

Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Masterarbeit

Accountable Secure Multi-Party Computation for Tally-Hiding E-Voting

Marc Rivinius

Course of Study: Informatik

Examiner: Prof. Dr. Ralf Küsters

Supervisor: Julian Liedtke, M.Sc.

Commenced: November 19, 2019

Completed: July 14, 2020

Abstract

With multi-party computation becoming more and more efficient and thus more practical, we can start to investigate application scenarios. One application where multi-party computation can be used to great effect is e-voting. Unlike classical e-voting protocols, one can get tally-hiding e-voting systems. There, some part of the tally (especially the whole set of votes) is not made public.

Notwithstanding this, most existing (verifiable) multi-party computation protocols are not suitable for e-voting. A property that is arguably more important than verifiability is missing: accountability – as a matter of fact, we need external accountability in this setting, where anyone audit the protocol. This is especially of importance for e-voting systems and more researchers are paying attention to it lately.

To this effect, we introduce a general multi-party computation protocol that meets all the requirements to be used in e-voting systems. Our protocol achieves accountability and fairness in the honest majority setting and is – to our best knowledge – the first one to do so.

Dedication

This thesis is dedicated to my family – my mother, my father, my brother, and my grandmothers. I want to thank them for their unconditional love and support.

And I am so grateful to my fiancée. Thank her for proof-reading this thesis and, of course, for her loving support.

I also want to use this opportunity to thank everyone who gave their best during this crisis. I wish there was more that I could have done myself, but I hope I can help to create a better future.

Contents

1	Introduction	19
2	Background	21
2.1	Notation and Basics	21
2.2	Secret Sharing Schemes	26
2.3	Encryption Schemes	28
2.4	Commitment Schemes	36
2.5	Universal Composability Framework	40
2.6	Zero-Knowledge	44
2.7	Multi-Party Computation	55
2.8	MPC Security Properties	57
2.9	E-Voting	60
3	Related Work	65
3.1	MPC Protocols	65
3.2	E-Voting Systems	68
4	Formal MPC Protocol	73
4.1	Prerequisite Functionalities	74
4.2	Main Protocol	76
4.3	Protocols for the Prerequisite Functionalities	86
5	Possible Implementations	97
5.1	Implementing a Bulletin Functionality	97
5.2	Implementing the Key Generation	98
5.3	Somewhat Homomorphic Encryption Scheme	99
5.4	Multiple Encryption Schemes	100
5.5	Multiple Commitment Schemes	100
5.6	Leveraging Preprocessing	100
5.7	Speed-Up in Case of Risk-Avoiding Adversaries	101
5.8	More Efficient Online Phase with Full-Threshold Secret Sharing	103
5.9	On Arithmetic Circuits	105
6	Security Analysis	107
6.1	Privacy	107
6.2	Correctness	107
6.3	Guaranteed Output Delivery	107
6.4	Fairness	108
6.5	Accountability	109
6.6	Verifiability	111

7	Comparison to Other MPC Protocols	115
7.1	Auditable SPDZ	116
7.2	Accountable SPDZ	117
8	Application to E-Voting Systems	119
8.1	Reusing Existing Components	119
8.2	Ordinos	120
8.3	Other E-Voting Systems	120
9	Conclusion and Outlook	123
	Bibliography	125
A	Accountable Secure Multi-Party-Computation für Tally-Hiding E-Voting	133

List of Figures

2.1	Security Games for Encryption Schemes	31
2.2	CCA-Adversary for Additive Homomorphic Encryption Schemes	31
2.3	Security Games for Commitment Schemes	37
2.4	Canonical Encryption and Commitment Procedures	39
2.5	Example for Connectivity of Interactive Turing Machines	41
2.6	Dummy Adversary	42
2.7	Interactive Proof Systems	46
2.8	Standard Σ -Protocol for a Homomorphic Function f	50
2.9	Extractor and Simulator for Σ_{std}	51
2.10	CPA-Adversary Using Zero-Knowledge Proofs	53
4.1	Functionality to Generate a Random Cipher	75
4.2	Functionality to Generate a Committed Cipher	76
4.3	Functionality for Threshold Decryption	77
4.4	Functionality for an Accountable Multi-Party Computation	78
4.5	Protocol for an Accountable Multi-Party Computation	79
4.6	Simulator for the Protocol for an Accountable Multi-Party Computation	86
4.7	Protocol Interactions to Distinguish for an Environment	87
4.8	Alternative Functionality to Generate a Random Cipher	89
4.9	Protocol to Generate a Random Cipher	90
4.10	Simulator for the Protocol to Generate a Random Cipher	91
4.11	Alternative Functionality to Generate a Committed Cipher	92
4.12	Protocol to Generate a Committed Cipher	93
4.13	Simulator for the Protocol to Generate a Committed Cipher	94
4.14	Protocol for Threshold Decryption	95
4.15	Simulator for the Protocol for Threshold Decryption	96
5.1	Functionality to Generate a Random Shared Cipher	103
5.2	Changes to the Protocol if Full-Threshold Secret Sharing is Used	104
5.3	Substitution of Multiplication Gates	106
6.1	Additions to Functionality and Protocol for the Accountability Goal	109

List of Tables

2.1	Notation of Spaces and Operations	22
2.2	Notation for Algorithms, Interactive Machines, and Protocols	23
2.3	Notation for Accountability and Verifiability in the KTV-Framework	58
2.4	Examples of Voting Protocol Sequences	61
5.1	Equivalent Operations When Using a Bulletin Board or Broadcast Channels	98
6.1	Overview of the Security Analysis	108
7.1	Public and Private Information of the Protocols	115
7.2	Complexity of the Protocols in the Online Phase	117

List of Definitions

2.1	Definition (Homomorphisms)	24
2.2	Definition (Negligible Functions)	25
2.3	Definition (Overwhelming Functions)	25
2.4	Definition (δ -Bounded Functions)	25
2.5	Definition (Indistinguishability)	26
2.6	Definition (Asymmetric Encryption Schemes)	28
2.7	Definition (Additive Homomorphic Encryption Schemes)	29
2.8	Definition (Multiplicative Homomorphic Encryption Schemes)	29
2.9	Definition (CPA-Secure Encryption Schemes)	29
2.10	Definition (CCA-Secure Encryption Schemes)	30
2.11	Definition (Commitment Schemes)	36
2.12	Definition (Computationally Hiding Commitment Schemes)	38
2.13	Definition (Computationally Binding Commitment Schemes)	38
2.14	Definition (Perfectly Hiding Commitment Schemes)	38
2.15	Definition (Perfectly Binding Commitment Schemes)	38
2.16	Definition (UC-Framework: Emulation)	43
2.17	Definition (UC-Framework: Realizing Ideal Functionalities)	43
2.18	Definition (Interactive Proof Systems)	45
2.19	Definition (Computational Zero-Knowledge)	46
2.20	Definition (Perfect Zero-Knowledge)	47
2.21	Definition (NP-Languages)	47
2.22	Definition (Proofs of Knowledge)	48
2.23	Definition (Σ -Protocols)	49
2.24	Definition (Accountability)	59
2.25	Definition (Verifiability)	60

List of Theorems

2.1	Corollary (Multiplication-with-Constants Using Homomorphisms)	24
2.2	Theorem (Homomorphic Encryption Schemes are not CCA-Secure)	30
2.3	Theorem (ElGamal Encryption is CPA-Secure)	34
2.4	Theorem (Paillier Encryption is CPA-Secure)	34
2.5	Theorem (RLWE Encryption is CPA-Secure)	36
2.6	Theorem (Pedersen Commitments are Hiding and Binding)	39
2.7	Theorem (Encryption Schemes as Commitment Schemes)	39
2.8	Theorem (UC-Framework: Composition)	44
2.9	Corollary (UC-Framework: Composition of Ideal Functionalities)	44
2.10	Corollary (IPs with Better Completeness and Soundness Bounds)	45
2.11	Corollary (NP-Languages)	47
2.12	Theorem (NP-Languages have Zero-Knowledge Proofs)	47
2.13	Theorem (Proving NP-Languages in Polynomial Time)	48
2.14	Theorem (Σ_{std} is a Σ -Protocol)	49
2.15	Corollary (Σ_{std} is a Proof of Plaintext-Knowledge)	52
2.16	Theorem (Proofs of Plaintext-Knowledge Require Information)	53
2.17	Theorem (Proofs of Decryption Require Information)	54
2.18	Corollary (Σ_{std} is a Proof of Committed Ciphers)	54
4.1	Theorem (Π_{MPC} Realizes \mathcal{F}_{MPC})	84
4.2	Theorem ($\Pi_{\text{rand-cipher}'}$ Realizes $\mathcal{F}_{\text{rand-cipher}'}$)	88
4.3	Theorem ($\Pi_{\text{com-cipher}'}$ Realizes $\mathcal{F}_{\text{com-cipher}'}$)	91
4.4	Theorem (Π_{key} Realizes \mathcal{F}_{key})	93
6.1	Theorem (Accountability of \mathcal{F}_{MPC})	110
6.2	Theorem (Verifiability of \mathcal{F}_{MPC})	111
6.3	Corollary (Accountability of Π_{MPC})	113
6.4	Corollary (Verifiability of Π_{MPC})	114

