reply (Retrieve, k).

recv (Equal?, ptr, ptr') **from** I/O **s.t.** $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k), \text{keymap}[\text{entity}_{\text{call}}, ptr'] = (t', k')$:
 if $t = t'$ and $k = k'$: {5. Testing equality of symmetric keys.}
 reply (Equal?, true).
 else:
 reply (Equal?, false).

recv GetPubKeyPKE, pid') **from** I/O: {6. Public encryption key request.}
 InitAsymKey(pke, pid'). {This function is defined in Figure F.10}
 reply (GetPubKeyPKE, $pk_{pke}[pid']$).

recv GetPubKeySig, pid') **from** I/O: {7. Public verification key request.}
 InitAsymKey(sig, pid'). {cf. Figure F.10}
 reply (GetPubKeySig, $pk_{sig}[pid']$).

recv (Derive, ptr, t', s) **from** I/O **s.t.** $t' \in \{\text{authenc-key, unauthenc-key, mac-key, pre-key}\},$
 $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k), t \in \{\text{pre-key, dh-key}\}$: {8. Key derivation.}
 if $t = \text{pre-key}$:
 $k' \leftarrow F_{\eta}(k, \text{Tag}_{t'}(s))$.
 else:
 $k' \leftarrow F'_{\eta}(k, \text{Tag}_{t'}(s))$.
 Create a new pointer ptr for $\text{entity}_{\text{call}}$ to (t', k') .
 reply (Derive, ptr).

The Main algorithm is continued in Figure F.9.

Figure F.8.: The ideal functionality for cryptographic primitives $\mathcal{P}_{\text{crypto}}$ (Part II).

Main (continued):

recv (Enc, ptr , x) **from** I/O **s.t.** $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k)$, $t \in \{\text{authenc-key}, \text{unauthenc-key}\}$,
and x is a well-tagged plaintext: {9. Symmetric encryption.
Obtain x' from x by replacing every pointer $\text{Tag}_{t'}(ptr')$ in x (where t' is a key type) by $\text{Tag}_{t'}(k')$
where $(t', k') = \text{keymap}[\text{entity}_{\text{call}}, ptr']$. If this is not possible because there is a pointer ptr' that is
not defined for the user (i.e., $\text{keymap}[\text{entity}_{\text{call}}, ptr'] = \perp$), then **reply** (Enc, \perp).
 $y \leftarrow \text{enc}_t(k, x')$.^a
reply (Enc, y).

recv (Dec, ptr , y) **from** I/O **s.t.** $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k)$, $t \in \{\text{authenc-key}, \text{unauthenc-key}\}$:
 $x' \leftarrow \text{dec}_t(k, y)$. {10. Symmetric decryption.
If x' is not well-tagged, then $x' \leftarrow \perp$.
Let x be the bit string obtained from x' by doing the following for every key $\text{Tag}_{t'}(k')$ in x' : Create
a new pointer ptr' for $\text{entity}_{\text{call}}$ to (t', k') and replace $\text{Tag}_{t'}(k')$ by $\text{Tag}_{t'}(ptr)$ in x' .
reply (Dec, x).

recv (PKEnc, pid' , pk , x) **from** I/O **s.t.** x is a well-tagged plaintext: {11. Encryption under public keys.
InitAsymKey(pke , pid'). {cf. Figure F.10
Obtain x' from x by replacing every pointer $\text{Tag}_{t'}(ptr')$ in x (where t' is a key type) by $\text{Tag}_{t'}(k')$
where $(t', k') = \text{keymap}[\text{entity}_{\text{call}}, ptr']$. If this is not possible because there is a pointer ptr' that is
not defined for the user (i.e., $\text{keymap}[\text{entity}_{\text{call}}, ptr'] = \perp$), then **reply** (Enc, \perp).
 $y \leftarrow \text{enc}_{pke}(pk, x')$.
reply (PKEnc, y).

recv (PKDec, y) **from** I/O: {12. Decryption with private keys.
InitAsymKey(pke , pid_{call}). {cf. Figure F.10
 $x' \leftarrow \text{dec}_{pke}(\text{sk}_{pke}[\text{pid}_{\text{call}}], y)$.
If x' is not well-tagged, then $x' \leftarrow \perp$.
Let x be the bit string obtained from x' by doing the following for every key $\text{Tag}_{t'}(k')$ in x' : Create
a new pointer ptr' for $\text{entity}_{\text{call}}$ to (t', k') and replace $\text{Tag}_{t'}(k')$ by $\text{Tag}_{t'}(ptr)$ in x' .
reply (PKDec, x).

recv (MAC, ptr , x) **from** I/O **s.t.** $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k)$, $t = \text{mac-key}$: {13. Creating MACs.
 $\sigma \leftarrow \text{mac}(k, x)$.
reply (MAC, σ).

recv (MACVerify, ptr , x , σ) **from** I/O **s.t.** $\text{keymap}[\text{entity}_{\text{call}}, ptr] = (t, k)$, $t = \text{mac-key}$: {14. Verifying MACs.
 $b \leftarrow \text{verify}_{\text{mac}}(k, x, \sigma)$.
reply (MACVerify, b).

recv (Sign, x) **from** I/O: {15. Creating Signatures.
InitAsymKey(sig , pid_{call}). {cf. Figure F.10
 $\sigma \leftarrow \text{sig}(\text{sk}_{\text{sig}}[\text{pid}_{\text{call}}], x)$.
reply (Sign, σ).

recv (SigVerify, pid' , pk , x , σ) **from** I/O: {16. Verifying Signatures.
InitAsymKey(sig , pid'). {cf. Figure F.10
 $b \leftarrow \text{verify}_{\text{sig}}(pk, x, \sigma)$.
reply (SigVerify, b).

recv NewNonce **from** I/O: {17. Generating fresh Nonces.
 $\text{nonce} \xleftarrow{\$} \{0, 1\}^\eta$.
reply (NewNonce, nonce).

The **Main** algorithm is continued in Figure F.10.

^aI.e., depending on t use either enc_{auth} or $\text{enc}_{\text{unauth}}$ for encryption.

Figure F.9.: The ideal functionality for cryptographic primitives $\mathcal{P}_{\text{crypto}}$ (Part III).

Main (continued):

rcv (IsSymKeyCorrupt?, ptr) **from** I/O: {18. Corruption status of symmetric keys.
 Set $b \leftarrow \text{true}$ if $(ptr, (\text{entity}_{\text{call}}, ptr)) \in \text{corruptedKeys}$;
 otherwise set $b \leftarrow \text{false}$.
reply (IsSymKeyCorrupt?, b).

rcv (IsPKEncKeyCorrupt?, pid') **from** I/O: {18. Corruption status of public encryption keys.
 InitAsymKey(pke, pid'). {cf. Figure F.10
 Set $b \leftarrow \text{true}$ if $(pke, pid') \in \text{corruptedKeys}$;
 otherwise set $b \leftarrow \text{false}$.
reply (IsPKEncKeyCorrupt?, b).

rcv (IsSigKeyCorrupt?, pid') **from** I/O: {19. Corruption status of public signing keys.
 InitAsymKey(sig, pid'). {cf. Figure F.10
 Set $b \leftarrow \text{true}$ if $(sig, pid') \in \text{corruptedKeys}$;
 otherwise set $b \leftarrow \text{false}$.
reply (IsSigKeyCorrupt?, b).

rcv GetDHGroup **from** I/O: {20. Getting generated group.
reply (GetDHGroup, (G, n, g)).

rcv GenExp **from** I/O: {21. Generating DH exponents.
 $e \xleftarrow{\$} \{1, \dots, n\}$. Create a new pointer ptr for $\text{entity}_{\text{call}}$ to this exponent.
reply (GenExp, ptr, g^e).

rcv (BlockDHShare, h) **from** I/O s.t. $h \in G$: {22. Blocking DH group elements.
reply (BlockDHShare, ok).

rcv (StoreExp, e) **from** I/O s.t. $e \in \{1, \dots, n\}$: {23. Storing DH exponents.
 Create a new pointer ptr for $\text{entity}_{\text{call}}$ to the exponent e .
reply (StoreExp, ptr).

rcv (RetrieveExp, ptr) **from** I/O s.t. $\text{expmap}[\text{entity}_{\text{call}}, ptr] = e (\neq \perp)$:
reply (RetrieveExp, e). {24. Retrieving DH exponents.

rcv (GenDHKey, ptr, h) **from** I/O s.t. $h \in G$ and $\text{expmap}[\text{entity}_{\text{call}}, ptr] = e (\neq \perp)$:
 $k \leftarrow h^e$. Create a new pointer ptr' for $\text{entity}_{\text{call}}$ to key $(\text{dh-key}, k)$. {25. Generating DH keys.
reply (GenDHKey, ptr').

Helperfunction:

function InitAsymKey($type, pid$)
if $type = pke$ and $pk_{pke}[pid] = \perp$:
send responsively (InitAsymKey, pke, pid) **to** NET;
wait for (InitAsymKey, $corrupted, pk, sk$) **s.t.** $corrupted \in \{\text{true}, \text{false}\}, pk \neq \perp, sk \neq \perp$.
if $corrupted = \text{true}$:
 Add (pke, pid) to corruptedKeys .
 $pk_{pke}[pid] \leftarrow pk, sk_{pke}[pid] \leftarrow sk$.
else:
 $(sk_{pke}[pid], pk_{pke}[pid]) \leftarrow \text{gen}_{pke}(1^n)$.
else if $type = sig$ and $pk_{sig}[pid] = \perp$:
send responsively (InitAsymKey, sig, pid) **to** NET;
wait for (InitAsymKey, $corrupted, pk, sk$) **s.t.** $corrupted \in \{\text{true}, \text{false}\}, pk \neq \perp, sk \neq \perp$.
if $corrupted = \text{true}$:
 Add (pke, pid) to corruptedKeys .
 $pk_{sig}[pid] \leftarrow pk, sk_{sig}[pid] \leftarrow sk$.
else:
 $(sk_{sig}[pid], pk_{sig}[pid]) \leftarrow \text{gen}_{sig}(1^n)$.

Figure F.10.: The ideal functionality for cryptographic primitives $\mathcal{F}_{\text{crypto}}$ (Part IV).

G. Formal Specifications of $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ and $\mathcal{F}_{\text{key-use}}^{\text{UA}}$

Here we give the formal specifications of $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ and $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ using our template from the iUC framework. These specifications have already been explained in Section 5.2 and mainly serve for reference purposes.

Description of the protocol $\mathcal{F}_{\text{key-use}}^{\text{MA}} = (\text{initiator}, \text{responder})$:

Participating roles: $\{\text{initiator}, \text{responder}\}$
Corruption model: dynamic corruption with secure erasures
Protocol parameters:
<ul style="list-style-type: none"> - $t_{\text{key}} \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}$.

$\left. \begin{array}{l} \text{\{ Type of the session key that is gener-} \\ \text{\} ated by a successful key exchange.} \end{array} \right\}$

Description of $M_{\text{initiator}, \text{responder}}$:

Implemented role(s): initiator, responder
Subroutines: $\mathcal{F}_{\text{crypto}}$
Internal state:
<ul style="list-style-type: none"> - caller : $(\{0, 1\}^*)^3 \rightarrow (\{0, 1\}^*)^3 \cup \perp$. <i>\{ Stores entity_{call} when a user starts a key exchange. Initially \perp for all entries.</i> - userStatus : $(\{0, 1\}^*)^3 \times \mathbb{N} \rightarrow \{\perp, \text{started}, \text{inSession}, \text{exchangeFinished}, \text{sessionClosed}, \text{corrupted}\}$. <i>\{ Current status of users in the key exchange. Initially \perp for all entries.</i> - intendedPartner : $(\{0, 1\}^*)^3 \rightarrow \{0, 1\}^* \cup \{\perp\}$. <i>\{ Mapping from users to the PIDs of their intended partners. Initially \perp for all entries.</i> - sessions $\subseteq (\{0, 1\}^*)^3 \times (\{0, 1\}^*)^3 = \emptyset$. <i>\{ Set of key exchange sessions, where a session consists of a pair of users.</i> - sessionKeyPointer : $(\{0, 1\}^*)^3 \rightarrow \mathbb{N} \cup \{\perp\}$. <i>\{ Mapping from users to the pointers for their session keys. Initially \perp for all entries.</i> <p>In addition to the above, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ also keeps track of past operations, which we keep implicit for better readability.</p>
CheckID ($pid, sid, role$): Accept all entities.
Corruption behavior:
<ul style="list-style-type: none"> - AllowCorruption($pid, sid, role$): Set $allowed \leftarrow \text{true}$ iff $\text{userStatus}[pid, sid, role] \neq \text{exchangeFinished}$; otherwise $allowed \leftarrow \text{false}$. If $allowed = \text{true}$, then $\text{userStatus}[pid, sid, role] \leftarrow \text{corrupted}$. Output $allowed$. - LeakedData($pid, sid, role$): If called while ($pid, sid, role$) determines its initial corruption status, use the default behavior of LeakedData. That is, output the initially received message and the sender of that message. Else if ($pid, sid, role$) has already closed his key exchange session, output the value of his last pointer to a symmetric key in $\mathcal{F}_{\text{crypto}}$.^a Otherwise output \perp. - AllowAdvMessage($pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{\text{receiver}}, m$): Check that $pid = pid_{\text{receiver}}$ and that $role_{\text{receiver}} \neq \mathcal{F}_{\text{crypto}} : \text{crypto}$. If all checks succeed, output true, otherwise output false.
Main:
<p>recv (InitKE, pid', m') from I/O s.t. $\text{userStatus}[\text{entity}_{\text{cur}}] = \perp$: <i>\{ I/O: Start key exchange.</i> $\text{userStatus}[\text{entity}_{\text{cur}}] \leftarrow \text{started}$. $\text{intendedPartner}[\text{entity}_{\text{cur}}] \leftarrow pid'$. $\text{caller}[\text{entity}_{\text{cur}}] \leftarrow \text{entity}_{\text{call}}$ send (InitKE, pid', m') to NET.</p>
Continue with Figure G.2.
^a Since pointer values in $\mathcal{F}_{\text{crypto}}$ only ever increase, once an attacker corrupts a party in a realization, he would learn the value of the next pointer. This is captured in $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ by leaking this information. Note that this is purely modeling related information that is not supposed to be secret anyway; in fact, $\mathcal{F}_{\text{crypto}}$ already partially leaks this information whenever it requests the attacker to provide a new key for a user. Also note that this does not clash with perfect forward secrecy: while the attacker learns the value for the next pointer, he is not able to access and use any keys from the key exchange session.