Acronyms

- BDOZ** Bendlin–Damgård–Orlandi–Zakarias; a multi-party computation protocol [22]. 68
- BGV** Brakerski–Gentry–Vaikuntanathan; an encryption scheme [27]. 35
- CCA** chosen-ciphertext attack. 29
- CPA** chosen-plaintext attack. 29
- EUFCMA** existential unforgeability chosen-message attack. 70
- FHE** fully homomorphic encryption. 56
- GSW** Gentry–Sahai–Waters; an encryption scheme [60]. 35
- IPS** interactive proof system. 45
- ITM** interactive Turing machine. 40
- KTV** Küsters–Truderung–Vogt; a framework for accountability and verifiability [81]. 58
- LWE** learning with errors. 35
- MAC** message authentication code. 65
- MPC** multi-party computation. 55
- NP** non-deterministic polynomial-time; a complexity class. 47
- PPT** probabilistic polynomial-time. 26
- RLWE** ring learning with errors. 35
- SHE** somewhat homomorphic encryption. 56
- SIMD** single instruction, multiple data. 35
- SPDZ** Smart–Pastro–Damgård–Zakarias; a multi-party computation protocol [49]. 65
- TM** Turing machine. 40
- UC** universal composability. 40
- ZK** zero-knowledge. 46

1 Introduction

General multi-party computation has been a topic of interest in cryptography for many years. In multi-party computation, each party chooses their secret input and together the parties can compute arbitrary functions without anyone learning more than the result. Obviously, this can be used in various applications and solves problems that (at first glance) seem to be impossible to solve.¹ One such application can be found in electronic voting (e-voting). Determining the winner(s) of elections can be seen as such a function that could be evaluated with multi-party computation protocols. Most proposed e-voting systems do not go that far to consider the whole election result as a function of the votes, but they reveal the sum of the votes for each candidate – or even all votes – and everyone can compute the election result. Not doing so (e. g., only revealing the result) can have several benefits [79] – especially in smaller elections where revealing the votes could reduce the privacy of some voters or embarrass candidates that get a small number of votes. Another example is an election with multiple rounds, where only revealing the winners of a round can help prevent bias in subsequent rounds. There, not revealing the number of votes for each candidate that advances to the next round can influence the voting behavior in the next round, where some voters might vote differently (i. e., some voters do not want to “waste their vote” on candidates that get a low number of votes, even if these candidates are their preferred choice).

While existing general multi-party computation protocols can be used to facilitate this so-called tally-hiding e-voting, they usually have a severe drawback. Most state-of-the-art protocols are only verifiable, e. g., they produce a result and parties (or anyone) can check if it is correct or not, while they might even abort – where we do not get any result. This is obviously subpar for e-voting systems, as we do not get a result and usually have to rerun either the computation or – in the worst case – the whole election. While we cannot prevent wrong results or abort without assuming someone (doing the computation) is completely trustworthy, we can at least do something to discourage behavior that leads to such results.

With protocols that are accountable, we do not only get to know that something is wrong (e. g., the result is wrong or the protocol aborted), we can also find the cause for it. More specifically, we can identify trustees that misbehaved – in the best case even individually (e. g., we know exactly who misbehaved, not only a group of which at least one misbehaved). Accountability seems to be a highly valuable security property of multi-party computation protocols for this very reason and not only e-voting systems can benefit from it.

¹for example, to compare two numbers that two persons hold without revealing “anything” about the values of the numbers

As a malicious actor has to fear to get caught when cheating, it should greatly increase the likelihood of running the protocol without misbehavior, as misbehavior can result in sanctions. For example, in an official election, misbehavior could be illegal and have severe consequences. In other applications, one could at least exclude the malicious actor from future computations and possibly take legal action.

Accountability can be used even further to actually improve the guarantees of multi-party computation protocols. If malicious actors can be already identified during the protocol run, the other parties can try to compensate for them, e. g., they can use knowledge about misbehavior to still produce correct results in some cases. Using well-known results about multi-party computation, we can use the fact that the result of any function can be computed if less than half of the involved parties are malicious. Secret sharing and threshold decryption enable this in our case. While we cannot guarantee anything if more than half of the parties misbehave, we argue that accountability – that is achievable even if all parties are malicious – might deter enough parties so at least half of the parties act honest, even if they have malicious intentions.

In this master’s thesis, we introduce such a multi-party computation protocol that achieves public verifiability, accountability, and fairness for honest majorities. We also show how to combine it with e-voting systems to support tally-hiding elections.

Organization

First, we introduce some background that is necessary to understand our protocol. This is done in Chapter 2 and includes various topics from secret sharing schemes and encryption to multi-party computation and e-voting. Then, we introduce several related systems in Chapter 3. Here, protocols that inspired our accountable multi-party computation protocol and several e-voting systems are described. After that we come to our protocol in Chapter 4. We describe it and several sub-protocols needed to implement it. Additionally, we prove that they are secure in the UC-framework. In Chapter 5, we outline several small modifications that can be applied to our protocol. Chapter 6 contains the security analysis of our protocol with respect to security properties of multi-party computation protocols. This includes accountability and verifiability that we prove in the KTV-framework. After that, we compare our protocol to related multi-party computation protocols. This happens in Chapter 7. Finally – in Chapter 8 – we show how existing e-voting systems can use our protocol. Appendix A contains a summary of this thesis in German language.

2 Background

In this chapter, we present the required background for our protocols. More details can be found in the literature [45, 62, 71].

First, we give some general notation and definitions (Section 2.1) needed for the rest of this work. After this, we introduce the main primitives needed for our multi-party computation protocol: secret sharing schemes, encryption schemes, and commitment schemes (in Sections 2.2 to 2.4). This includes commonly used schemes: full-threshold secret sharing and Shamir secret sharing, ElGamal and Pailler encryption, and Pedersen commitments. These are used in related protocols and also in e-voting systems – our protocol can use all of them as well. Additionally, the RLWE encryption scheme of SPDZ [49] is presented. It acts as an efficient somewhat homomorphic encryption scheme for our protocol as well.

Then, we discuss the background related to protocols. The UC-framework (Section 2.5) gives definitions for interactive Turing machines and protocols. It also allows us to prove that our protocol can be composed, e. g., used together with other protocols and instances of itself. With a notion of interactive machines, we can also discuss zero-knowledge (Section 2.6). Zero-knowledge proofs allow us to prove statements without revealing secret information and are used as primitives in the protocol. They are used to prevent cheating and also reveal which parties misbehaved (as misbehaving parties fail to provide valid proofs for the wrong values they send). After this, multi-party computation is discussed. First, the general idea and setting are described in Section 2.7. Then, we give the security properties that one wants to achieve in Section 2.8. This includes accountability and verifiability. For this, we also introduce the KTV-framework designed to formalize these two properties. Lastly – in Section 2.9 – we talk about e-voting. This includes common techniques and security properties.

2.1 Notation and Basics

First, we will discuss the notation used in this thesis. A summary of the notation for the most important spaces and operations on their elements can be found in Table 2.1. Note that the secret sharing schemes we use (see Section 2.2) do not require a distinction between \mathbb{X} and \mathbb{S} , e. g., we can assume $\mathbb{X} = \mathbb{S}$. but we make the distinction as they are semantically different. In contrast to this, \mathbb{R} can be actually different for encryption schemes and commitment schemes,¹ e. g., $\mathbb{R}_{\mathcal{E}}$ for an encryption scheme \mathcal{E} and $\mathbb{R}_{\mathcal{C}}$ for a commitment scheme \mathcal{C} should be used. As we do not use a randomness for one also for the other, it should be always clear which \mathbb{R} we mean. We need to encrypt commitment-randomness at one point (see Sections 2.6.10 and 4.3.2), but this is possible with the commitment schemes presented here. The set of all (compute) parties is defined

¹and different from \mathbb{X}

2 Background

Space	Notation for Elements	Addition*	Subtraction*	Multiplication*	Multiplication with Plaintext**	
Plaintexts	\mathbb{X}	x	+	-	\cdot	n/a^\dagger
Randomness	\mathbb{R}	\tilde{x}	+	-	n/a^\ddagger	\times
Shares	\mathbb{S}	$\langle x \rangle_i$	+	-		\cdot
Ciphertexts	\mathbb{Y}	(x)	\oplus	\ominus	\odot	\otimes
Commitments	\mathbb{C}	$\llbracket x \rrbracket$	\boxplus	\boxminus	n/a^\ddagger	\boxtimes
Views	$\mathbb{S} \times \mathbb{R} \times \mathbb{C}^n$	$[x]_i$	+	-		\cdot
Public Keys	\mathbb{K}	k				
Private Keys	$\hat{\mathbb{K}}$	\hat{k}			n/a	
Commitment Parameters	$\check{\mathbb{K}}$	\check{k}				
Parties	\mathbb{P}	\mathbb{P}_i				

* For space A , the operation is $A \times A \rightarrow A$.

** For space A , the operation is $\mathbb{X} \times A \rightarrow A$.

† Is the same as multiplication.

‡ The protocol is designed in a way that shares never have to be multiplied. Therefore, randomness and commitments require no multiplication as well.

Table 2.1: Notation of spaces and operations.

as $\mathbb{P} = \{\mathbb{P}_1, \dots, \mathbb{P}_n\}$ (e. g., parties that will later compute an arithmetic circuit) and the number of parties is n . The currently participating parties in the compute phase of the protocol are $\mathbb{P}_{\text{curr}} \subseteq \mathbb{P}$ – this is the set of parties that are assumed to be non-malicious by an honest party. t is the minimal number of parties needed for reconstruction in the secret sharing scheme (see Section 2.2) and the minimal number needed for threshold decryption (see Section 2.3.2). For vectors (lists or tuples) of values we use an arrow-notation, i. e., \vec{y} .