Figure G.1.: The ideal functionality for key exchange with mutual authentication $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ (Part I).

Main (continued):

recv m **from** I/O **s.t.** $\text{userStatus}[\text{entity}_{\text{cur}}] = \text{exchangeFinished}$ and m is a valid New, Equal?, Enc, Dec, MAC, MACVerify, or Derive request for $\mathcal{F}_{\text{crypto}}$:^a
{I/O: Use keys after a successful key exchange.}

send m **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** m' . *{Note that responses from $\mathcal{F}_{\text{crypto}}$ are immediate due to the use of responsive environments.}*
reply m' .

recv CloseSession **from** I/O **s.t.** $\text{userStatus}[\text{entity}_{\text{cur}}] = \text{exchangeFinished}$: *{I/O: Close session.}*
 $\text{userStatus}[\text{entity}_{\text{cur}}] \leftarrow \text{sessionClosed}$.
send responsively CloseSession **to** NET;
wait for ..
reply (CloseSession, ok).

recv (GroupSession, $(\text{pid}_I, \text{sid}_I, \text{initiator}), (\text{pid}_R, \text{sid}_R, \text{responder})$) **from** NET:
{NET: Group two users into a key exchange session.}^b

Check that $\text{userStatus}[\text{pid}_I, \text{sid}_I, \text{initiator}] \in \{\text{started}, \text{corrupted}\}$,
 $\text{userStatus}[\text{pid}_R, \text{sid}_R, \text{responder}] \in \{\text{started}, \text{corrupted}\}$,
and neither $(\text{pid}_I, \text{sid}_I, \text{initiator})$ nor $(\text{pid}_R, \text{sid}_R, \text{responder})$ are part of a session.^c

if the above check succeeds:
Add $((\text{pid}_I, \text{sid}_I, \text{initiator}), (\text{pid}_R, \text{sid}_R, \text{responder}))$ to sessions
If $\text{userStatus}[\text{pid}_I, \text{sid}_I, \text{initiator}] = \text{started}$,
then $\text{userStatus}[\text{pid}_I, \text{sid}_I, \text{initiator}] \leftarrow \text{inSession}$.
If $\text{userStatus}[\text{pid}_R, \text{sid}_R, \text{responder}] = \text{started}$,
then $\text{userStatus}[\text{pid}_R, \text{sid}_R, \text{responder}] \leftarrow \text{inSession}$.
Choose a fresh bit string name that has not been used to create a PSK in $\mathcal{F}_{\text{crypto}}$ yet.
send (GetPSK, $t_{\text{key}}, \text{name}$) **from** $(\text{pid}_I, \text{sid}_I, \text{initiator})$ **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
wait for (GetPSK, ptr_I).
send (IsSymKeyCorrupt?, ptr_I) **from** $(\text{pid}_I, \text{sid}_I, \text{initiator})$ **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
wait for (IsSymKeyCorrupt?, b).
if $b = \text{true}$:
Go back to the step where name was chosen and rerun the whole PSK generation with a fresh (and different) name .
send (GetPSK, $t_{\text{key}}, \text{name}$) **from** $(\text{pid}_R, \text{sid}_R, \text{responder})$ **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
wait for (GetPSK, ptr_R).
 $\text{sessionKeyPointer}[\text{pid}_I, \text{sid}_I, \text{initiator}] \leftarrow \text{ptr}_I$.
 $\text{sessionKeyPointer}[\text{pid}_R, \text{sid}_R, \text{responder}] \leftarrow \text{ptr}_R$.
reply (GroupSession, success)

recv FinishKE **from** NET **s.t.** $\text{userStatus}[\text{entity}_{\text{cur}}] = \text{inSession}$:
{NET: Finish a key exchange for an (honest) user.}

Check that user $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}})$ is in a session with the intended partner $\text{pid}' := \text{intendedPartner}[\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}}]$, i.e., there exists $(\text{pid}', \text{sid}', \text{role}')$ such that one of the tuples $((\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}}), (\text{pid}', \text{sid}', \text{role}'))$ or $((\text{pid}', \text{sid}', \text{role}'), (\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}}))$ is in sessions.

if the above check succeeds:
 $\text{userStatus}[\text{entity}_{\text{cur}}] \leftarrow \text{exchangeFinished}$.
send (FinishKE, $\text{sessionKeyPointer}[\text{entity}_{\text{cur}}]$) **to** $\text{caller}[\text{entity}_{\text{cur}}]$.

^aBy valid we mean a request that is accepted and processed by $\mathcal{F}_{\text{crypto}}$ and that will result in some output message. In particular, if a pointer to a symmetric key is to be used by the request, then the pointer has previously been generated by that user and therefore points to a key. Note that this can easily be checked by $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ by keeping track of past operations.

^bThis request is part of modeling local session IDs: Key exchange sessions are not defined by sharing the same (global) SID. Instead, the adversary gets to choose which users form a session, even if they have different (locally chosen) SIDs, subject to certain restrictions that guarantee agreement on corresponding roles and unique sessions for honest users.

^cFormally, an entity entity is part of a session if $\exists \text{entity}'$ such that $(\text{entity}, \text{entity}') \in \text{sessions}$ or $(\text{entity}', \text{entity}) \in \text{sessions}$.

Figure G.2.: The ideal functionality for key exchange with mutual authentication $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ (Part II).

Description of the protocol $\mathcal{F}_{\text{key-use}}^{\text{UA}} = (\text{initiator}, \text{responder})$:

<p>Participating roles: $\{\text{initiator}, \text{responder}\}$ Corruption model: dynamic corruption with secure erasures Protocol parameters:</p> <ul style="list-style-type: none"> $\text{t}_{\text{key}} \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}.$ 	<p>$\left. \begin{array}{l} \text{\{ Type of the session key that is gener-} \\ \text{\} ated by a successful key exchange.} \end{array} \right\}$</p>
--	---

Description of $M_{\text{initiator}, \text{responder}}$:

<p>Implemented role(s): $\text{initiator}, \text{responder}$ Subroutines: $\mathcal{F}_{\text{crypto}}$ Internal state:</p> <ul style="list-style-type: none"> $\text{caller} : (\{0, 1\}^*)^3 \rightarrow (\{0, 1\}^*)^3 \cup \perp.$ <i>\{ Stores entity_{call} when a user starts a key exchange. Initially \perp for all entries. \}</i> $\text{userStatus} : (\{0, 1\}^*)^3 \times \mathbb{N} \rightarrow \{\perp, \text{started}, \text{inSession}, \text{exchangeFinished}, \text{sessionClosed}, \text{corrupted}\}.$ <i>\{ Current status of users in the key exchange. Initially \perp for all entries. \}</i> $\text{intendedPartner} : (\{0, 1\}^*)^3 \rightarrow \{0, 1\}^* \cup \{\perp\}.$ <i>\{ Mapping from users to the PIDs of their intended partners. Initially \perp for all entries. \}</i> $\text{sessions} \subseteq (\{0, 1\}^*)^3 \times (\{0, 1\}^*)^3 = \emptyset.$ <i>\{ Set of key exchange sessions, where a session consists of a pair of users. \}</i> $\text{sessionKeyPointer} : (\{0, 1\}^*)^3 \rightarrow \mathbb{N} \cup \{\perp\}.$ <i>\{ Mapping from users to the pointers for their session keys. Initially \perp for all entries. \}</i> <p>In addition to the above, $\mathcal{F}_{\text{key-use}}^{\text{MA}}$ also keeps track of past operations, which we keep implicit for better readability.</p> <p>CheckID($pid, sid, role$): Accept all entities.</p> <p>Corruption behavior:</p> <ul style="list-style-type: none"> AllowCorruption($pid, sid, role$): Set $\text{allowed} \leftarrow \text{true}$ iff $\text{userStatus}[pid, sid, role] \neq \text{exchangeFinished}$; otherwise $\text{allowed} \leftarrow \text{false}$. If $\text{allowed} = \text{true}$, then $\text{userStatus}[pid, sid, role] \leftarrow \text{corrupted}$. Output allowed. LeakedData($pid, sid, role$): If called while $(pid, sid, role)$ determines its initial corruption status, use the default behavior of LeakedData. That is, output the initially received message and the sender of that message. Else if $(pid, sid, role)$ has already closed his key exchange session, output the value of his last pointer to a symmetric key in $\mathcal{F}_{\text{crypto}}$. Otherwise, output \perp. AllowAdvMessage($pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{\text{receiver}}, m$): Check that $pid = pid_{\text{receiver}}$ and that $role_{\text{receiver}} \neq \mathcal{F}_{\text{crypto}} : \text{crypto}$. If all checks succeed, output true, otherwise output false. <p>Main:</p> <p>recv (InitKE, pid', m') from I/O to $(-, -, \text{initiator})$ s.t. $\text{userStatus}[\text{entity}_{\text{cur}}] = \perp$: <i>\{ I/O: Initiator start key exchange. \}</i></p> <p>$\text{userStatus}[\text{entity}_{\text{cur}}] \leftarrow \text{started}.$ $\text{intendedPartner}[\text{entity}_{\text{cur}}] \leftarrow pid'.$ $\text{caller}[\text{entity}_{\text{cur}}] \leftarrow \text{entity}_{\text{call}}$ send (InitKE, pid', m') to NET.</p> <p>recv (InitKE, m') from I/O to $(-, -, \text{responder})$ s.t. $\text{userStatus}[\text{entity}_{\text{cur}}] = \perp$: <i>\{ I/O: Responder start key exchange. \}</i></p> <p>$\text{userStatus}[\text{entity}_{\text{cur}}] \leftarrow \text{started}.$ $\text{caller}[\text{entity}_{\text{cur}}] \leftarrow \text{entity}_{\text{call}}$ send (InitKE, m') to NET.</p> <p>Continue with Figure G.4.</p>

Figure G.3.: The ideal functionality for key exchange with mutual authentication $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ (Part I).

Main (continued):

recv m **from** I/O **s.t.** $\text{userStatus}[\text{entity}_{\text{cur}}] = \text{exchangeFinished}$ and m is a valid New, Equal?, Enc, Dec, MAC, MACVerify, or Derive request for $\mathcal{F}_{\text{crypto}}$:^a

{I/O: Use keys after a successful key exchange.}

send m **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** m' . *{Note that responses from $\mathcal{F}_{\text{crypto}}$ are immediate due to the use of responsive environments.}*

reply m' .

recv CloseSession **from** I/O **s.t.** $\text{userStatus}[\text{entity}_{\text{cur}}] = \text{exchangeFinished}$: *{I/O: Close session.}*

$\text{userStatus}[\text{entity}_{\text{cur}}] \leftarrow \text{sessionClosed}$.

send responsively CloseSession **to** NET;

wait for ..

reply (CloseSession, ok).

recv (GroupSession, $(\text{pid}_I, \text{sid}_I, \text{initiator})$, $(\text{pid}_R, \text{sid}_R, \text{responder})$) **from** NET:

{NET: Group two users into a key exchange session.}

Check that $\text{userStatus}[\text{pid}_I, \text{sid}_I, \text{initiator}] \in \{\text{started}, \text{corrupted}\}$,

$\text{userStatus}[\text{pid}_R, \text{sid}_R, \text{responder}] \neq \perp$,

and neither $(\text{pid}_I, \text{sid}_I, \text{initiator})$ nor $(\text{pid}_R, \text{sid}_R, \text{responder})$ are part of a session.

Furthermore, if $\text{userStatus}[\text{pid}_R, \text{sid}_R, \text{responder}] \in \{\text{exchangeFinished}, \text{sessionClosed}\}$, then check that the pointer $\text{sessionKeyPointer}[\text{pid}_R, \text{sid}_R, \text{responder}]$ to his session key is uncorrupted in $\mathcal{F}_{\text{crypto}}$ by sending a IsSymKeyCorrupt? request.

if all of the above checks succeed:

Add $((\text{pid}_I, \text{sid}_I, \text{initiator}), (\text{pid}_R, \text{sid}_R, \text{responder}))$ to sessions

If $\text{userStatus}[\text{pid}_I, \text{sid}_I, \text{initiator}] = \text{started}$,

then $\text{userStatus}[\text{pid}_I, \text{sid}_I, \text{initiator}] \leftarrow \text{inSession}$.

If $\text{userStatus}[\text{pid}_R, \text{sid}_R, \text{responder}] = \text{started}$,

then $\text{userStatus}[\text{pid}_R, \text{sid}_R, \text{responder}] \leftarrow \text{inSession}$.

if $\text{userStatus}[\text{pid}_R, \text{sid}_R, \text{responder}] \in \{\text{exchangeFinished}, \text{sessionClosed}\}$:

Let name be the string that was used for the PSK to generate the session key pointer $\text{sessionKeyPointer}[\text{pid}_R, \text{sid}_R, \text{responder}]$ of the responder.

send (GetPSK, $\text{t}_{\text{key}}, \text{name}$) **from** $(\text{pid}_I, \text{sid}_I, \text{initiator})$ **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;

wait for (GetPSK, ptr_I).

$\text{sessionKeyPointer}[\text{pid}_I, \text{sid}_I, \text{initiator}] \leftarrow \text{ptr}_I$.

else:

Choose a fresh bit string name that has not been used to create a PSK in $\mathcal{F}_{\text{crypto}}$ yet.

send (GetPSK, $\text{t}_{\text{key}}, \text{name}$) **from** $(\text{pid}_I, \text{sid}_I, \text{initiator})$ **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;

wait for (GetPSK, ptr_I).

send (IsSymKeyCorrupt?, ptr_I) **from** $(\text{pid}_I, \text{sid}_I, \text{initiator})$ **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;

wait for (IsSymKeyCorrupt?, b).

if $b = \text{true}$:

Go back to the step where name was chosen and rerun the whole PSK generation with a fresh (and different) name .

send (GetPSK, $\text{t}_{\text{key}}, \text{name}$) **from** $(\text{pid}_R, \text{sid}_R, \text{responder})$ **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;

wait for (GetPSK, ptr_R).

$\text{sessionKeyPointer}[\text{pid}_I, \text{sid}_I, \text{initiator}] \leftarrow \text{ptr}_I$.

$\text{sessionKeyPointer}[\text{pid}_R, \text{sid}_R, \text{responder}] \leftarrow \text{ptr}_R$.

reply (GroupSession, success)

Continue with Figure G.5.

^acf. Figure G.2 for the definition of “valid”.Figure G.4.: The ideal functionality for key exchange with mutual authentication $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ (Part II).

Description of $M_{\text{initiator, responder}}$ (continued):

Main (continued):

recv FinishKE from NET to $(-, -, \text{initiator})$ **s.t.** $\text{userStatus}[\text{entity}_{\text{cur}}] = \text{inSession}$:
{NET: Finish a key exchange for an (honest) initiator.}

Check that user $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}})$ is in a session with the intended partner $\text{pid}' := \text{intendedPartner}[\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}}]$, i.e., there exists $(\text{pid}', \text{sid}', \text{role}')$ such that one of the tuples $((\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}}), (\text{pid}', \text{sid}', \text{role}'))$ or $((\text{pid}', \text{sid}', \text{role}'), (\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}}))$ is in sessions.

if the above check succeeds:
 $\text{userStatus}[\text{entity}_{\text{cur}}] \leftarrow \text{exchangeFinished}$.
 send $(\text{FinishKE}, \text{sessionKeyPointer}[\text{entity}_{\text{cur}}])$ **to** $\text{caller}[\text{entity}_{\text{cur}}]$.

recv FinishKE from NET to $(-, -, \text{responder})$ **s.t.** $\text{userStatus}[\text{entity}_{\text{cur}}] \in \{\text{started}, \text{inSession}\}$:
{NET: Finish a key exchange for an (honest) responder.}

if $\text{userStatus}[\text{entity}_{\text{cur}}] = \text{started}$:
 Choose a fresh bit string name that has not been used to generate a PSK yet.
 send $(\text{GetPSK}, \text{t}_{\text{key}}, \text{name})$ **from** $\text{entity}_{\text{cur}}$ **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
 wait for $(\text{GetPSK}, \text{ptr})$.
 $\text{sessionKeyPointer}[\text{entity}_{\text{cur}}] \leftarrow \text{ptr}$.
 $\text{userStatus}[\text{entity}_{\text{cur}}] \leftarrow \text{exchangeFinished}$.
 send $(\text{FinishKE}, \text{sessionKeyPointer}[\text{entity}_{\text{cur}}])$ **to** $\text{caller}[\text{entity}_{\text{cur}}]$.

Figure G.5.: The ideal functionality for key exchange with mutual authentication $\mathcal{F}_{\text{key-use}}^{\text{UA}}$ (Part III).

H. Formal Specifications of our Case Study Key Exchange Protocols

Here we give the formal specifications of the protocols \mathcal{P}_{ISO} , $\mathcal{P}_{\text{SIGMA}}$, and $\mathcal{P}_{\text{OPTLS}}$ using our template from the iUC framework. These specifications have already been explained in Section 5.3 and mainly serve for reference purposes.

Description of the protocol $\mathcal{P}_{\text{ISO}} = (\text{initiator}, \text{responder})$:

Participating roles: initiator, responder
Corruption model: dynamic corruption with secure erasures
Protocol parameters:
<ul style="list-style-type: none"> - $t_{\text{key}} \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}.$ { Type of the session key that is generated by a successful key exchange. }

Description of $M_{\text{initiator}}$:

Implemented role(s): initiator
Subroutines: $\mathcal{F}_{\text{crypto}}$
Internal state:
<ul style="list-style-type: none"> - $\text{caller} \in (\{0, 1\}^*)^3 \cup \{\perp\} = \perp.$ { Stores entity_{call} when the key exchange starts. } - $\text{userStatus} \in \{\perp, \text{started}, \text{exchangeFinished}, \text{sessionClosed}\} = \perp.$ { Current status in the key exchange. } - $\text{intendedPartner} \in \{0, 1\}^* \cup \{\perp\} = \perp.$ { PID of the intended partner. } - $(G, n, g) \in \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^* = (\epsilon, 0, \epsilon).$ { DH group. } - $\text{exponentPointer} \in \mathbb{N} \cup \{\perp\} = \perp.$ { Pointer to the secret exponent. } - $\text{gx} \in G \cup \{\perp\} = \perp.$ { Key share g^x of initiator. } - $\text{sessionKeyPointer} \in \mathbb{N} \cup \{\perp\} = \perp.$ { Pointer to the session key. } - $\text{initialCorrStatus} \in \{\text{true}, \text{false}\} = \text{false}.$ { Corruption status at the start of the key exchange. } - $\text{blocked} \in \{\text{true}, \text{false}\} = \text{false}.$ { Is this instance blocked due to misbehavior of the adversary? }

In addition to the above, $M_{\text{initiator}}$ also keeps track how many pointers have already been generated by the user once the key exchange has been finished.

Corruption behavior:

- **DetermineCorrStatus**($pid, sid, role$):
 - if** $\text{userStatus} \in \{\perp, \text{started}, \text{waitingForLastMessage}\}$:^a
 - Let corr be the corruption status of the signature key of party pid in $\mathcal{F}_{\text{crypto}}$.^b
 - If $\text{intendedPartner} \neq \perp$, let $\text{corr}_{\text{partner}}$ be the corruption status of the signature key of party intendedPartner in $\mathcal{F}_{\text{crypto}}$; otherwise let $\text{corr}_{\text{partner}} = \text{false}$.
 - Return $\text{corr} \vee \text{corr}_{\text{partner}}$.
 - else:**
 - Return initialCorrStatus .
- **AllowCorruption**($pid, sid, role$):
 - if** $\text{userStatus} \in \{\perp, \text{sessionClosed}\}$:
 - Let corr be the corruption status of the signature key of party pid in $\mathcal{F}_{\text{crypto}}$. Return corr .
 - else:**
 - Return false .
- **LeakedData**($pid, sid, role$): If $(pid, sid, role)$ is corrupted while determining its initial corruption status, then use the default behavior of **LeakedData**. Otherwise, return \perp .^c
- **AllowAdvMessage**($pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{\text{receiver}}, m$):
 - Check that $pid = pid_{\text{receiver}}$.
 - If $role_{\text{receiver}} = \mathcal{F}_{\text{crypto}}$, then also check that the adversary does not issue a request of the form **New**, **GenExp**, **GetPSK**, **PKEnc**, **PKDec**, **NewNonce** and that the adversary does not use a pointer (e.g., as key for a primitive or as part of a plaintext) that was created before corruption occurred.
 - If all checks succeed, output true , otherwise output false .

Initialization:

send **GetDHGroup** to $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** $(\text{GetDHGroup}, (G, n, g))$.
 $(G, n, g) \leftarrow (G, n, g)$.

Continue with Figure H.2.

^a**waitingForLastMessage** is a status that only exists for responders. We include it here so we can reuse the same algorithm for responders.

^bOn a technical level, this status is obtained by sending a **IsSigKeyCorrupt?** request to $\mathcal{F}_{\text{crypto}}$ and waiting for the response. Note that the response is always immediate due to responsive environments.

^cThis models that, when a key exchange session is closed, all values from the key exchange are erased securely.

Figure H.1.: The ISO protocol for mutually authenticated key exchange \mathcal{P}_{ISO} (Part I).

MessagePreprocessing:

If blocked = true then abort.

Main:

```

recv (InitKE,  $pid'$ ,  $m'$ ) from I/O s.t.  $userStatus = \perp$ :           { Initiator: Start key exchange
                                                                    { and send first message.
   $userStatus \leftarrow started$ .
   $intendedPartner \leftarrow pid'$ .
   $caller \leftarrow entity_{call}$ .
   $initialCorrStatus \leftarrow DetermineCorrStatus(entity_{cur})$ .
  send GenExp to  $(\epsilon, \epsilon, \mathcal{F}_{crypto} : crypto)$ ; wait for (GenExp,  $ptr$ ,  $gx$ ).
   $exponentPointer \leftarrow ptr$ ,  $gx \leftarrow gx$ .
  if  $initialCorrStatus = true$ :a
    send (RetrieveExp,  $exponentPointer$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{crypto} : crypto)$ ; wait for ..
    send ( $pid_{cur}$ ,  $gx$ ) to NET.
  recv ( $intendedPartner$ ,  $gy$ ,  $\sigma$ ) from NET s.t.  $userStatus = started$ ,  $gy \in G$ :   { Initiator: Receive
                                                                    { second message,
                                                                    { send third message,
                                                                    { and finish KE.

     $corr \leftarrow DetermineCorrStatus(entity_{cur})$ .
    if  $corr \neq initialCorrStatus$ :           { Check that the attacker stayed within corruption model.
       $blocked \leftarrow true$ .
      abort.
    send (BlockDhShare,  $gy$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{crypto} : crypto)$ ; wait for ..
     $m \leftarrow (gx, gy, pid_{cur})$ .
    send (GetPubKeySig,  $intendedPartner$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{crypto} : crypto)$ ; wait for (GetPubKeySig,  $pk$ ).
    send (SigVerify,  $intendedPartner$ ,  $pk$ ,  $m$ ,  $\sigma$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{crypto} : crypto)$ ; wait for (SigVerify,  $b$ ).
    if  $b = true$ :
      send (GenDHKey,  $exponentPointer$ ,  $gy$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{crypto} : crypto)$ ; wait for (GenDHKey,  $ptr$ ).
      send (Derive,  $ptr$ ,  $t_{key}$ ,  $\epsilon$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{crypto} : crypto)$ ; wait for (Derive,  $ptr'$ ).
       $sessionKeyPointer \leftarrow ptr'$ .
       $m' \leftarrow (gy, gx, intendedPartner)$ .
      send (Sign,  $m'$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{crypto} : crypto)$ ; wait for (Sign,  $\sigma'$ ).
      send responsively  $\sigma'$  to NET;           { Leak third message to the network without losing control.
      wait for ..
       $userStatus \leftarrow exchangeFinished$ .
      send (FinishKE, 0) to caller.           { The pointer 0 for the higher-level protocol will be mapped
                                                                    { to sessionKeyPointer internally by  $\mathcal{P}_{ISO}$ ; see below.
    else:
       $blocked \leftarrow true$            { Verification failed; protocol is aborted.

  recv  $m$  from I/O s.t.  $userStatus = exchangeFinished$  and  $m$  is a valid New, Equal?, Enc, Dec,
  MAC, MACVerify, or Derive request for  $\mathcal{F}_{crypto}$ :b
                                                                    { Initiator: Use keys after a successful key exchange.
  Replace pointers in  $m$  with internal pointers for  $\mathcal{F}_{crypto}$ , i.e., pointer 0 is changed to
  sessionKeyPointer, pointer 1 is changed to sessionKeyPointer + 1, etc.
  send  $m$  to  $(\epsilon, \epsilon, \mathcal{F}_{crypto} : crypto)$ ; wait for  $m'$ .
  Replace pointers in  $m'$  with external pointers for the higher-level protocol, i.e., pointer
  sessionKeyPointer is changed to 0, pointer sessionKeyPointer + 1 is changed to 1, etc.
  reply  $m'$ .

  recv CloseSession from I/O s.t.  $userStatus = exchangeFinished$ :           { Initiator: Close session.
     $userStatus \leftarrow sessionClosed$ .
    reply (CloseSession, ok).