More general notation concerning algorithms, interactive Turing machines, and protocols can be found in Table 2.2. Additionally, we require security games to return either win or lose to indicate if an adversary won or lost the game. Similarly, zero-knowledge proof verifiers should return only accept or reject. Also, an environment that is supposed to distinguish an ideal functionality and a protocol is supposed to return either ideal or real. η is the security parameter for games and algorithms. For protocols of interactive Turing machines, it is assumed that 1^η (e. g., a bit-string of length η consisting of only 1s) is part of each sent message.

Formal languages \mathcal{L} are defined over an alphabet Λ , e. g., $\mathcal{L} \subseteq \Lambda^*$. Usually we assume words are bit-strings: $\Lambda = \{0, 1\}$. We use $\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z} = \{z \bmod n \mid z \in \mathbb{Z}\} = \{0, \dots, n-1\}$ for integers modulo n . $\mathbb{Z}_n^* = \{z \in \mathbb{Z}_n \mid \exists z' \in \mathbb{Z}_n : z \cdot z' \bmod n = 1\}$ is the subset of \mathbb{Z}_n that is invertible w. r. t. multiplication modulo n . Real and natural numbers are \mathbb{R} and \mathbb{N} , respectively, while a superscript plus (i. e., \mathbb{R}^+) indicates the subset that only includes positive numbers.

For algorithms (and functions) we use the following notation:

$$f: \quad A \rightarrow B, \quad a \mapsto f(a).$$

Sample Statement	Meaning
$\text{Coins}(\text{ALG})$	The space of random coins that an algorithm uses. Might also be used for interactive Turing machines or protocols. In the latter case, it is the Cartesian product of the random-coin spaces of each machine of the protocol.
$(M_0 \parallel M_1)$	The protocol that is formed by combining the interactive Turing machines M_0 and M_1 . It can also be used to combine interactive machines and protocols. Then, the resulting protocol is formed by adding the machines to the protocol.
$\Pi(x)$	“Running” the protocol Π with input x . We define all main machines to get this the input x . If the output of such a run is used (i. e., “ $y \leftarrow \Pi(x)$ ”), only a single main machine is allowed to output a value.
$\Pr[\Pi(x) = v]$	The probability that a protocol outputs the value v on input x . The probability is taken over the random coins of the protocol.
$\Pr[\Pi(x) \mapsto \{M : v \mid P(v)\}]$	The probability that the machine M outputs a v with $P(v)$ when the protocol Π gets the input x .
$\text{Trans}(\Pi, x)$	The transcript of protocol Π on input x . This includes all messages send between the interactive Turing machines that are part of the protocol.

(a) Operations on algorithms, interactive machines, and protocols.

Sample Statement	Meaning
$x \leftarrow v$	The value v is assigned to the variable x .
$x \leftarrow \text{ALG}()$	The output of algorithm ALG is assigned to the variable x .
$x \stackrel{\$}{\leftarrow} A$	An element of the set A is selected uniformly at random and assigned to the variable x .
$x \stackrel{\$}{\leftarrow} \text{ALG}()$	Used to emphasize that ALG is a non-deterministic algorithm. It is equivalent to $\alpha \stackrel{\$}{\leftarrow} \text{Coins}(\text{ALG})$ $x \leftarrow \text{ALG}^\alpha()$.

(b) Assignment notation in algorithms.

Table 2.2: Notation for algorithms, interactive machines, and protocols.

If we only specify the domain and co-domain of an algorithm (i. e., $f : A \rightarrow B$), we do not imply that the algorithm is deterministic (like a function) but only try to use a consistent notation for functions and algorithms. Finally, we end definitions with \diamond and proofs with \square .

2.1.1 Terminology

There are alternative terms for a few terms that we use in this work. What we call CPA-security is also called IND-CPA-security to highlight the fact that we are interested in indistinguishability. Similarly, our notion of CCA-security could be called IND-CCA-security or IND-CCA2-security. The last notion highlights that we give the decryption oracle ENC^{-1} to the attacker in both phases of the security game (see Figure 2.1b). The Fiat–Shamir transform (see Section 2.6.7) is also called Fiat–Shamir heuristic.

We use the terms encryption scheme and cryptosystem interchangeably. We call the protocol of Baum et al. [12] auditable SPDZ and the protocol of Cunningham et al. [44] accountable SPDZ – even though there are other SPDZ-like protocols that are auditable or accountable as well (see Sections 3.1.2 and 3.1.3 for descriptions of the protocols). The other terms and acronyms we use are commonly used in the field of cryptography.

2.1.2 Miscellaneous

Now, we will give some definitions that are used later on. They mainly cover homomorphisms and negligible functions (and types of functions related to negligible functions). The latter is used for many other definitions throughout this work.

Definition 2.1 (Homomorphisms).

We call a function $f: A \rightarrow B$ homomorphic if

$$\forall x, y \in A : \quad f(x) + f(y) = f(x + y)$$

with respective (and potentially different) $+$ operations in A and B . ◇

Remark. Note that $+$ might be different from an addition, i. e., it might be a multiplication or an otherwise defined operation.

Corollary 2.1 (Multiplication-with-Constants Using Homomorphisms).

Let $f: A \rightarrow B$ be a homomorphism. For every natural number $n \in \mathbb{N}$, we have

$$\forall x \in A : \quad n \cdot f(x) = f(n \cdot x)$$

with the respective \cdot in A and B defined as repeated application of the $+$ operation.

Proof (Sketch). A proof by induction can be done using the definition of homomorphisms and the definition of multiplication. □

Remark 1. If $(A, +)$ is a finite group, we can reduce the number of additions needed to multiply with n . If $x \in A$ generates a sub-group A_x , we have

$$\forall m \in \mathbb{N}^+ : \quad n \cdot x = (n \bmod m \cdot |A_x|) \cdot x$$

which implies

$$\forall m \in \mathbb{N}^+ : \quad n \cdot f(x) = (n \bmod m \cdot |A_x|) \cdot f(x).$$

If all $x \in A$ generate the same sub-group (or there is a natural number $m \in \mathbb{N}^+$ that is a multiple of $|A_x|$ for all x), one can use this fact to multiply any $f(x)$ without knowing x .

Remark 2. This can be used to multiply with negative numbers, invert elements, or subtract elements (of A and B).² For example with $|A_x| = m$:

$$\begin{aligned} -4 \cdot x &= (-4 \bmod m) \cdot x & -4 \cdot f(x) &= (-4 \bmod m) \cdot f(x) \\ -x &= (-1 \bmod m) \cdot x & -f(x) &= (-1 \bmod m) \cdot f(x) \\ y - x &= y + (-1 \bmod m) \cdot x & f(y) - f(x) &= f(y) + (-1 \bmod m) \cdot f(x). \end{aligned}$$

Remark 3. Various fast algorithms are available to do fast multiplication of elements [68].

Definition 2.2 (Negligible Functions).

A function $f: \mathbb{N} \rightarrow [0, 1]$ is negligible if

$$\forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n > n_0 : \quad f(n) \leq \frac{1}{n^c}. \quad \diamond$$

Definition 2.3 (Overwhelming Functions).

A function f is overwhelming if $1 - f$ is negligible. ◇

Definition 2.4 (δ -Bounded Functions).

A function f is δ -bounded if $f - \delta$ is negligible. ◇

Remark. We might say a probability of something is negligible (or overwhelming or δ -bounded) in this thesis. Generally, it will be clear what the parameter will be with respect to which we require negligibility. For example, for

$$\Pr[\text{ALG}(1^\eta) = v]$$

we require negligibility w. r. t. the input of ALG. As this is not a natural number but a bit-string, we mean negligibility w. r. t. the input length, e. g., $|1^\eta| = \eta$. Another example is a security game. A negligible probability

$$\Pr[\mathfrak{G}(\eta) = \text{win}]$$

would mean that we require the probability to win to be negligible w. r. t. η .

²If no such operations were defined already.

Definition 2.5 (Indistinguishability).

Let \mathcal{L} be a language. The probability ensembles $\{X_e\}_{e \in \mathcal{L}}$ and $\{Y_e\}_{e \in \mathcal{L}}$ are computationally indistinguishable if

$$\forall A \forall e \in \mathcal{L} : \quad |\Pr[A(X_e, e) = 1] - \Pr[A(Y_e, e) = 1]|$$

is negligible. Note that A is a PPT Turing machine, e. g., a probabilistic Turing machine with polynomially bounded runtime. \diamond

Remark. The output of A should indicate if it “thinks” its input came from $\{X_e\}_{e \in \mathcal{L}}$ or $\{Y_e\}_{e \in \mathcal{L}}$.

One cryptographic primitive we do not describe in detail are hash functions. A central property of these functions is that it should be hard to find two inputs that map to the same output. More details can be found on literature on the topic [62, 71, 103, 107]. (Digital) signature schemes – used to verify that a message originated from a specific sender and was not changed – are not discussed in this chapter as well [62, 71].