```

^aImplicitly corrupted users do not expect any security guarantees and hence mark their exponents as known. See also the explanation in Section 5.3.1.

^bAnalogous to Figure G.2, by valid we mean that after the pointers in the message have been replaced (see first step of this “receive”-block), the resulting message will be accepted and processed by \mathcal{F}_{crypto} and lead to a response. In particular, if a pointer to a key is used then that pointer has already been generated before.

Figure H.2.: The ISO protocol for mutually authenticated key exchange \mathcal{P}_{ISO} (Part II).

Implemented role(s): responder

Subroutines: $\mathcal{F}_{\text{crypto}}$

Internal state:

- $\text{caller} \in (\{0, 1\}^*)^3 \cup \{\perp\} = \perp$. {Stores $\text{entity}_{\text{call}}$ when the key exchange starts.}
- $\text{userStatus} \in \{\perp, \text{started}, \text{waitingForLastMessage}, \text{exchangeFinished}, \text{sessionClosed}\} = \perp$. {Current status in the key exchange.}
- $\text{intendedPartner} \in \{0, 1\}^* \cup \{\perp\} = \perp$. {PID of the intended partner.}
- $(G, n, g) \in \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^* = (\epsilon, 0, \epsilon)$. {DH group.}
- $\text{exponentPointer} \in \mathbb{N} \cup \{\perp\} = \perp$. {Pointer to the secret exponent.}
- $\text{gx} \in G \cup \{\perp\} = \perp$. {Key share g^x of initiator.}
- $\text{gy} \in G \cup \{\perp\} = \perp$. {Key share g^y of responder.}
- $\text{sessionKeyPointer} \in \mathbb{N} \cup \{\perp\} = \perp$. {Pointer to the session key.}
- $\text{initialCorrStatus} \in \{\text{true}, \text{false}\} = \text{false}$. {Corruption status at the start of the key exchange.}
- $\text{blocked} \in \{\text{true}, \text{false}\} = \text{false}$. {Is this instance blocked due to misbehavior of the adversary?}

In addition to the above, $M_{\text{responder}}$ also keeps track how many pointers have already been generated by the user once the key exchange has been finished.

Corruption behavior, Initialization, MessagePreprocessing:

Identical to $M_{\text{initiator}}$, cf. Figure H.1.

Main:

```

recv (InitKE,  $\text{pid}'$ ,  $m'$ ) from I/O s.t.  $\text{userStatus} = \perp$ : {Responder: Start key exchange.}
   $\text{userStatus} \leftarrow \text{started}$ .
   $\text{intendedPartner} \leftarrow \text{pid}'$ .
   $\text{caller} \leftarrow \text{entity}_{\text{call}}$ .
   $\text{initialCorrStatus} \leftarrow \text{DetermineCorrStatus}(\text{entity}_{\text{cur}})$ . {No message is sent as the responder waits for the first KE message.}
{Control reverts to the environment.}

recv ( $\text{intendedPartner}$ ,  $gx$ ) from NET s.t.  $\text{userStatus} = \text{started}$ ,  $gx \in G$ : {Responder: Receive first message, send second message.}
  send (BlockDHShare,  $gx$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for ..
   $gx \leftarrow gx$ .
  send GenExp to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (GenExp,  $\text{ptr}$ ,  $gy$ ).
   $\text{exponentPointer} \leftarrow \text{ptr}$ ,  $gy \leftarrow gy$ .
   $m \leftarrow (gx, gy, \text{intendedPartner})$ .
  if  $\text{initialCorrStatus} = \text{true}$ :
    send (RetrieveExp,  $\text{exponentPointer}$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for ..
  send (Sign,  $m$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (Sign,  $\sigma$ ).
   $\text{userStatus} \leftarrow \text{waitingForLastMessage}$ .
  send ( $\text{pid}_{\text{cur}}$ ,  $gy$ ,  $\sigma$ ) to NET.

recv  $\sigma$  from NET s.t.  $\text{userStatus} = \text{waitingForLastMessage}$ : {Responder: Receive third message and finish KE.}
   $\text{corr} \leftarrow \text{DetermineCorrStatus}(\text{entity}_{\text{cur}})$ .
  if  $\text{corr} \neq \text{initialCorrStatus}$ : {Check that the attacker stayed within corruption model.}
     $\text{blocked} \leftarrow \text{true}$ .
    abort.
   $m \leftarrow (gy, gx, \text{pid}_{\text{cur}})$ .
  send (GetPubKeySig,  $\text{intendedPartner}$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (GetPubKeySig,  $pk$ ).
  send (SigVerify,  $\text{intendedPartner}$ ,  $pk$ ,  $m$ ,  $\sigma$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (SigVerify,  $b$ ).
  if  $b = \text{true}$ :
    send (GenDHKey,  $\text{exponentPointer}$ ,  $gx$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (GenDHKey,  $\text{ptr}$ ).
    send (Derive,  $\text{ptr}$ ,  $t_{\text{key}}$ ,  $\epsilon$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (Derive,  $\text{ptr}'$ ).
     $\text{sessionKeyPointer} \leftarrow \text{ptr}'$ .
     $\text{userStatus} \leftarrow \text{exchangeFinished}$ .
    send (FinishKE, 0) to caller. {The pointer 0 for the higher-level protocol will be mapped to sessionKeyPointer internally by  $\mathcal{P}_{\text{ISO}}$ ; see below.}
  else:
     $\text{blocked} \leftarrow \text{true}$  {Verification failed; protocol is aborted.}

```

The Main algorithm is continued in Figure H.4.

Figure H.3.: The ISO protocol for mutually authenticated key exchange \mathcal{P}_{ISO} (Part III).

Description of $M_{\text{responder}}$ (continued):

Main (continued):

recv m **from** I/O **s.t.** $\text{userStatus} = \text{exchangeFinished}$ and m is a valid **New**, **Equal?**, **Enc**, **Dec**,
MAC, **MACVerify**, or **Derive** request for $\mathcal{F}_{\text{crypto}}$:^a

{Responder: Use keys after a successful key exchange.}

Replace pointers in m with internal pointers for $\mathcal{F}_{\text{crypto}}$, i.e., pointer 0 is changed to sessionKeyPointer , pointer 1 is changed to $\text{sessionKeyPointer} + 1$, etc.

send m **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** m' .

Replace pointers in m' with external pointers for the higher-level protocol, i.e., pointer sessionKeyPointer is changed to 0, pointer $\text{sessionKeyPointer} + 1$ is changed to 1, etc.

reply m' .

recv **CloseSession** **from** I/O **s.t.** $\text{userStatus} = \text{exchangeFinished}$: *{Responder: Close session.}*

$\text{userStatus} \leftarrow \text{sessionClosed}$.

reply (**CloseSession**, ok).

^aCf. Figure H.2 for the definition of “valid”.

Figure H.4.: The ISO protocol for mutually authenticated key exchange \mathcal{P}_{ISO} (Part IV).

Description of the protocol $\mathcal{P}_{\text{SIGMA}} = (\text{initiator}, \text{responder})$:

<p>Participating roles: initiator, responder Corruption model: dynamic corruption with secure erasures Protocol parameters:</p> <ul style="list-style-type: none"> - $t_{\text{key}} \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}$. 	$\left. \begin{array}{l} \{ \text{Type of the session key that is gener-} \\ \{ \text{ated by a successful key exchange.} \end{array} \right\}$
--	---

Description of $M_{\text{initiator}}$:

<p>Implemented role(s): initiator Subroutines: $\mathcal{F}_{\text{crypto}}$ Internal state:</p> <ul style="list-style-type: none"> - $\text{caller} \in (\{0, 1\}^*)^3 \cup \{\perp\} = \perp$. {Stores $\text{entity}_{\text{call}}$ when the key exchange starts.} - $\text{userStatus} \in \{\perp, \text{started}, \text{exchangeFinished}, \text{sessionClosed}\} = \perp$. {Current status in the key exchange.} - $\text{intendedPartner} \in \{0, 1\}^* \cup \{\perp\} = \perp$. {PID of the intended partner.} - $(G, n, g) \in \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^* = (\epsilon, 0, \epsilon)$. {DH group.} - $\text{exponentPointer} \in \mathbb{N} \cup \{\perp\} = \perp$. {Pointer to the secret exponent.} - $\text{gx} \in G \cup \{\perp\} = \perp$. {Key share g^x of initiator.} - $\text{sessionKeyPointer} \in \mathbb{N} \cup \{\perp\} = \perp$. {Pointer to the session key.} - $\text{initialCorrStatus} \in \{\text{true}, \text{false}\} = \text{false}$. {Corruption status at the start of the key exchange.} - $\text{blocked} \in \{\text{true}, \text{false}\} = \text{false}$. {Is this instance blocked due to misbehavior of the adversary?} <p>In addition to the above, $M_{\text{initiator}}$ also keeps track how many pointers have already been generated by the user once the key exchange has been finished.</p> <p>Corruption behavior, Initialization, MessagePreprocessing: Identical to $M_{\text{initiator}}$ of \mathcal{P}_{ISO}, cf. Figure H.1.</p> <p>Main:</p> <p>recv (InitKE, pid', m') from I/O s.t. $\text{userStatus} = \perp$: {Initiator: Start key exchange and send first message.}</p> <p style="padding-left: 20px;"> $\text{userStatus} \leftarrow \text{started}$. $\text{intendedPartner} \leftarrow pid'$. $\text{caller} \leftarrow \text{entity}_{\text{call}}$. $\text{initialCorrStatus} \leftarrow \text{DetermineCorrStatus}(\text{entity}_{\text{cur}})$. send GenExp to $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; wait for (GenExp, ptr, gx). $\text{exponentPointer} \leftarrow ptr$, $\text{gx} \leftarrow gx$. send gx to NET. </p> <p>The Main algorithm is continued in Figure H.6.</p>

Figure H.5.: The SIGMA protocol for mutually authenticated key exchange $\mathcal{P}_{\text{SIGMA}}$ (Part I).