2.2 Secret Sharing Schemes

Secret sharing is an important foundation for multi-party computation. It enables computations where no single party knows the data that is operated on, and also threshold encryption. If shares are constructed in a correct way, the reconstruction algorithm

$$\text{REC} : \quad (\mathbb{S} \cup \{\varepsilon\})^n \rightarrow \mathbb{X} \cup \{\perp\}$$

can compute a shared value x from its shares $\langle x \rangle_i$ for $\mathbb{P}_i \in \mathbb{P}$. ε denotes that a share is “not available” (i. e., if we find out that a share is not correct or a party is not trusted) or not given and \perp denotes that the reconstruction is not possible because not enough shares (less than a threshold t) were given. A formal definition of a secret sharing scheme would also include a sharing procedure to generate the shares from a given plaintext. As we do not directly use such a procedure but protocols that only work similarly, we do not define them here, but we describe the general idea of generating shares. Additionally, we want a secret sharing scheme to be additive homomorphic, e. g.,

$$\begin{aligned} \text{REC}(\langle x \rangle_1 + \langle y \rangle_1, \dots, \langle x \rangle_n + \langle y \rangle_n) &= \text{REC}(\langle x \rangle_1, \dots, \langle x \rangle_n) + \text{REC}(\langle y \rangle_1, \dots, \langle y \rangle_n) \\ &= x + y \end{aligned}$$

for correct shares $\langle x \rangle_i$ of x and shares $\langle y \rangle_i$ of y . We introduce two popular but equally essential secret sharing schemes.

2.2.1 Full-Threshold Secret Sharing

In full-threshold secret sharing, every party \mathbb{P}_i gets a share $\langle x \rangle_i$ of a value x such that

$$(2.1) \quad x = \sum_{i=1}^n \langle x \rangle_i.$$

For this kind of secret sharing scheme, the shares of all parties are required for reconstruction, e. g., $t = n$. The reconstruction is then simply

$$\text{REC:} \quad (\langle x \rangle_1, \dots, \langle x \rangle_n) \mapsto \begin{cases} \perp & \bigvee_{i=1}^n \langle x \rangle_i = \varepsilon \\ \sum_{i=1}^n \langle x \rangle_i & \text{otherwise} \end{cases}.$$

Obviously, this secret sharing scheme is additive homomorphic, e. g., we can add up the shares of two shared values x and y to get shares that reconstruct to the sum of x and y :

$$\begin{aligned} \text{REC}(\langle x \rangle_1 + \langle y \rangle_1, \dots, \langle x \rangle_n + \langle y \rangle_n) &= \sum_{i=1}^n \langle x \rangle_i + \langle y \rangle_i \\ &= \sum_{i=1}^n \langle x \rangle_i + \sum_{i=1}^n \langle y \rangle_i \\ &= x + y, \end{aligned}$$

assuming we are given all shares (e. g., $\langle x \rangle_i \neq \varepsilon$ and $\langle y \rangle_i \neq \varepsilon$). To get shares that fulfill Equation (2.1), one could simply pick $\langle x \rangle_i$ uniformly at random for $i \neq 1$ and set $\langle x \rangle_1 = x - \sum_{i=2}^n \langle x \rangle_i$.

2.2.2 Shamir Secret Sharing

Shamir secret sharing [110] uses polynomials to construct a secret sharing scheme. It uses the fact that t distinct points $((x_1, y_1), \dots, (x_t, y_t))$ uniquely determine a polynomial p of degree at most $t - 1$ with $y_i = p(x_i)$ for $i \in \{1, \dots, t\}$. Defining a polynomial with $p(0) = x$ for a secret value x , while sampling p on at least t points $x_i \neq 0$, makes it possible to reconstruct the polynomial from the samples (by interpolation) and the interpolated polynomial can be evaluated at position 0 to recover the secret. To use this for secret sharing, one can simply define

$$p: \quad l \mapsto x + \sum_{j=1}^{t-1} a_j \cdot l^j$$

and give $p(i)$ to every party $\mathbb{P}_i \in \mathbb{P}$ (e. g., $x_i = i$ and $\langle x \rangle_i = p(i)$). The reconstruction is then the Lagrange interpolation of $p(0)$, e. g.,

$$\text{REC:} \quad (\langle x \rangle_1, \dots, \langle x \rangle_n) \mapsto \begin{cases} \perp & |\{\mathbb{P}_i \in \mathbb{P} \mid \langle x \rangle_i \neq \varepsilon\}| < t \\ \sum_{i \in I} \lambda_{i,I}(0) \cdot \langle x \rangle_i & \text{otherwise} \end{cases}$$

with the Lagrange polynomials

$$\lambda_{i,I}: \quad v \mapsto \prod_{j \in I \setminus \{i\}} \frac{v - \langle x \rangle_j}{i - j}$$

and a set $I \subseteq \{i \mid \mathbb{P}_i \in \mathbb{P}, \langle x \rangle_i \neq \varepsilon\}$ with $|I| = t$. The choice of I does not matter if the shares are correct – e. g., samples of the same polynomial at the respective position $(\langle x \rangle_i$ at position i).

Again, we can simply add up shares to get shares that reconstruct to the sum of the secrets. If, for x and y , the polynomials p_x (with coefficients a_j) and p_y (with coefficients b_j) are used to generate the shares, we get

$$\begin{aligned}
 \text{REC}(\langle x \rangle_1 + \langle y \rangle_1, \dots, \langle x \rangle_n + \langle y \rangle_n) &= \sum_{i \in I} \lambda_{i,I}(0) \cdot (\langle x \rangle_i + \langle y \rangle_i) \\
 &= \sum_{i \in I} \lambda_{i,I}(0) \cdot (p_x(i) + p_y(i)) \\
 &= \sum_{i \in I} \lambda_{i,I}(0) \cdot \left((x + \sum_{j=1}^{t-1} a_j \cdot i^j) + (y + \sum_{j=1}^{t-1} b_j \cdot i^j) \right) \\
 &= \sum_{i \in I} \lambda_{i,I}(0) \cdot (x + y + \sum_{j=1}^{t-1} (a_j + b_j) \cdot i^j),
 \end{aligned}$$

e. g., we interpolate the polynomial $p' = p_x + p_y$ at position 0 and get $p'(0) = x + y$. This illustrates that adding up polynomials implies adding up the coefficients. Multiplying polynomials is the same as convolving their coefficients. While not used here, it is possible to multiply shares directly with multi-party protocols [39, 89]. We use multiplication triples [14] as described in Figure 4.5.

2.3 Encryption Schemes

Encryption is a core part of many systems to achieve security and privacy. It can keep information secret – even when working with data in public, e. g., when working on encrypted data. Additionally, one can also give certain guarantees about the encrypted data. For this work, we only consider asymmetric encryption (or public-key encryption). In contrast to symmetric encryption, where the capability of encrypting values (knowing the private key) also allows one to decrypt values, we need the option for parties to encrypt values without necessarily being able to decrypt them as well. One example is e-voting, where the voters encrypt their values. Here, they should be unable to decrypt other voters' votes. Also, in multi-party computation, where inputs can be sent to other parties in an encrypted form, we do not want any single party to be able to decrypt the inputs of other parties.

Definition 2.6 (Asymmetric Encryption Schemes).

An encryption scheme (or cryptosystem) \mathcal{E} is a tuple $(\text{GEN}, \text{ENC}, \text{DEC})$ of algorithms with

$$\begin{aligned}
 \text{GEN}: & \quad \{1^n\} \rightarrow \mathbb{K} \times \hat{\mathbb{K}} \\
 \text{ENC}: & \quad \mathbb{K} \times \mathbb{X} \times \mathbb{R} \rightarrow \mathbb{Y} \\
 \text{DEC}: & \quad \hat{\mathbb{K}} \times \mathbb{Y} \rightarrow \mathbb{X}
 \end{aligned}$$

and the following property.

Correctness. For all key pairs (k, \hat{k}) generated by GEN we have

$$\forall x \in \mathbb{X} \forall \tilde{x} \in \mathbb{R} : \quad \Pr[\text{DEC}(\hat{k}, \text{ENC}(k, x, \tilde{x})) = x] = 1. \quad \diamond$$

In this work, we call asymmetric encryption schemes simply encryption schemes. $\text{ENC}_k(x, \tilde{x})$ is used instead of $\text{ENC}(k, x, \tilde{x})$ if k is fixed. For $(x) = \text{ENC}_k(x, \tilde{x})$, we call (x) a ciphertext for plaintext x . This notion omits the randomness \tilde{x} used to encrypt x .

Additionally, some encryption schemes are additive or multiplicative homomorphic. This means, we can perform certain operations (addition or multiplication) on ciphertexts, and decrypting the result of these operations yields a plaintext that is the same as if the operations were applied to plaintexts.

Definition 2.7 (Additive Homomorphic Encryption Schemes).

An encryption scheme $\mathcal{E} = (\text{GEN}, \text{ENC}, \text{DEC}, \oplus)$ is an additive homomorphic encryption scheme, if $(\text{GEN}, \text{ENC}, \text{DEC})$ is an encryption scheme and the additive homomorphic property holds for \mathcal{E} .

Additive Homomorphic Property. For all public keys k generated by GEN we have

$$(2.2) \quad \forall x, y \in \mathbb{X} \quad \forall \tilde{x}, \tilde{y} \in \mathbb{R} : \quad \text{ENC}(k, x, \tilde{x}) \oplus \text{ENC}(k, y, \tilde{y}) = \text{ENC}(k, x + y, \tilde{x} + \tilde{y})$$

where \oplus is the addition of ciphertexts and $+$ is the addition of plaintexts or randomness. \diamond

Definition 2.8 (Multiplicative Homomorphic Encryption Schemes).

An encryption scheme $\mathcal{E} = (\text{GEN}, \text{ENC}, \text{DEC}, \oplus, \odot)$ is a multiplicative homomorphic encryption scheme, if $(\text{GEN}, \text{ENC}, \text{DEC}, \oplus)$ is an additive homomorphic encryption scheme and the multiplicative homomorphic property holds for \mathcal{E} .