Main (continued):

recv (gy, c) **from** NET s.t. $\text{userStatus} = \text{started}$, $gy \in G$: { *Initiator: Receive second message, send third message, and finish KE.*
 $\text{corr} \leftarrow \text{DetermineCorrStatus}(\text{entity}_{\text{cur}})$. { *Check that attacker stayed within corruption model.*
if $\text{corr} \neq \text{initialCorrStatus}$:
 $\text{blocked} \leftarrow \text{true}$.
 abort.
send ($\text{BlockDHShare}, gy$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** ..
send ($\text{GenDHKey}, \text{exponentPointer}, gy$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** ($\text{GenDHKey}, ptr_{DH}$).
send ($\text{Derive}, ptr_{DH}, \text{unauthenc-key}, 0$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (Derive, ptr_{Enc}).
send ($\text{Derive}, ptr_{DH}, \text{mac-key}, 0$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (Derive, ptr_{MAC}).
send (Dec, ptr_{Enc}, c) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (Dec, m_{Enc}).
Try to parse m_{Enc} as $(\text{intendedPartner}, \sigma_{Sig}, \sigma_{MAC})$.
if parsing fails:
 $\text{blocked} \leftarrow \text{true}$.
 abort.
 $m_{Sig} \leftarrow (gx, gy)$, $m_{MAC} \leftarrow \text{intendedPartner}$.
send ($\text{GetPubKeySig}, \text{intendedPartner}$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** ($\text{GetPubKeySig}, pk$).
send ($\text{SigVerify}, \text{intendedPartner}, pk, m_{Sig}, \sigma_{Sig}$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** ($\text{SigVerify}, b$).
send ($\text{MACVerify}, m_{MAC}, \sigma_{MAC}$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** ($\text{MACVerify}, b'$).
if $b \neq \text{true} \vee b' \neq \text{true}$:
 $\text{blocked} \leftarrow \text{true}$.
 abort. { *Second message was valid. Generate session key and third message. Note that the seed is different from previously generated keys to guarantee a fresh session key.*
send ($\text{Derive}, ptr_{DH}, t_{\text{key}}, 1$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
wait for (Derive, ptr_{sk}).
 $\text{sessionKeyPointer} \leftarrow ptr_{sk}$.
 $m'_{Sig} \leftarrow (gy, gx)$, $m'_{MAC} \leftarrow \text{pid}_{\text{cur}}$.
send (Sign, m'_{Sig}) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** ($\text{Sign}, \sigma'_{Sig}$).
send ($\text{MAC}, ptr_{MAC}, m'_{MAC}$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** ($\text{MAC}, \sigma'_{MAC}$).
 $m'_{Enc} \leftarrow (\text{pid}_{\text{cur}}, \sigma'_{Sig}, \sigma'_{MAC})$.
send ($\text{Enc}, ptr_{Enc}, m'_{Enc}$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (Enc, c').
send responsively c' **to** NET; { *Leak third message to the network without losing control.*
wait for ..
 $\text{userStatus} \leftarrow \text{exchangeFinished}$.
send ($\text{FinishKE}, 0$) **to** caller. { *The pointer 0 for the higher-level protocol will be mapped to sessionKeyPointer internally by $\mathcal{P}_{\text{SIGMA}}$; see below.*

recv m **from** I/O s.t. $\text{userStatus} = \text{exchangeFinished}$ and m is a valid New, Equal?, Enc, Dec, MAC, MACVerify, or Derive request for $\mathcal{F}_{\text{crypto}}$.^a
{ *Initiator: Use keys after a successful key exchange.*
Replace pointers in m with internal pointers for $\mathcal{F}_{\text{crypto}}$, i.e., pointer 0 is changed to sessionKeyPointer , pointer 1 is changed to $\text{sessionKeyPointer} + 1$, etc.
send m **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** m' .
Replace pointers in m' with external pointers for the higher-level protocol, i.e., pointer sessionKeyPointer is changed to 0, pointer $\text{sessionKeyPointer} + 1$ is changed to 1, etc.
reply m' .

recv CloseSession **from** I/O s.t. $\text{userStatus} = \text{exchangeFinished}$: { *Initiator: Close session.*
 $\text{userStatus} \leftarrow \text{sessionClosed}$.
reply (CloseSession, ok).

^aCf. Figure H.2 for the definition of “valid”.Figure H.6.: The SIGMA protocol for mutually authenticated key exchange $\mathcal{P}_{\text{SIGMA}}$ (Part II).

Description of $M_{\text{responder}}$:

Implemented role(s): responder

Subroutines: $\mathcal{F}_{\text{crypto}}$

Internal state:

- $\text{caller} \in (\{0, 1\}^*)^3 \cup \{\perp\} = \perp$. {Stores $\text{entity}_{\text{call}}$ when the key exchange starts.}
- $\text{userStatus} \in \{\perp, \text{started}, \text{waitingForLastMessage}, \text{exchangeFinished}, \text{sessionClosed}\} = \perp$. {Current status in the key exchange.}
- $\text{intendedPartner} \in \{0, 1\}^* \cup \{\perp\} = \perp$. {PID of the intended partner.}
- $(G, n, g) \in \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^* = (\epsilon, 0, \epsilon)$. {DH group.}
- $\text{exponentPointer} \in \mathbb{N} \cup \{\perp\} = \perp$. {Pointer to the secret exponent.}
- $\text{gx} \in G \cup \{\perp\} = \perp$. {Key share g^x of initiator.}
- $\text{gy} \in G \cup \{\perp\} = \perp$. {Key share g^y of responder.}
- $\text{dhKeyPointer} \in \mathbb{N} \cup \{\perp\} = \perp$. {Pointer to the DH key.}
- $\text{encKeyPointer} \in \mathbb{N} \cup \{\perp\} = \perp$. {Pointer to the encryption key.}
- $\text{macKeyPointer} \in \mathbb{N} \cup \{\perp\} = \perp$. {Pointer to the MAC key.}
- $\text{sessionKeyPointer} \in \mathbb{N} \cup \{\perp\} = \perp$. {Pointer to the session key.}
- $\text{initialCorrStatus} \in \{\text{true}, \text{false}\} = \text{false}$. {Corruption status at the start of the key exchange.}
- $\text{blocked} \in \{\text{true}, \text{false}\} = \text{false}$. {Is this instance blocked due to misbehavior of the adversary?}

In addition to the above, $M_{\text{responder}}$ also keeps track how many pointers have already been generated by the user once the key exchange has been finished.

Corruption behavior, Initialization, MessagePreprocessing:

Identical to $M_{\text{initiator}}$ of \mathcal{P}_{ISO} , cf. Figure H.1.

Main:

```

recv (InitKE,  $\text{pid}'$ ,  $m'$ ) from I/O s.t.  $\text{userStatus} = \perp$ : {Responder: Start key exchange.}
   $\text{userStatus} \leftarrow \text{started}$ .
   $\text{intendedPartner} \leftarrow \text{pid}'$ .
   $\text{caller} \leftarrow \text{entity}_{\text{call}}$ .
   $\text{initialCorrStatus} \leftarrow \text{DetermineCorrStatus}(\text{entity}_{\text{cur}})$ . {No message is sent as the responder waits for the first KE message. Control reverts to the environment.}

recv  $gx$  from NET s.t.  $\text{userStatus} = \text{started}$ ,  $gx \in G$ : {Responder: Receive first message, send second message.}
  send (BlockDHShare,  $gx$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for ...
   $gx \leftarrow gx$ .
  send GenExp to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (GenExp,  $\text{ptr}$ ,  $gy$ ).
   $\text{exponentPointer} \leftarrow \text{ptr}$ ,  $gy \leftarrow gy$ .
  send (GenDHKey,  $\text{exponentPointer}$ ,  $gx$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (GenDHKey,  $\text{ptr}_{DH}$ ).
  send (Derive,  $\text{ptr}_{DH}$ , unauthenc-key, 0) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (Derive,  $\text{ptr}_{Enc}$ ).
  send (Derive,  $\text{ptr}_{DH}$ , mac-key, 0) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (Derive,  $\text{ptr}_{MAC}$ ).
   $\text{dhKeyPointer} \leftarrow \text{ptr}_{DH}$ ,  $\text{encKeyPointer} \leftarrow \text{ptr}_{Enc}$ ,  $\text{macKeyPointer} \leftarrow \text{ptr}_{MAC}$ .
   $m_{Sig} \leftarrow (gx, gy)$ ,  $m_{MAC} \leftarrow \text{pid}_{\text{cur}}$ .
  send (Sign,  $m_{Sig}$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (Sign,  $\sigma_{Sig}$ ).
  send (MAC,  $\text{macKeyPointer}$ ,  $m_{MAC}$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (MAC,  $\sigma_{MAC}$ ).
   $m_{Enc} \leftarrow (\text{pid}_{\text{cur}}, \sigma_{Sig}, \sigma_{MAC})$ .
  send (Enc,  $\text{encKeyPointer}$ ,  $m_{Enc}$ ) to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for (Enc,  $c$ ).
   $\text{userStatus} \leftarrow \text{waitingForLastMessage}$ .
  send ( $gy$ ,  $c$ ) to NET.

```

The Main algorithm is continued in Figure H.8.

Figure H.7.: The SIGMA protocol for mutually authenticated key exchange $\mathcal{P}_{\text{SIGMA}}$ (Part III).

Description of $M_{\text{responder}}$ (continued):

Main (continued):

```

recv  $c$  from NET s.t.  $\text{userStatus} = \text{waitingForLastMessage}$ : { Responder: Receive third
   $\text{corr} \leftarrow \text{DetermineCorrStatus}(\text{entity}_{\text{cur}})$ . { message and finish KE.
  if  $\text{corr} \neq \text{initialCorrStatus}$ : { Check that the attacker stayed within corruption model.
     $\text{blocked} \leftarrow \text{true}$ .
    abort.
  send  $(\text{Dec}, \text{encKeyPointer}, c)$  to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for  $(\text{Dec}, m_{\text{Enc}})$ .
  Try to parse  $m_{\text{Enc}}$  as  $(\text{intendedPartner}, \sigma_{\text{Sig}}, \sigma_{\text{MAC}})$ .
  if parsing fails:
     $\text{blocked} \leftarrow \text{true}$ .
    abort.
   $m_{\text{Sig}} \leftarrow (\text{gy}, \text{gx})$ ,  $m_{\text{MAC}} \leftarrow \text{intendedPartner}$ .
  send  $(\text{GetPubKeySig}, \text{intendedPartner})$  to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for  $(\text{GetPubKeySig}, pk)$ .
  send  $(\text{SigVerify}, \text{intendedPartner}, pk, m_{\text{Sig}}, \sigma_{\text{Sig}})$  to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for  $(\text{SigVerify}, b)$ .
  send  $(\text{MACVerify}, m_{\text{MAC}}, \sigma_{\text{MAC}})$  to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for  $(\text{MACVerify}, b')$ .
  if  $b \neq \text{true} \vee b' \neq \text{true}$ :
     $\text{blocked} \leftarrow \text{true}$ .
    abort.
  send  $(\text{Derive}, \text{dhKeyPointer}, t_{\text{key}}, 1)$  to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for  $(\text{Derive}, ptr_{sk})$ .
   $\text{sessionKeyPointer} \leftarrow ptr_{sk}$ .
   $\text{userStatus} \leftarrow \text{exchangeFinished}$ .
  send  $(\text{FinishKE}, 0)$  to caller. { The pointer 0 for the higher-level protocol will be mapped
{ to sessionKeyPointer internally by  $\mathcal{P}_{\text{SIGMA}}$ ; see below.

recv  $m$  from I/O s.t.  $\text{userStatus} = \text{exchangeFinished}$  and  $m$  is a valid New, Equal?, Enc, Dec,
  MAC, MACVerify, or Derive request for  $\mathcal{F}_{\text{crypto}}$ :a
{ Responder: Use keys after a successful key exchange.
  Replace pointers in  $m$  with internal pointers for  $\mathcal{F}_{\text{crypto}}$ , i.e., pointer 0 is changed to
   $\text{sessionKeyPointer}$ , pointer 1 is changed to  $\text{sessionKeyPointer} + 1$ , etc.
  send  $m$  to  $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$ ; wait for  $m'$ .
  Replace pointers in  $m'$  with external pointers for the higher-level protocol, i.e., pointer
   $\text{sessionKeyPointer}$  is changed to 0, pointer  $\text{sessionKeyPointer} + 1$  is changed to 1, etc.
  reply  $m'$ .

recv CloseSession from I/O s.t.  $\text{userStatus} = \text{exchangeFinished}$ : { Responder: Close session.
   $\text{userStatus} \leftarrow \text{sessionClosed}$ .
  reply  $(\text{CloseSession}, \text{ok})$ .

```

^aCf. Figure H.2 for the definition of “valid”.

Figure H.8.: The SIGMA protocol for mutually authenticated key exchange $\mathcal{P}_{\text{SIGMA}}$ (Part IV).

Description of the protocol $\mathcal{P}'_{\text{OPTLS}} = (\text{initiator}, \text{responder})$:

Participating roles: initiator, responder	
Corruption model: dynamic corruption with secure erasures	
Protocol parameters:	
- $t_{\text{key}} \in \{\text{pre-key}, \text{unauthenc-key}, \text{authenc-key}, \text{mac-key}\}$.	$\left\{ \begin{array}{l} \text{Type of the session key that is gener-} \\ \text{ated by a successful key exchange.} \end{array} \right.$

Description of $M_{\text{initiator}}$:

Implemented role(s): initiator	$\left\{ \begin{array}{l} \mathcal{F}_{\text{uncorruptDB}} \text{ (cf. Figure H.13) is a modeling tool for storing DH} \\ \text{shares of uncorrupted initiators. This is then used to determine} \\ \text{the corruption status of responders, cf. Section 5.3.2} \end{array} \right.$
Subroutines: $\mathcal{F}_{\text{crypto}}, \mathcal{F}_{\text{uncorruptDB}}$	
Internal state:	
- $\text{caller} \in (\{0, 1\}^*)^3 \cup \{\perp\} = \perp$.	$\left\{ \begin{array}{l} \text{Stores entity}_{\text{call}} \text{ when the key exchange starts.} \end{array} \right.$
- $\text{userStatus} \in \{\perp, \text{started}, \text{exchangeFinished}, \text{sessionClosed}\} = \perp$.	$\left\{ \begin{array}{l} \text{Current status in the key exchange.} \end{array} \right.$
- $\text{intendedPartner} \in \{0, 1\}^* \cup \{\perp\} = \perp$.	$\left\{ \begin{array}{l} \text{PID of the intended partner.} \end{array} \right.$
- $\text{chello} \in \{0, 1\}^* \cup \{\perp\} = \perp$.	$\left\{ \begin{array}{l} \text{Client hello.} \end{array} \right.$
- $(G, n, g) \in \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^* = (\epsilon, 0, \epsilon)$.	$\left\{ \begin{array}{l} \text{DH group.} \end{array} \right.$
- $\text{exponentPointer} \in \mathbb{N} \cup \{\perp\} = \perp$.	$\left\{ \begin{array}{l} \text{Pointer to the secret exponent.} \end{array} \right.$
- $\text{gx} \in G \cup \{\perp\} = \perp$.	$\left\{ \begin{array}{l} \text{Key share } g^x \text{ of initiator.} \end{array} \right.$
- $\text{sessionKeyPointer} \in \mathbb{N} \cup \{\perp\} = \perp$.	$\left\{ \begin{array}{l} \text{Pointer to the session key.} \end{array} \right.$
- $\text{initialCorrStatus} \in \{\text{true}, \text{false}\} = \text{false}$.	$\left\{ \begin{array}{l} \text{Corruption status at the start of the key exchange.} \end{array} \right.$
- $\text{blocked} \in \{\text{true}, \text{false}\} = \text{false}$.	$\left\{ \begin{array}{l} \text{Is this instance blocked due to misbehavior of the adversary?} \end{array} \right.$
In addition to the above, $M_{\text{initiator}}$ also keeps track how many pointers have already been generated by the user once the key exchange has been finished.	

Corruption behavior, Initialization, MessagePreprocessing:

Identical to $M_{\text{initiator}}$ of \mathcal{P}_{ISO} , cf. Figure H.1, except for **AllowAdvMessage** which is changed to the following algorithm:

AllowAdvMessage($pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{\text{receiver}}, m$):

Check that $pid = pid_{\text{receiver}}$ and $role_{\text{receiver}} \neq \mathcal{F}_{\text{uncorruptDB}}$: **uncorruptDB**.^a

If $role_{\text{receiver}} = \mathcal{F}_{\text{crypto}}$: **crypto**, then also check that the adversary does not issue a request of the form **New**, **GenExp**, **GetPSK**, **PKEnc**, **PKDec**, **NewNonce** and that the adversary does not use a pointer (e.g., as key for a primitive or as part of a plaintext) that was created before corruption occurred.

If all checks succeed, output **true**, otherwise output **false**.

Main:

recv (InitKE, pid', m') from I/O s.t. $\text{userStatus} = \perp$:	$\left\{ \begin{array}{l} \text{Initiator: Start key exchange} \\ \text{and send first message.} \end{array} \right.$
$\text{userStatus} \leftarrow \text{started}$.	
$\text{intendedPartner} \leftarrow pid'$.	
$\text{chello} \leftarrow m'$.	
$\text{caller} \leftarrow \text{entity}_{\text{call}}$.	
$\text{initialCorrStatus} \leftarrow \text{DetermineCorrStatus}(\text{entity}_{\text{cur}})$.	
send GenExp to $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; wait for (GenExp, ptr, gx).	
$\text{exponentPointer} \leftarrow ptr, gx \leftarrow gx$.	
if $\text{initialCorrStatus} = \text{false}$:	
send (AddShareOfUncorruptedUser, gx) to $(\epsilon, \epsilon, \mathcal{F}_{\text{uncorruptDB}} : \text{uncorruptDB})$; wait for ..	
else . ^b	
send (RetrieveExp, exponentPointer) to $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; wait for ..	
send (chello, gx) to NET.	

The Main algorithm is continued in Figure H.10.

^aOnly honest users may add DH shares to $\mathcal{F}_{\text{uncorruptDB}}$. Otherwise the attacker could break out of the intended corruption model.

^bImplicitly corrupted users do not expect any security guarantees and hence mark their exponents as known. See also the explanation in Section 5.3.1.

Figure H.9.: The modified OPTLS protocol for unilaterally authenticated key exchange $\mathcal{P}'_{\text{OPTLS}}$ (Part I).

Main (continued):

recv (*shello*, *gy*, *c*) **from** NET s.t. `userStatus = started`, $gy \in G$: *{Initiator: Receive second message and finish KE.}*
 $corr \leftarrow \text{DetermineCorrStatus}(\text{entity}_{\text{cur}})$.
if $corr \neq \text{initialCorrStatus}$: *{Check that attacker stayed within corruption model.}*
 `blocked` \leftarrow `true`.
 abort.
send (`BlockDHShare`, *gy*) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** `_`.
send (`GenDHKey`, `exponentPointer`, *gy*) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (`GenDHKey`, ptr_{DH}).
 $(ptr_{Enc}, ptr_{MAC}, \text{sessionKeyPointer}) \leftarrow \text{DeriveKeys}(ptr_{DH}, \text{chello}, \text{shello})$. *{see below for DeriveKeys.}*
send (`Dec`, ptr_{Enc} , *c*) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (`Dec`, m_{Enc}).
Try to parse m_{Enc} as $(\sigma_{Sig}, \sigma_{MAC})$.
if parsing fails:
 `blocked` \leftarrow `true`. *{This models that the user aborts his key exchange.}*
 abort.
 $m_{Sig} \leftarrow (gx, gy)$, $m_{MAC} \leftarrow (\text{chello}, gx, \text{shello}, gy, \sigma_{Sig})$.
send (`GetPubKeySig`, `intendedPartner`) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (`GetPubKeySig`, *pk*).
send (`SigVerify`, `intendedPartner`, *pk*, m_{Sig} , σ_{Sig}) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (`SigVerify`, *b*).
send (`MACVerify`, m_{MAC} , σ_{MAC}) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (`MACVerify`, *b'*).
if $b \neq \text{true} \vee b' \neq \text{true}$:
 `blocked` \leftarrow `true`.
 abort.
`userStatus` \leftarrow `exchangeFinished`.
send (`FinishKE`, 0) **to** caller. *{The pointer 0 for the higher-level protocol will be mapped to sessionKeyPointer internally by $\mathcal{P}'_{\text{OPTLS}}$; see below.}*

recv *m* **from** I/O s.t. `userStatus = exchangeFinished` and *m* is a valid `New`, `Equal?`, `Enc`, `Dec`, `MAC`, `MACVerify`, or `Derive` request for $\mathcal{F}_{\text{crypto}}$:^a
{Initiator: Use keys after a successful key exchange.}
Replace pointers in *m* with internal pointers for $\mathcal{F}_{\text{crypto}}$, i.e., pointer 0 is changed to `sessionKeyPointer`, pointer 1 is changed to `sessionKeyPointer + 1`, etc.
send *m* **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** *m'*.
Replace pointers in *m'* with external pointers for the higher-level protocol, i.e., pointer `sessionKeyPointer` is changed to 0, pointer `sessionKeyPointer + 1` is changed to 1, etc.
reply *m'*.

recv `CloseSession` **from** I/O s.t. `userStatus = exchangeFinished`: *{Initiator: Close session.}*
`userStatus` \leftarrow `sessionClosed`.
reply (`CloseSession`, `ok`).

Helperfunction:

function `DeriveKeys`(ptr_{DH} , *chello*, *shello*)
 send (`Derive`, ptr_{DH} , `unauthenc-key`, $0 || \text{shello} || 0$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
 wait for (`Derive`, ptr_{Enc}).
 send (`Derive`, ptr_{DH} , `pre-key`, $1 || \text{shello} || 0$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
 wait for (`Derive`, ptr_{derive}).
 send (`Derive`, ptr_{DH} , `pre-key`, $0 || \text{chello} || 0$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
 wait for (`Derive`, ptr_{seed}).
 send (`Derive`, ptr_{DH} , `mac-key`, $1 || \text{chello} || 0$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
 wait for (`Derive`, ptr_{MAC}).
 send (`Retrieve`, ptr_{seed}) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
 wait for (`Retrieve`, *seed*).
 send (`Derive`, ptr_{derive} , `tkey`, $\epsilon || \text{seed}$) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$;
 wait for (`Derive`, ptr_{sk}).
 return (ptr_{Enc} , ptr_{MAC} , ptr_{sk}).

^aCf. Figure H.2 for the definition of “valid”.Figure H.10.: The modified OPTLS protocol for unilaterally authenticated key exchange $\mathcal{P}'_{\text{OPTLS}}$ (Part II).

Description of $M_{\text{responder}}$:

Implemented role(s): responder

Subroutines: $\mathcal{F}_{\text{crypto}}$

Internal state:

- $\text{caller} \in (\{0, 1\}^*)^3 \cup \{\perp\} = \perp$. {Stores $\text{entity}_{\text{call}}$ when the key exchange starts.}
- $\text{userStatus} \in \{\perp, \text{started}, \text{exchangeFinished}, \text{sessionClosed}\} = \perp$. {Current status in the key exchange.}
- $\text{shello} \in \{0, 1\}^* \cup \{\perp\} = \perp$. {Server hello.}
- $(G, n, g) \in \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^* = (\epsilon, 0, \epsilon)$. {DH group.}
- $\text{sessionKeyPointer} \in \mathbb{N} \cup \{\perp\} = \perp$. {Pointer to the session key.}
- $\text{corrFlag} \in \{\text{true}, \text{false}\} = \text{false}$. {Consider user corrupted because DH share g^x was not generated by uncorrupted initiator; cf. Section 5.3.2.}
- $\text{CorrStatusAfterFinish} \in \{\text{true}, \text{false}\} = \text{false}$. {Corruption status at the point when the key exchange is finished. Used to make sure that corruption status requests are answered consistently also after the key exchange has finished, i.e., a corrupted user won't suddenly become uncorrupted.}

In addition to the above, $M_{\text{responder}}$ also keeps track how many pointers have already been generated by the user once the key exchange has been finished.

Corruption behavior, Initialization, MessagePreprocessing:

Identical to $M_{\text{initiator}}$, cf. Figure H.9, except for **DetermineCorrStatus** which is changed to the following algorithm:

DetermineCorrStatus($\text{pid}, \text{sid}, \text{role}$):

if $\text{userStatus} \in \{\perp, \text{started}\}$:

Let corr be the corruption status of the signature key of party pid in $\mathcal{F}_{\text{crypto}}$.

Return $\text{corr} \vee \text{corrFlag}$.

else:

Return $\text{CorrStatusAfterFinish}$.

Main:

recv (InitKE, m') **from** I/O **s.t.** $\text{userStatus} = \perp$:

$\text{userStatus} \leftarrow \text{started}$.

$\text{shello} \leftarrow m'$.

$\text{caller} \leftarrow \text{entity}_{\text{call}}$.

{Responder: Start key exchange.}

{No message is sent as the responder waits for the first KE message. Control reverts to the environment.}

The Main algorithm is continued in Figure H.12.

Figure H.11.: The modified OPTLS protocol for unilaterally authenticated key exchange $\mathcal{P}'_{\text{OPTLS}}$ (Part III).

Description of $M_{\text{responder}}$ (continued):

Main (continued):

recv (*chello*, *gx*) **from** NET **s.t.** *userStatus* = *started*, $gx \in G$: { *Responder: Receive first message, send second message, and finish KE.*

send (*IsShareFromUncorruptedUser?*, *gx*) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{uncorruptDB}} : \text{uncorruptDB})$;
wait for (*IsShareFromUncorruptedUser?*, *b*).
corrFlag $\leftarrow \neg b$.
send (*BlockDHShare*, *gx*) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** ..
send *GenExp* **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (*GenExp*, *ptr_{exp}*, *gy*).
if *DetermineCorrStatus*(*entity_{cur}*) = *true*:
 send (*RetrieveExp*, *ptr_{exp}*) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** ..
send (*GenDHKey*, *ptr_{exp}*, *gx*) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (*GenDHKey*, *ptr_{DH}*).
(*ptr_{Enc}*, *ptr_{MAC}*, *sessionKeyPointer*) \leftarrow *DeriveKeys*(*ptr_{DH}*, *chello*, *shello*). {cf. Figure H.10.
m_{Sig} \leftarrow (*gx*, *gy*).
send (*Sign*, *m_{Sig}*) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (*Sign*, σ_{Sig}).
m_{MAC} \leftarrow (*chello*, *gx*, *shello*, *gy*, σ_{Sig}).
send (*MAC*, *ptr_{MAC}*, *m_{MAC}*) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (*MAC*, σ_{MAC}).
m_{Enc} \leftarrow (σ_{Sig} , σ_{MAC}).
send (*Enc*, *ptr_{Enc}*, *m_{Enc}*) **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** (*Enc*, *c*).