Multiplicative Homomorphic Property. For all public keys k generated by GEN we have

$$(2.3) \quad \forall x, y \in \mathbb{X} \quad \forall \tilde{x}, \tilde{y} \in \mathbb{R} : \quad \text{ENC}(k, x, \tilde{x}) \odot \text{ENC}(k, y, \tilde{y}) = \text{ENC}(k, x \cdot y, \tilde{x} \cdot \tilde{y})$$

where \odot is the multiplication of ciphertexts and \cdot is the multiplication of plaintexts or randomness. \diamond

The encryption schemes we consider are all CPA-secure, e. g., they are secure against a so-called chosen-plaintext attack. Intuitively, this means that – even when knowing (polynomially) many encryptions of plaintexts – nobody can identify the plaintext for a given ciphertext.

Definition 2.9 (CPA-Secure Encryption Schemes).

An encryption scheme \mathcal{E} is called CPA-secure if

$$\text{Adv}_{\mathcal{E}, A}^{\text{CPA}}(\eta) = \left| 2 \cdot \left(\Pr[\mathfrak{G}_{\mathcal{E}, A}^{\text{CPA}}(\eta) = \text{win}] - \frac{1}{2} \right) \right|$$

is negligible for all PPT adversaries $A = (F, G)$. See Figure 2.1a for $\mathfrak{G}_{\mathcal{E}, A}^{\text{CPA}}$. \diamond

A stronger security notion is CCA-security. Here, an adversary has also access to a decryption oracle that can decrypt any ciphertext except the one for which they have to identify the plaintext for.

Definition 2.10 (CCA-Secure Encryption Schemes).

An encryption scheme \mathcal{E} is called CCA-secure if

$$\text{Adv}_{\mathcal{E},A}^{\text{CCA}}(\eta) = \left| 2 \cdot \left(\Pr[\mathfrak{G}_{\mathcal{E},A}^{\text{CCA}}(\eta) = \text{win}] - \frac{1}{2} \right) \right|$$

is negligible for all PPT adversaries $A = (F, G)$. See Figure 2.1b for $\mathfrak{G}_{\mathcal{E},A}^{\text{CCA}}$. \diamond

Unfortunately, the additive homomorphic encryption schemes we use cannot be CCA-secure. This is shown by the following theorem.

Theorem 2.2 (Homomorphic Encryption Schemes are not CCA-Secure).

Additive homomorphic encryption schemes are not CCA-secure.

Proof. The adversary $A_{\text{CCA}} = (F_{\text{CCA}}, G_{\text{CCA}})$ (see Figure 2.2) always wins the CCA-security game. Let us first consider the value of x (see Figure 2.2b), e. g.,

$$\begin{aligned} x &= \text{DEC}(\hat{k}, \text{ENC}(k, x_b, r) \oplus \text{ENC}(k, 1, s)) \\ &= \text{DEC}(\hat{k}, \text{ENC}(k, x_b + 1, r + s)) \\ &= x_b + 1. \end{aligned}$$

This means, G_{CCA} always returns the right $b' = b$. Additionally, both F_{CCA} and G_{CCA} do not query $\text{ENC}^{-1}(y)$. This implies that F_{CCA} wins with $\text{Adv}_{\mathcal{E},A_{\text{CCA}}}^{\text{CCA}}(\eta) = 1$. \square

Remark. We could get a similar theorem for encryption schemes where ciphertexts can be (only) multiplied by each other.

2.3.1 Verifiable Encryption

Verifiable encryption is a property of an encryption scheme which “allows a verifier to check certain propert[ies] of the content that is encrypted.” [108] Relevant properties for this work are 1. proof that the encrypting party knows the plaintext that was encrypted and 2. proof that a plaintext is the content of a ciphertext.

The first property is needed to prevent parties from choosing their inputs depending on other parties’ inputs. This is a problem – especially in combination with homomorphic encryption schemes. Consider the following example, where all parties want to compute

$$\langle x \rangle = \bigoplus_{i=1}^n \langle x_i \rangle$$

where each party \mathbb{P}_i provides a cipher $\langle x_i \rangle$ (a similar computation is done in the MPC protocol in Figure 4.9). Assume party \mathbb{P}_n is malicious and all other parties already provided their cipher. If the malicious party provides

$$\langle x_n \rangle = \langle y \rangle \ominus \bigoplus_{i=1}^{n-1} \langle x_i \rangle$$

```

1: game  $\mathfrak{G}_{\mathcal{E},A}^{\text{CPA}}(\eta)$ 
2:    $(k, \hat{k}) \xleftarrow{\$} \text{GEN}(1^\eta)$ 
3:    $(x_0, x_1) \xleftarrow{\$} F(1^\eta, k)$ 
4:    $(b, r) \xleftarrow{\$} \{0, 1\} \times \mathbb{R}; y \leftarrow \text{ENC}(k, x_b, r)$ 
5:    $b' \xleftarrow{\$} G(1^\eta, k, y)$ 
6:   if  $b' = b$  then
7:     return win
8:   else
9:     return lose
10:  end if
11: end game

```

(a) Security game for CPA-security.

```

1: game  $\mathfrak{G}_{\mathcal{E},A}^{\text{CCA}}(\eta)$ 
2:    $(k, \hat{k}) \xleftarrow{\$} \text{GEN}(1^\eta); \text{ENC}^{-1}: y \mapsto \text{DEC}(\hat{k}, y)$ 
3:    $(x_0, x_1) \xleftarrow{\$} F(1^\eta, k, \text{ENC}^{-1})$ 
4:    $(b, r) \xleftarrow{\$} \{0, 1\} \times \mathbb{R}; y \leftarrow \text{ENC}(k, x_b, r)$ 
5:    $b' \xleftarrow{\$} G(1^\eta, k, \text{ENC}^{-1}, y)$ 
6:   if  $A$  did not use  $\text{ENC}^{-1}$  for  $y \wedge b' = b$  then
7:     return win
8:   else
9:     return lose
10:  end if
11: end game

```

(b) Security game for CCA-security.

Figure 2.1: Security games for encryption schemes.

```

1: adversary  $F_{\text{CCA}}(1^\eta, k, \text{ENC}^{-1})$ 
2:   return  $(0, 1)$ 
3: end adversary

```

(a) Finder to win the CCA-security game.

```

1: adversary  $G_{\text{CCA}}(1^\eta, k, \text{ENC}^{-1}, y)$ 
2:    $s \xleftarrow{\$} \mathbb{R}$ 
3:    $x \leftarrow \text{ENC}^{-1}(y \oplus \text{ENC}(k, 1, s))$ 
4:   if  $x = 1$  then
5:     return 0
6:   else
7:     return 1
8:   end if
9: end adversary

```

(b) Guesser to win the CCA-security game.

Figure 2.2: CCA-adversary for additive homomorphic encryption schemes.

the value of $\langle x \rangle$ is completely determined by party \mathbb{P}_n :

$$\langle x \rangle = \bigoplus_{i=1}^{n-1} \langle x_i \rangle \oplus \langle y \rangle \ominus \bigoplus_{i=1}^{n-1} \langle x_i \rangle = \langle y \rangle.$$

This means, the malicious party is capable of manipulating the output of this computation and set it to a cipher $\langle y \rangle$ for which they might also know the corresponding plaintext.

The second property is important for verifiable decryption. In the protocol, parties are asked to (partially) decrypt ciphers $\langle x \rangle$. To prevent cheating (parties could just provide any value y and claim that it is the correct decryption – CCA-security implies that distinguishing x and y is hard), a proof that

$$y = \text{DEC}(\hat{k}, \langle x \rangle) = x$$

is needed. A simple way to prove that is providing the plaintext and randomness used to generate $\langle x \rangle$. This is not always possible as the cipher might come from other parties, so the randomness is unknown to the decrypting party.

Both desirable properties (proof of plaintext-knowledge and proof of correct decryption) can be achieved with zero-knowledge proofs. Zero-knowledge proofs are described in Section 2.6 and the important proofs for verifiable encryption can be found in Sections 2.6.8 and 2.6.9.

2.3.2 Threshold Cryptosystems

Threshold cryptosystems allow groups of parties the use of an encryption scheme. If at least t out of n parties work together, they can successfully employ the decryption of the cryptosystem, e. g., together they can decrypt, while a smaller group of parties should not be able to do so. This is also called threshold decryption. This bears similarities to secret sharing schemes (see Section 2.2) where t parties are needed to reconstruct a secret from individual shares, but fewer parties learn nothing about the secret. In fact, threshold cryptosystems can be based on secret sharing by sharing the private key among the n parties.

As we are in the public-key setting, this is what we want from threshold cryptosystems: 1. Anyone should be able to encrypt values and 2. at least t parties can decrypt ciphers together while everyone of the parties learns the decryption – less than t parties or “external” parties should not be able to successfully decrypt ciphers. In the systems in this work we have a partial decryption algorithm

$$\text{PARTIALDEC}: \quad \hat{\mathbb{K}} \times \mathbb{Y} \rightarrow \mathbb{X}'$$

and a reconstruction algorithm (similar to the reconstruction used for secret sharing as described in Section 2.2)

$$\text{DECREC}: \quad (\mathbb{X}' \cup \{\varepsilon\})^n \rightarrow \mathbb{X} \cup \{\perp\}$$

that takes the partial decryptions (or ε if no decryption result is given by this party) and gives the same result as normal decryption, e. g.,

$$\text{DEC}(\hat{k}, \langle x \rangle) = \text{DECREC}(\text{PARTIALDEC}(\hat{k}_1, \langle x \rangle), \dots, \text{PARTIALDEC}(\hat{k}_n, \langle x \rangle))$$

if at least t partial decryptions were given (otherwise \perp signals that the reconstruction is not possible). Such algorithms can be developed for various cryptosystems – i. e., it can be done for Paillier encryption as suggested by Damgård and Jurik [47] or like the suggested RLWE threshold cryptosystem in SPDZ [49].