CorrStatusAfterFinish \leftarrow *DetermineCorrStatus*(*entity_{cur}*). {Store the current corruption status before finishing the KE.

send responsively (*shello*, *gy*, *c*) **to** NET;
wait for .. {Leak second message without losing control. See Section 5.3.1 where this modeling is discussed for initiators in \mathcal{P}_{ISO} .
userStatus \leftarrow *exchangeFinished*. {The pointer 0 for the higher-level protocol will be mapped to *sessionKeyPointer* internally by $\mathcal{P}'_{\text{OPTLS}}$; see below.
send (*FinishKE*, 0) **to** caller.

recv *m* **from** I/O **s.t.** *userStatus* = *exchangeFinished* and *m* is a valid *New*, *Equal?*, *Enc*, *Dec*, *MAC*, *MACVerify*, or *Derive* request for $\mathcal{F}_{\text{crypto}}$:^a
{*Responder: Use keys after a successful key exchange.*

Replace pointers in *m* with internal pointers for $\mathcal{F}_{\text{crypto}}$, i.e., pointer 0 is changed to *sessionKeyPointer*, pointer 1 is changed to *sessionKeyPointer* + 1, etc.
send *m* **to** $(\epsilon, \epsilon, \mathcal{F}_{\text{crypto}} : \text{crypto})$; **wait for** *m'*.
Replace pointers in *m'* with external pointers for the higher-level protocol, i.e., pointer *sessionKeyPointer* is changed to 0, pointer *sessionKeyPointer* + 1 is changed to 1, etc.
reply *m'*.

recv *CloseSession* **from** I/O **s.t.** *userStatus* = *exchangeFinished*: {*Responder: Close session.*
userStatus \leftarrow *sessionClosed*.
reply (*CloseSession*, *ok*).

^aCf. Figure H.2 for the definition of “valid”.

Figure H.12.: The modified OPTLS protocol for unilaterally authenticated key exchange $\mathcal{P}'_{\text{OPTLS}}$ (Part IV).

Description of the protocol $\mathcal{F}_{\text{uncorruptDB}} = (\text{uncorruptDB})$:

Participating roles: uncorruptDB
Corruption model: incorruptible

Description of $M_{\text{uncorruptDB}}$:

Implemented role(s): uncorruptDB

Internal state:

– setOfShares $\subseteq \{0, 1\}^* = \emptyset$.

{ Set of all DH shares that were generated by initiators who consider themselves to be uncorrupted. }

Main:

recv (AddShareOfUncorruptedUser, share) **from** I/O:

{ Add a DH key share. }

Add share to setOfShares.

reply (AddShareOfUncorruptedUser, ok).

recv (IsShareFromUncorruptedUser?, share) **from** I/O:

{ Check whether a DH key share is uncorrupted. }

Set $b \leftarrow \text{true}$ iff share is in setOfShares; otherwise, $b \leftarrow \text{false}$.

reply (IsShareFromUncorruptedUser?, b).

Figure H.13.: The subroutine $\mathcal{F}_{\text{uncorruptDB}}$ of the modified OPTLS protocol $\mathcal{P}'_{\text{OPTLS}}$. Note that this subroutine is only part of the corruption modeling and not an algorithm that actually exists in reality.

Bibliography

- [1] M. Abdalla, M. Bellare, and P. Rogaway. The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES. In *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2001.
- [2] M. Abe and M. Ohkubo. A Framework for Universally Composable Non-committing Blind Signatures. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 435–450. Springer, 2009.
- [3] M. Abe and M. Ohkubo. A framework for universally composable non-committing blind signatures. *IJACT*, 2(3):229–249, 2012.
- [4] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Z. Béguelin, and P. Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 5–17. ACM, 2015.
- [5] M. R. Albrecht, K. G. Paterson, and G. J. Watson. Plaintext Recovery Attacks against SSH. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 16–26. IEEE Computer Society, 2009.
- [6] J. Alwen, R. Ostrovsky, H. Zhou, and V. Zikas. Incoercible Multi-party Computation and Universally Composable Receipt-Free Voting. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 763–780. Springer, 2015.
- [7] D. Antonioli, N. O. Tippenhauer, and K. B. Rasmussen. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1047–1061. USENIX Association, 2019.

- [8] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohnney, S. Engels, C. Paar, and Y. Shavitt. DROWN: Breaking TLS Using SSLv2. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 689–706. USENIX Association, 2016.
- [9] M. Backes, M. Dürmuth, D. Hofheinz, and R. Küsters. Conditional reactive simulatability. *Int. J. Inf. Sec.*, 7(2):155–169, 2008.
- [10] M. Backes and D. Hofheinz. How to Break and Repair a Universally Composable Signature Functionality. In *Information Security, 7th International Conference, ISC 2004, Palo Alto, CA, USA, September 27-29, 2004, Proceedings*, volume 3225 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2004.
- [11] M. Backes, B. Pfitzmann, and M. Waidner. The reactive simulatability (RSIM) framework for asynchronous systems. *Inf. Comput.*, 205(12):1685–1720, 2007.
- [12] C. Badertscher, C. Matt, U. Maurer, P. Rogaway, and B. Tackmann. Augmented Secure Channels and the Goal of the TLS 1.3 Record Layer. In *Provable Security - 9th International Conference, ProvSec 2015, Kanazawa, Japan, November 24-26, 2015, Proceedings*, volume 9451 of *Lecture Notes in Computer Science*, pages 85–104. Springer, 2015.
- [13] B. Barak, Y. Lindell, and T. Rabin. Protocol Initialization for the Framework of Universal Composability. *IACR Cryptology ePrint Archive*, 2004:6, 2004.
- [14] G. Barthe, B. Grégoire, S. Héraud, and S. Z. Béguelin. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.
- [15] M. Bellare and P. Rogaway. Entity Authentication and Key Distribution. In *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1993.
- [16] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffei. Refinement Types for Secure Implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pages 17–32. IEEE Computer Society, 2008.
- [17] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 535–552. IEEE Computer Society, 2015.

- [18] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with Verified Cryptographic Security. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 445–459. IEEE Computer Society, 2013.
- [19] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Z. Béguelin. Proving the TLS Handshake Secure (As It Is). In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 235–255. Springer, 2014.
- [20] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, pages 82–96. IEEE Computer Society, 2001.
- [21] B. Blanchet. A Computationally Sound Mechanized Prover for Security Protocols. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 140–154. IEEE Computer Society, 2006.
- [22] H. Böck, J. Somorovsky, and C. Young. Return Of Bleichenbacher’s Oracle Threat (ROBOT). In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 817–849. USENIX Association, 2018.
- [23] J. Brendel, M. Fischlin, F. Günther, and C. Janson. PRF-ODH: Relations, Instantiations, and Impossibility Results. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 651–681. Springer, 2017.
- [24] C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss. State Separation for Code-Based Game-Playing Proofs. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 222–249. Springer, 2018.
- [25] C. Brzuska, M. Fischlin, N. P. Smart, B. Warinschi, and S. C. Williams. Less is more: relaxed yet composable security notions for key exchange. *Int. J. Inf. Sec.*, 12(4):267–297, 2013.
- [26] C. Brzuska, M. Fischlin, B. Warinschi, and S. C. Williams. Composability of bellare-rogaway key exchange protocols. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 51–62. ACM, 2011.

- [27] J. Camenisch, M. Drijvers, and A. Lehmann. Universally Composable Direct Anonymous Attestation. In *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, volume 9615 of *Lecture Notes in Computer Science*, pages 234–264. Springer, 2016.
- [28] J. Camenisch, M. Dubovitskaya, K. Haralambiev, and M. Kohlweiss. Composable and Modular Anonymous Credentials: Definitions and Practical Constructions. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 262–288. Springer, 2015.
- [29] J. Camenisch, R. R. Enderlein, S. Krenn, R. Küsters, and D. Rausch. Universal Composition with Responsive Environments. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 807–840, 2016. See [30] for the full version.
- [30] J. Camenisch, R. R. Enderlein, S. Krenn, R. Küsters, and D. Rausch. Universal Composition with Responsive Environments. *IACR Cryptology ePrint Archive*, 2016:34, 2016.
- [31] J. Camenisch, S. Krenn, R. Küsters, and D. Rausch. iUC: Flexible Universal Composability Made Simple. In *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*, volume 11923 of *Lecture Notes in Computer Science*, pages 191–221. Springer, 2019. See [32] for the full version.
- [32] J. Camenisch, S. Krenn, R. Küsters, and D. Rausch. iUC: Flexible Universal Composability Made Simple. *IACR Cryptology ePrint Archive*, 2019:1073, 2019.
- [33] J. Camenisch, S. Krenn, and V. Shoup. A Framework for Practical Universally Composable Zero-Knowledge Protocols. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 449–467. Springer, 2011.
- [34] J. Camenisch, A. Lysyanskaya, and G. Neven. Practical yet universally composable two-server password-authenticated secret sharing. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 525–536. ACM, 2012.
- [35] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. *IACR Cryptology ePrint Archive*, 2000, 2000. also provides new versions of the

UC model from December 2005, July 2013, December 2018, August 2019, and February 2020.

- [36] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001. See [35] for updated versions.
- [37] R. Canetti. Universally Composable Signature, Certification, and Authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.
- [38] R. Canetti, S. Chari, S. Halevi, B. Pfitzmann, A. Roy, M. Steiner, and W. Z. Venema. Composable Security Analysis of OS Services. In *Applied Cryptography and Network Security - 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, 2011. Proceedings*, volume 6715 of *Lecture Notes in Computer Science*, pages 431–448, 2011.
- [39] R. Canetti, L. Cheung, D. K. Kaynar, M. D. Liskov, N. A. Lynch, O. Pereira, and R. Segala. Analyzing Security Protocols Using Time-Bounded Task-PIOAs. *Discrete Event Dynamic Systems*, 18(1):111–159, 2008.
- [40] R. Canetti, A. Cohen, and Y. Lindell. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.
- [41] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally Composable Security with Global Setup. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85. Springer, 2007.
- [42] R. Canetti and M. Fischlin. Universally Composable Commitments. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2001.
- [43] R. Canetti and S. Gajek. Universally Composable Symbolic Analysis of Diffie-Hellman based Key Exchange. *IACR Cryptology ePrint Archive*, 2010:303, 2010.
- [44] R. Canetti, S. Halevi, and J. Katz. Adaptively-Secure, Non-interactive Public-Key Encryption. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 150–168. Springer, 2005.

- [45] R. Canetti and J. Herzog. Universally Composable Symbolic Security Analysis. *J. Cryptology*, 24(1):83–147, 2011.
- [46] R. Canetti, K. Hogan, A. Malhotra, and M. Varia. A Universally Composable Treatment of Network Time. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 360–375. IEEE Computer Society, 2017.
- [47] R. Canetti and H. Krawczyk. Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels. In *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, volume 2045 of *Lecture Notes in Computer Science*, pages 453–474. Springer, 2001.
- [48] R. Canetti and H. Krawczyk. Security Analysis of IKE’s Signature-Based Key-Exchange Protocol. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 143–161. Springer, 2002.
- [49] R. Canetti and H. Krawczyk. Universally Composable Notions of Key Exchange and Secure Channels. In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2002.
- [50] R. Canetti, H. Krawczyk, and J. B. Nielsen. Relaxing Chosen-Ciphertext Security. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 565–582. Springer, 2003.
- [51] R. Canetti and T. Rabin. Universal Composition with Joint State. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 2003.
- [52] R. Canetti, D. Shahaf, and M. Vald. Universally Composable Authentication and Key-Exchange with Global PKI. In *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, volume 9615 of *Lecture Notes in Computer Science*, pages 265–296. Springer, 2016.
- [53] R. Canetti, A. Stoughton, and M. Varia. EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 167–183. IEEE, 2019.