One might build a threshold cryptosystem from a secret sharing scheme and an encryption scheme as follows: If

$$\hat{k} = \text{REC}(\hat{k}_1, \dots, \hat{k}_n)$$

we could define

$$\begin{aligned} \text{PARTIALDEC:} & & (\hat{k}_i, \langle x \rangle) & \mapsto T_{\langle x \rangle}(\text{DEC}(\hat{k}_i, \langle x \rangle)) \\ \text{DECREC:} & & (d_1, \dots, d_n) & \mapsto T_{\langle x \rangle}^{-1}(\text{REC}(d_1, \dots, d_n)) \end{aligned}$$

with a bijective function $T_{\langle x \rangle}$ that depends on the ciphertext. For ElGamal encryption (see Section 2.3.3) one could use

$$\begin{aligned} T_{(c_0, c_1)} &: & x & \mapsto x \cdot c_1^{-1} \\ T_{(c_0, c_1)}^{-1} &: & x & \mapsto x \cdot c_1. \end{aligned}$$

The advantage of using PARTIALDEC based on DEC is that the zero-knowledge proofs of the underlying cryptosystems can simply be reused – we require verifiable encryption and decryption for the threshold cryptosystems. The proof for encryption can be used as-is and the proof for decryption for the threshold cryptosystem is the combination of decryption proofs for each party (application of $T_{\langle x \rangle}$, $T_{\langle x \rangle}^{-1}$, and DECREC can be done by anyone).

2.3.3 ElGamal Encryption Scheme

ElGamal encryption [51] is defined as

$$\text{ENC:} \quad G \times G \times \mathbb{Z}_q \rightarrow G^2, \quad (k, x, r) \mapsto (g^r, x \cdot k^r)$$

and decryption as

$$\text{DEC:} \quad \mathbb{Z}_q \times G^2 \rightarrow G, \quad (\hat{k}, (c_0, c_1)) \mapsto c_0^{-\hat{k}} \cdot c_1$$

for a group (G, \cdot) of order q and a generator $g \in G$. The group operation is \cdot , e. g., we use a multiplicative notation. Additionally, the following is true for the public key $k \in G$ and the private key $\hat{k} \in \mathbb{Z}_q$:

$$k = g^{\hat{k}}.$$

Addition can be done as follows:

$$\oplus : \quad ((a_0, a_1), (b_0, b_1)) \mapsto (a_0 \cdot b_0, a_1 \cdot b_1),$$

e. g., with component-wise multiplication.

Theorem 2.3 (ElGamal Encryption is CPA-Secure).

The ElGamal encryption scheme is an additive homomorphic CPA-secure encryption scheme.

Proof. For proofs, we refer the reader to the literature [42, 51, 113]. □

Remark 1. Note that the security of ElGamal encryption is based on the hardness of the Diffie–Hellman problem [50].

Remark 2. Examples for groups that can be used for ElGamal encryption are the subgroup of quadratic residues (with prime order q) of \mathbb{Z}_p^* (with prime p) [113] or elliptic curves [76].

Remark 3. The ciphertext operation \oplus yields multiplications of plaintexts in G . \mathbb{Z}_q can be used as a plaintext space instead by mapping $x \in \mathbb{Z}_q$ to $g^x \in G$. The ciphertext operation is then equivalent to an addition in plaintext space (now \mathbb{Z}_q). To decrypt, one has to compute the discrete logarithm to get a plaintext from a group element. In general, this is not feasible, but if the results of computations are expected to lay in a small range, one can compute a mapping between all expected plaintexts and group elements.

2.3.4 Paillier Encryption Scheme

The Paillier cryptosystem [96] is constructed quite differently. For two primes p and q , one can set $m = p \cdot q$ and use $\hat{k} = \text{lcm}(p - 1, q - 1)$ as the private key. The public key k can then be a random element of $\mathbb{Z}_{m^2}^*$ (m is also part of the public parameters of the encryption scheme). Encryption and decryption are then

$$\text{ENC: } \mathbb{Z}_{m^2}^* \times \mathbb{Z}_m \times \mathbb{Z}_m^* \rightarrow \mathbb{Z}_{m^2}^*, \quad (k, x, r) \mapsto k^x \cdot r^m \text{ mod } m^2$$

and

$$\text{DEC: } \mathbb{Z} \times \mathbb{Z}_{m^2}^* \rightarrow \mathbb{Z}_m, \quad (\hat{k}, c) \mapsto \frac{L(c^{\hat{k}} \text{ mod } m^2)}{L(k^{\hat{k}} \text{ mod } m^2)} \text{ mod } m$$

with $L: x \rightarrow \frac{x-1}{m}$. Addition of ciphertexts

$$\oplus : \quad (a, b) \mapsto a \cdot b \text{ mod } m^2$$

is simply a multiplication modulo m^2 .

Theorem 2.4 (Paillier Encryption is CPA-Secure).

The Paillier encryption scheme is an additive homomorphic CPA-secure encryption scheme.

Proof. For proofs, we refer the reader to the literature [96]. □

Remark. Note that the security of Paillier encryption is based on the hardness of the composite residuosity class problem [96].

2.3.5 RLWE Encryption

In this section we will briefly sketch a different kind of encryption scheme. Encryption schemes that are based on learning with errors (LWE) [106] or ring learning with errors (RLWE) [88] are somewhat homomorphic encryption schemes (see also Section 2.7.2). This means, in addition to be able to perform additions on ciphertexts, we can also multiply them (and the decryption of the result would be the product of the plaintexts of ciphertexts we multiplied). As this is only possible for a limited number of operations (after too many additions or multiplications, the noise grows too much to recover the intended plaintext), these schemes are not fully homomorphic encryption schemes. While this can be remedied to achieve fully homomorphic encryption, i. e., with so-called bootstrapping, we do not use FHE here. As schemes based on LWE (or RLWE) are quite complex, we refer to the literature for more details and a more complete overview [99].

Here, we describe one variant of the BGV scheme [27] used in SPDZ [49] that uses aspects of some other RLWE encryption schemes [29, 59, 111]. Other schemes like the GSW scheme [28, 60] could be used here as well. Other variants of SPDZ use a different encryption scheme [48].

For this RLWE scheme, the ring $R_q = \mathbb{Z}_q[x]/f(x)$ is used. $\mathbb{Z}_q[x]$ are the polynomials with coefficients in \mathbb{Z}_q (e. g., coefficients are taken modulo q), elements of R_q are polynomials with such coefficients, and operations (addition and multiplication) in R_q are done modulo $f(x)$ (i. e., Euclidean division is done on the result of the operation to get the remainder).

In SPDZ, plaintexts are assumed to be field elements of \mathbb{F}_{p^l} for some integer l and prime p . With a SIMD [53] factor s , elements of $(\mathbb{F}_{p^l})^s$ are mapped to R_q , which allows to add (or multiply) s plaintexts with one operations in R_q . Oversimplifying – the elements of R_q are polynomials of degree less than m as we compute modulo $f(x)$. This means, there are m coefficients that are added or multiplied³ when adding or multiplying polynomials. These coefficients can then be mapped to plaintexts.

With the private key \hat{k} and the public key $k = (a, b) = (a, a \cdot \hat{k} + p \cdot e)$, encryption and decryption are described as follows:

$$\begin{aligned} \text{ENC: } & R_q^2 \times R_q \times R_q^3 \rightarrow R_q^3, \quad ((a, b), x, (u, v, w)) \mapsto (b \cdot v + p \cdot w + x, a \cdot v + p \cdot u, 0) \\ \text{DEC: } & R_q \times R_q^3 \rightarrow R_q, \quad (\hat{k}, (c_0, c_1, c_2)) \mapsto c_0 - \hat{k} \cdot c_1 - \hat{k} \cdot \hat{k} \cdot c_2 \end{aligned}$$

where \hat{k}, e, u, v, w are all sampled according to a discrete Gaussian distribution and a is sampled uniformly at random. For the exact choice of parameters⁴ (i. e., the parameters of the Gaussian distribution, q or $f(x)$) see the calculations by Damgård et al. [49]. The ciphertext addition is defined as

$$\oplus : \quad ((a_0, a_1, a_2), (b_0, b_1, b_2)) \mapsto (a_0 + b_0, a_1 + b_1, a_2 + b_2)$$

and multiplication as

$$\odot : \quad ((a_0, a_1, 0), (b_0, b_1, 0)) \mapsto (a_0 \cdot b_0, a_1 \cdot b_0 + a_0 \cdot b_1, -a_1 \cdot b_1)$$

³As multiplying polynomials is equal to convolving their coefficients, the coefficients are multiplied in the Fourier domain.

⁴Note that we use Shamir secret sharing which requires more ciphertext additions than full-threshold secret sharing, so the parameters might need some adjustment.

where we can only multiply ciphertexts with a zero-entry at the third place.

Theorem 2.5 (RLWE Encryption is CPA-Secure).

The above RLWE encryption scheme is a multiplicative homomorphic CPA-secure encryption scheme.

Proof. For the proof, see the SPDZ paper [49]. □

Remark 1. Note that the security of RLWE encryption is based on the hardness of the RLWE problem [88].