- [54] I. Cervesato, A. D. Jaggard, A. Scedrov, J. Tsay, and C. Walstad. Breaking and fixing public-key Kerberos. *Inf. Comput.*, 206(2-4):402–424, 2008.
- [55] P. Chaidos, O. Fourtounelli, A. Kiayias, and T. Zacharias. A Universally Composable Framework for the Privacy of Email Ecosystems. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 191–221. Springer, 2018.
- [56] S. Chari, C. S. Jutla, and A. Roy. Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.
- [57] M. Chase and A. Lysyanskaya. On Signatures of Knowledge. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 78–96. Springer, 2006.
- [58] Y. Chen, M. He, S. Zeng, and X. Li. Universally composable asymmetric group key agreement protocol. In *10th International Conference on Information, Communications and Signal Processing, ICICS 2015, Singapore, December 2-4, 2015*, pages 1–6. IEEE, 2015.
- [59] S. G. Choi, J. Katz, D. Schröder, A. Yerukhimovich, and H. Zhou. (Efficient) Universally Composable Oblivious Transfer Using a Minimal Number of Stateless Tokens. *J. Cryptology*, 32(2):459–497, 2019.
- [60] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe. Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 470–485. IEEE Computer Society, 2016.
- [61] C. J. F. Cremers. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.
- [62] C. J. F. Cremers and M. Feltz. Beyond eCK: Perfect Forward Secrecy under Actor Compromise and Ephemeral-Key Reveal. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, volume 7459 of *Lecture Notes in Computer Science*, pages 734–751. Springer, 2012.

- [63] I. Damgård, D. Hofheinz, E. Kiltz, and R. Thorbek. Public-Key Encryption with Non-interactive Opening. In *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*, volume 4964 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 2008.
- [64] I. Damgård and A. Scafuro. Unconditionally Secure and Universally Composable Commitments from Physical Assumptions. In *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, volume 8270 of *Lecture Notes in Computer Science*, pages 100–119. Springer, 2013.
- [65] J. P. Degabriele and K. G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 493–504. ACM, 2010.
- [66] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1197–1210. ACM, 2015.
- [67] R. Dowsley, J. Müller-Quade, A. Otsuka, G. Hanaoka, H. Imai, and A. C. A. Nascimento. Universally Composable and Statistically Secure Verifiable Secret Sharing Scheme Based on Pre-Distributed Data. *IEICE Transactions*, 94-A(2):725–734, 2011.
- [68] D. Felsch, M. Grothe, J. Schwenk, A. Czubak, and M. Szymanek. The Dangers of Key Reuse: Practical Attacks on IPsec IKE. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 567–583. USENIX Association, 2018.
- [69] D. Fett, P. Hosseyni, and R. Küsters. An Extensive Formal Security Analysis of the OpenID Financial-Grade API. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 453–471. IEEE, 2019.
- [70] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the Browser ID SSO System. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 673–688. IEEE Computer Society, 2014.
- [71] D. Fett, R. Küsters, and G. Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1204–1215. ACM, 2016.

- [72] E. S. V. Freire, J. Hesse, and D. Hofheinz. Universally Composable Non-Interactive Key Exchange. In *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, volume 8642 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2014.
- [73] O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.
- [74] C. Hazay and M. Venkatasubramanian. On Black-Box Complexity of Universally Composable Security in the CRS Model. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 183–209. Springer, 2015.
- [75] D. Hofheinz and V. Shoup. GNUC: A New Universal Composability Framework. *J. Cryptology*, 28(3):423–508, 2015.
- [76] D. Hofheinz, D. Unruh, and J. Müller-Quade. Polynomial Runtime and Composability. *J. Cryptology*, 26(3):375–441, 2013.
- [77] K. Hogan, H. Maleki, R. Rahaeimehr, R. Canetti, M. van Dijk, J. Hennessey, M. Varia, and H. Zhang. On the Universally Composable Security of OpenStack. In *2019 IEEE Cybersecurity Development, SecDev 2019, Tysons Corner, VA, USA, September 23-25, 2019*, pages 20–33. IEEE, 2019.
- [78] ISO/IEC IS 9798-3, Entity authentication mechanisms — Part 3: Entity authentication using asymmetric techniques, 1993.
- [79] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the Security of TLS-DHE in the Standard Model. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 273–293. Springer, 2012.
- [80] F. Kiefer and M. Manulis. Universally Composable Two-Server PAKE. In *Information Security - 19th International Conference, ISC 2016, Honolulu, HI, USA, September 3-6, 2016, Proceedings*, volume 9866 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 2016.
- [81] M. Kohlweiss, U. Maurer, C. Onete, B. Tackmann, and D. Venturi. (De-)Constructing TLS 1.3. In *Progress in Cryptology - INDOCRYPT 2015 - 16th International Conference on Cryptology in India, Bangalore, India, December 6-9, 2015, Proceedings*, volume 9462 of *Lecture Notes in Computer Science*, pages 85–102. Springer, 2015.

- [82] H. Krawczyk. SIGMA: The 'SIGn-and-MAC' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 400–425. Springer, 2003.
- [83] H. Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648. Springer, 2010.
- [84] H. Krawczyk, K. G. Paterson, and H. Wee. On the Security of the TLS Protocol: A Systematic Analysis. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 429–448. Springer, 2013.
- [85] H. Krawczyk and H. Wee. The OPTLS Protocol and TLS 1.3. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 81–96. IEEE, 2016.
- [86] K. Kurosawa and J. Furukawa. Universally Composable Undeniable Signature. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 524–535. Springer, 2008.
- [87] R. Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, pages 309–320. IEEE Computer Society, 2006. See [99] for a full and revised version.
- [88] R. Küsters, J. Liedtke, J. Müller, D. Rausch, and A. Vogt. Ordinos: A Verifiable Tally-Hiding Remote E-Voting System. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020*. IEEE, 2020. To appear.
- [89] R. Küsters and D. Rausch. A Framework for Universally Composable Diffie-Hellman Key Exchange. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 881–900. IEEE Computer Society, 2017. See [90] for the full version.
- [90] R. Küsters and D. Rausch. A Framework for Universally Composable Diffie-Hellman Key Exchange. *IACR Cryptology ePrint Archive*, 2017:256, 2017.

- [91] R. Küsters, D. Rausch, and M. Simon. Accountability in a Permissioned Blockchain: Formal Analysis of Hyperledger Fabric. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020*. IEEE, 2020. To appear.
- [92] R. Küsters, E. Scapin, T. Truderung, and J. Graf. Extending and Applying a Framework for the Cryptographic Verification of Java Programs. In *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 220–239. Springer, 2014.
- [93] R. Küsters, T. Truderung, and J. Graf. A Framework for the Cryptographic Verification of Java-Like Programs. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 198–212. IEEE Computer Society, 2012.
- [94] R. Küsters and M. Tuengerthal. Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pages 270–284. IEEE Computer Society, 2008. See [98] for the full version.
- [95] R. Küsters and M. Tuengerthal. Universally Composable Symmetric Encryption. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 293–307. IEEE Computer Society, 2009.
- [96] R. Küsters and M. Tuengerthal. Composition theorems without pre-established session identifiers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 41–50. ACM, 2011.
- [97] R. Küsters and M. Tuengerthal. Ideal Key Derivation and Encryption in Simulation-Based Security. In *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, volume 6558 of *Lecture Notes in Computer Science*, pages 161–179. Springer, 2011.
- [98] R. Küsters, M. Tuengerthal, and D. Rausch. Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation. *IACR Cryptology ePrint Archive*, 2008:6, 2008. To appear in *Journal of Cryptology*.
- [99] R. Küsters, M. Tuengerthal, and D. Rausch. The IITM Model: a Simple and Expressive Model for Universal Composability. *IACR Cryptology ePrint Archive*, 2013:25, 2013. To appear in *Journal of Cryptology*.

- [100] P. Laud and L. Ngo. Threshold Homomorphic Encryption in the Universally Composable Cryptographic Library. In *Provable Security, Second International Conference, ProvSec 2008, Shanghai, China, October 30 - November 1, 2008. Proceedings*, volume 5324 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2008.
- [101] T. Matsuo and S. Matsuo. On Universal Composable Security of Time-Stamping Protocols. In *Applied Public Key Infrastructure - 4th International Workshop: IWAP 2005, Singapore, September 21-23, 2005*, volume 128 of *Frontiers in Artificial Intelligence and Applications*, pages 169–181. IOS Press, 2005.
- [102] U. Maurer. Constructive Cryptography - A New Paradigm for Security Definitions and Proofs. In *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 2011.
- [103] U. Maurer and R. Renner. Abstract Cryptography. In *Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 7-9, 2011. Proceedings*, pages 1–21. Tsinghua University Press, 2011.
- [104] U. M. Maurer and S. Wolf. Diffie-Hellman Oracles. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 268–282. Springer, 1996.
- [105] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.
- [106] S. J. Murdoch, S. Drimer, R. J. Anderson, and M. Bond. Chip and PIN is Broken. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, pages 433–446. IEEE Computer Society, 2010.
- [107] B. Pfitzmann and M. Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In *2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001*, pages 184–200. IEEE Computer Society, 2001.
- [108] E. Rescorla. The transport layer security (TLS) protocol version 1.3 (draft 09), October 2015. <https://tools.ietf.org/html/draft-ietf-tls-tls13-09>.
- [109] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.

- [110] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.
- [111] Y. Tian and C. Peng. Universally Composable Secure Group Communication. *IACR Cryptology ePrint Archive*, 2014:647, 2014.
- [112] M. Tuengerthal. *Analysis of real-world security protocols in a universal composability framework*. PhD thesis, University of Trier, 2013.
- [113] M. Vanhoef and F. Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1313–1328. ACM, 2017.
- [114] D. Wikström. Simplified Universal Composability Framework. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*, volume 9562 of *Lecture Notes in Computer Science*, pages 566–595. Springer, 2016.
- [115] B. Yuan and J. Liu. A universally composable secure grouping-proof protocol for RFID tags. *Concurrency and Computation: Practice and Experience*, 28(6):1872–1883, 2016.
- [116] S. Zhao, Q. Zhang, Y. Qin, and D. Feng. Universally Composable Secure TNC Protocol Based on IF-T Binding to TLS. In *Network and System Security - 8th International Conference, NSS 2014, Xi'an, China, October 15-17, 2014, Proceedings*, volume 8792 of *Lecture Notes in Computer Science*, pages 110–123. Springer, 2014.

Academic Curriculum and Publications

Academic Curriculum

- January 2017 - **University of Stuttgart, Germany**
Ph.D. student at the Institute of Information Security
Supervisor: Prof. Dr. Ralf Küsters
- October 2014 - December 2016 **University of Trier, Germany**
Ph.D. student at the Chair of
Information Security and Cryptography
Supervisor: Prof. Dr. Ralf Küsters
- October 2012 - September 2014 **University of Trier, Germany**
Master of Science in Computer Science
Thesis title: *Modulare kryptographische Analyse
des PeerKey Handshakes in WPA2*
Supervisor: Prof. Dr. Ralf Küsters
- October 2009 - September 2012 **University of Trier, Germany**
Bachelor of Education in
Computer Science and Mathematics
Thesis title: *Interaktives Zusammenführen
von Hierarchien*
Supervisor: Prof. Dr. Stephan Diehl

List of Publications

- J. Camenisch, R. R. Enderlein, S. Krenn, R. Küsters, and D. Rausch. Universal Composition with Responsive Environments. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 807–840, 2016.
- J. Camenisch, S. Krenn, R. Küsters, and D. Rausch. iUC: Flexible Universal Composability Made Simple. In *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*, volume 11923 of *Lecture Notes in Computer Science*, pages 191–221. Springer, 2019.

- R. Küsters, J. Liedtke, J. Müller, D. Rausch, and A. Vogt. Ordinos: A Verifiable Tally-Hiding Remote E-Voting System. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020*. IEEE, 2020. To appear.
- R. Küsters and D. Rausch. A Framework for Universally Composable Diffie-Hellman Key Exchange. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 881–900. IEEE Computer Society, 2017.
- R. Küsters, D. Rausch, and M. Simon. Accountability in a Permissioned Blockchain: Formal Analysis of Hyperledger Fabric. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020*. IEEE, 2020. To appear.
- R. Küsters, M. Tuengerthal, and D. Rausch. Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation. *IACR Cryptology ePrint Archive*, 2008:6, 2008. To appear in Journal of Cryptology.
- R. Küsters, M. Tuengerthal, and D. Rausch. The IITM Model: a Simple and Expressive Model for Universal Composability. *IACR Cryptology ePrint Archive*, 2013:25, 2013. To appear in Journal of Cryptology.
- R. Lutz, D. Rausch, F. Beck, and S. Diehl. A directory comparison and manipulation tool. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, pages 213–214. IEEE Computer Society, 2014.
- R. Lutz, D. Rausch, F. Beck, and S. Diehl. Get your directories right: From hierarchy visualization to hierarchy manipulation. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, pages 25–32. IEEE Computer Society, 2014.