Remark 2. Ciphertexts can be added up only a limited number of times before the noise grows too big and decryption leads to wrong results, e. g., the result is no longer the sum of the plaintexts. Similarly, multiplication can be done only once with only a limited number of additions before and after the multiplication. Damgård et al. computed the parameters of the encryption scheme that are needed for it to be used with SPDZ [49] and a similar calculation should be done before implementing our protocol.

2.4 Commitment Schemes

Just like encryption and zero-knowledge proofs,⁵ commitment schemes are an essential part of many protocols. Especially in multi-party protocols, they remove one possibility of cheating. A commitment should reveal no information about the value that is committed to (similar to encryption) while it should be also hard to find two values with the same commitments (encryptions of two distinct values are always distinct to allow for decryption). This can be used in many protocols as follows: First, parties commit to their values. After all commitments are sent to their recipient, the decommitments (the values and randomness to produce the commitments) are sent as well. Without commitment schemes, a party could change the value they should reveal after seeing other values. The properties of commitment schemes prevent this and, therefore, are used as primitives in many protocols. Just like encryption schemes (see Section 2.3) or digital signatures, the commitment schemes we present here can be considered as public-key primitives.

Definition 2.11 (Commitment Schemes).

A commitment scheme C is a tuple (GEN, COM) with

$$\begin{aligned} \text{GEN} : & \quad \{1^n\} \rightarrow \check{\mathbb{K}} \\ \text{COM} : & \quad \check{\mathbb{K}} \times \mathbb{X} \times \mathbb{R} \rightarrow \mathbb{C}. \end{aligned}$$

We also require a verification predicate

$$\text{VERIFYCOM} : \quad \check{\mathbb{K}} \times \mathbb{C} \times \mathbb{X} \times \mathbb{R} \rightarrow \{\text{false}, \text{true}\}$$

⁵In fact, commitments are often used in zero-knowledge proofs.

```

1: game  $\mathcal{G}_{C,A}^{\text{hiding}}(\eta)$ 
2:    $\check{k} \xleftarrow{\$} \text{GEN}(1^\eta)$ 
3:    $(x_0, x_1) \xleftarrow{\$} F(1^\eta, \check{k})$ 
4:    $(b, r) \xleftarrow{\$} \{0, 1\} \times \mathbb{R}; c \leftarrow \text{COM}(\check{k}, x_b, r)$ 
5:    $b' \xleftarrow{\$} G(1^\eta, \check{k}, c)$ 
6:   if  $b' = b$  then
7:     return win
8:   else
9:     return lose
10:  end if
11: end game

```

(a) Security game for the hiding property of commitment schemes.

```

1: game  $\mathcal{G}_{C,A}^{\text{binding}}(\eta)$ 
2:    $\check{k} \xleftarrow{\$} \text{GEN}(1^\eta)$ 
3:    $(c, x_0, r_0, x_1, r_1) \xleftarrow{\$} A(1^\eta, \check{k})$ 
4:   if  $x_0 \neq x_1 \wedge \text{COM}(\check{k}, x_0, r_0) = c \wedge \text{COM}(\check{k}, x_1, r_1) = c$  then
5:     return win
6:   else
7:     return lose
8:   end if
9: end game

```

(b) Security game for binding property of commitment schemes.

Figure 2.3: Security games for commitment schemes.

for the commitment scheme with

$$\forall x \in \mathbb{X} \forall \tilde{x} \in \mathbb{R} : \quad \Pr[\text{VERIFYCOM}(\check{k}, \text{COM}(\check{k}, x, \tilde{x}), x, \tilde{x}) = \text{true}] = 1$$

for all \check{k} generated by GEN. We usually use the canonical

$$\text{VERIFYCOM} : \quad (\check{k}, \llbracket x \rrbracket, x, \tilde{x}) \mapsto \begin{cases} \text{true} & \llbracket x \rrbracket = \text{COM}(\check{k}, x, \tilde{x}) \\ \text{false} & \llbracket x \rrbracket \neq \text{COM}(\check{k}, x, \tilde{x}) \end{cases}.$$

Additionally, a commitment scheme has to be hiding and binding. Different “strengths” for both properties are defined in Definitions 2.12 to 2.15. Furthermore, we require all commitment schemes that we use to be additive homomorphic.

Additive Homomorphic Property. For all parameters \check{k} generated by GEN we have

$$\forall x, y \in \mathbb{X} \forall \tilde{x}, \tilde{y} \in \mathbb{R} : \quad \text{COM}(\check{k}, x, \tilde{x}) \boxplus \text{COM}(\check{k}, y, \tilde{y}) = \text{COM}_{\check{k}}(x + y, \tilde{x} + \tilde{y})$$

where \boxplus is the addition of commitments and $+$ is the addition of plaintexts or randomness. \diamond

$\text{COM}_{\check{k}}(x, \tilde{x})$ is used instead of $\text{COM}(\check{k}, x, \tilde{x})$ if \check{k} is fixed. For $\llbracket x \rrbracket = \text{COM}_{\check{k}}(x, \tilde{x})$, we call $\llbracket x \rrbracket$ the commitment for x (\tilde{x} is omitted in this notation, similarly to the notation for ciphertexts) and (x, \tilde{x}) the decommitment for $\llbracket x \rrbracket$.

Commitment schemes should be hiding and binding. This means, it should be hard to figure out which value was committed to and hard to find two values that commit to the same commitment. The hiding and binding properties can be defined for a commitment scheme C and an adversary A in terms of the security games $\mathfrak{G}_{C,A}^{\text{hiding}}$ and $\mathfrak{G}_{C,A}^{\text{binding}}$ (see Figure 2.3).

In the hiding-game (Figure 2.3a), the adversary $A = (F, G)$ can choose two values. A commitment is generated for one of the values and the adversary has to figure out which value was committed to using only the commitment (and previous knowledge from the find phase).

In the binding-game (Figure 2.3b), the adversary A tries to find two values (each with corresponding randomness) and a commitment, such that the values commit to the commitment.

Definition 2.12 (Computationally Hiding Commitment Schemes).

A commitment scheme C is computationally hiding if

$$\text{Adv}_{C,A}^{\text{hiding}}(\eta) = \left| 2 \cdot \left(\Pr[\mathfrak{G}_{C,A}^{\text{hiding}}(\eta) = \text{win}] - \frac{1}{2} \right) \right|$$

is negligible for all PPT adversaries $A = (F, G)$. See Figure 2.3a for $\mathfrak{G}_{C,A}^{\text{hiding}}$. ◇

Definition 2.13 (Computationally Binding Commitment Schemes).

A commitment scheme C is computationally binding if

$$\text{Adv}_{C,A}^{\text{binding}}(\eta) = \Pr[\mathfrak{G}_{C,A}^{\text{binding}}(\eta) = \text{win}]$$

is negligible for all PPT adversaries A . See Figure 2.3b for $\mathfrak{G}_{C,A}^{\text{binding}}$. ◇

Definition 2.14 (Perfectly Hiding Commitment Schemes).

A commitment scheme C is perfectly hiding if

$$\forall x, x' \in \mathbb{X} \forall c \in \mathbb{C} : \Pr[\text{COMMIT}(\check{k}, x) = c] = \Pr[\text{COMMIT}(\check{k}, x') = c]$$

for all parameters $\check{k} \in \check{\mathbb{K}}$. See Figure 2.4b for the COMMIT procedure. ◇

Definition 2.15 (Perfectly Binding Commitment Schemes).

A commitment scheme C is perfectly binding if $\text{Adv}_{C,A}^{\text{binding}}(A) = 0$ for all adversaries A . ◇

Remark. Opposed to computationally binding commitment schemes, the adversary is no longer computationally bounded.

<pre> 1: procedure ENCRYPT(k, x) 2: $r \xleftarrow{\\$} \mathbb{R}$ 3: $c \leftarrow \text{ENC}(k, x, r)$ 4: return c 5: end procedure </pre> <p style="text-align: center;">(a) Encryption procedure.</p>	<pre> 1: procedure COMMIT(\check{k}, x) 2: $r \xleftarrow{\\$} \mathbb{R}$ 3: $c \leftarrow \text{COM}(\check{k}, x, r)$ 4: return c 5: end procedure </pre> <p style="text-align: center;">(b) Commitment procedure.</p>
---	--

Figure 2.4: Canonical encryption and commitment procedures.

2.4.1 Pedersen Commitment Scheme

The Pedersen commitment scheme [98] is defined for a group (G, \cdot) of order q . GEN randomly picks two generators g and h and the commitment procedure is the following:

$$\text{COM}: \quad G^2 \times \mathbb{Z}_q \times \mathbb{Z}_q \rightarrow G, \quad ((g, h), x, r) \mapsto g^x \cdot h^r.$$

Commitments can be added up with

$$\boxplus: \quad (a, b) \mapsto a \cdot b,$$

e. g., with a single group operation.

Theorem 2.6 (Pedersen Commitments are Hiding and Binding).

The Pedersen commitment scheme is a perfectly hiding and computationally binding commitment scheme.

Proof. For the proof, we refer the reader to the literature [98]. □

Remark 1. Note that the binding property is based on the hardness of computing discrete logarithms [98].

Remark 2. One could use, for example, the same groups as for ElGamal encryption (see Section 2.3.3).

2.4.2 Encryption Schemes as Commitment Schemes

Let $\mathcal{E} = (\text{GEN}_{\mathcal{E}}, \text{ENC}, \text{DEC})$ be a CPA-secure encryption scheme. We can build a commitment scheme $C_{\mathcal{E}} = (\text{GEN}'_{\mathcal{E}}, \text{ENC})$ where $\text{GEN}'_{\mathcal{E}}$ is $\text{GEN}_{\mathcal{E}}$, but the private key is discarded and only the public key returned, e. g., $\check{k} = k$.

Theorem 2.7 (Encryption Schemes as Commitment Schemes).

The commitment scheme $C_{\mathcal{E}}$ defined by a CPA-secure encryption scheme \mathcal{E} is a computationally hiding and perfectly binding commitment scheme.

Proof. The binding property follows from the correctness of the encryption scheme: Every ciphertext can be correctly decrypted, which means, the ciphertexts for different plaintexts can never be the same (for any randomness). This implies that no adversary A can find two different plaintexts with the same commitments (e. g., which encrypt to the same ciphertext) and $\text{Adv}_{\mathcal{C}_{\mathcal{E},A}}^{\text{binding}} = 0$.

The hiding property follows from the fact that $\mathfrak{G}_{\mathcal{C}_{\mathcal{E},A}}^{\text{hiding}}$ and $\mathfrak{G}_{\mathcal{E},A}^{\text{CPA}}$ are identical (up to syntactical differences). This means, $\text{Adv}_{\mathcal{C}_{\mathcal{E},A}}^{\text{hiding}}(\eta) = \text{Adv}_{\mathcal{E},A}^{\text{CPA}}(\eta)$, which is negligible for all PPT adversaries A by assumption. \square

Remark 1. The additive homomorphic property of commitment schemes follows from the additive homomorphic property of the encryption scheme.

Remark 2. Here, it is quite obvious to see that some commitment schemes can have a “backdoor”. If anyone gets hold of the (discarded) private key, all commitments can be decrypted. To get a hiding commitment scheme, nobody should have this private key. On the other hand, other commitment schemes can have a backdoor that enables someone to decommit to any plaintext (i. e., in the Pedersen commitment scheme [98]).

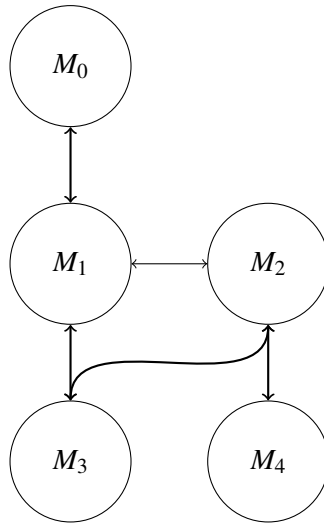
2.5 Universal Composability Framework

This section outlines a simple notion of interactive Turing machines and a (simplified) version of a framework for universal composability (UC) by Canetti [30]. A more complex version of the UC-framework that also includes more capable machine models exists as well [30] but is not required in this work.

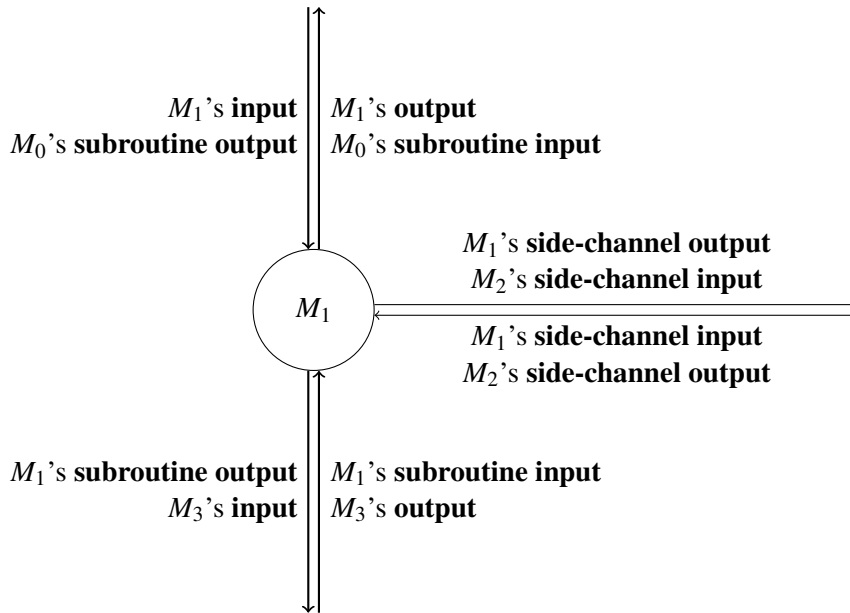
2.5.1 Interactive Turing Machines

For interactive protocols, we assume interactive Turing machines (ITMs) similar to the ones used to describe the simplified UC-framework [30]. Exact details about the Turing machine are left out here. These machines can interact with each other using different input and output tapes. There are six distinct communication tapes: 1. the **input** tape, 2. the **output** tape, 3. the **subroutine input** tape, 4. the **subroutine output** tape, 5. the **side-channel input** tape, and 6. the **side-channel output** tape. We assume communication tapes are always connected in a pair-wise way:

- An **input** tape is another machine’s **subroutine output** tape.
- An **output** tape is another machine’s **subroutine input** tape.
- A **subroutine input** tape is another machine’s **output** tape.
- A **subroutine output** tape is another machine’s **input** tape.
- A **side-channel input** tape is another machine’s **side-channel output** tape.
- A **side-channel output** tape is another machine’s **side-channel input** tape.



(a) Example of connected interactive Turing machines.



(b) Connections of M_1 .

Figure 2.5: Example for connectivity of interactive Turing machines.

This is visualized by Figure 2.5. Here, bold wires (\longrightarrow) are “normal” communication tapes and non-bold (\longrightarrow) wires are **side-channel** tapes. Connections for **input** and **output** tapes are drawn at the top of machines, **side-channel** tapes are on the sides, and **subroutine** tapes are at the bottom.

Additionally, machines have an identity and a program (their “code”). The communication capability of a machine can be formalized as a communication set. This allows us to define machines as caller or subroutine (in an intuitive way). See Canetti’s paper [30] for more details and the full UC-framework that models more capable machines, which is not strictly necessary here.

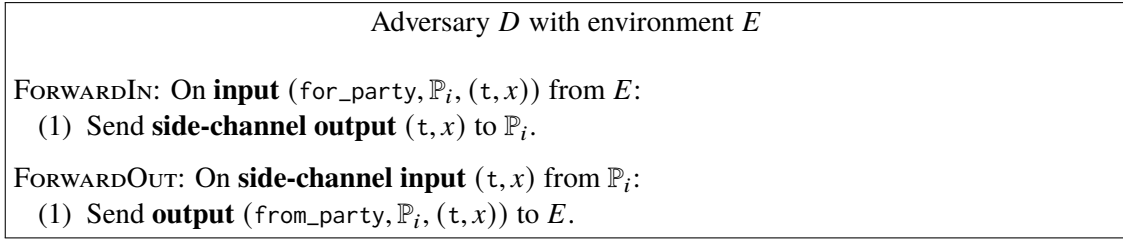


Figure 2.6: Dummy adversary.

2.5.2 Protocols and Functionalities

Protocols can be modeled as sets of interactive machines. Here, the communication set of the machines has to be considered to form a valid protocol (i. e., a machine should not be caller of a machine that is not part of the protocol). All machines that are subroutines of machines that are not part of the protocol are called main machines. Intuitively, this allows us to use the protocol in some other protocol.

(Ideal) functionalities can be considered as machines that perfectly implement their specification. This means, their (output) behavior and capabilities of adversaries to interact with the functionality (or manipulate it) are exactly what is described when defining the functionality. Ideally, one would then want to design a functionality to always output the correct intended result, but this is not possible as described in the next section.

Additionally, we can define the ideal protocol Π^{ideal} for a functionality \mathcal{F} by adding dummy parties for each caller of \mathcal{F} . Dummy parties could be defined similarly to the dummy adversary (Figure 2.6), but they forward their **input** as **side-channel output** etc. The reason for this is that a multi-party protocol could (formally) easily be distinguishable from a functionality, because the functionality is only a single machine, whereas an environment interacting with the protocol would interact with multiple machines.

2.5.3 Emulation

The security of protocols can be captured by the notion of emulation. If a protocol emulates another one (see Definition 2.16), no environment is able to (successfully) distinguish them. This means, they are just “as secure” because their outputs are similar enough and their side effects give not enough information to distinguish them.

As it is harder to show that protocols are indistinguishable for all adversaries and environments, we introduce the dummy adversary in Figure 2.6.

To do simpler modelling of protocols, we use wrappers to model corruption on top of protocols where everyone is assumed to be honest, e. g., we consider a protocol Π^{wrap} that can be corrupted instead of the original protocol Π . Many types of corruption and other properties can be modeled on top of protocols, but we limit ourselves to static corruption wrappers in this work. Note that the UC-framework models only point-to-point communication between machines. Additionally, the messages are sent through the adversary, e. g., **side-channel** messages are sent to the adversary and they send it to the intended recipient. This models a network adversary. We neglect this as all

messages in our protocol are managed by a separate functionality (a bulletin board $\mathcal{F}_{\text{bulletin}}$ that is functionally similar to a broadcast channel) and the adversary gets a copy of all messages through that functionality and can potentially delay messages for other parties.

Here, it is assumed that the first step of a protocol run is an initialization and the first step of that is corrupting parties. The functionality will ask the adversary to corrupt parties and the adversary will answer with a set of corrupted parties. The wrapper will do the same and the following after receiving the set: It sends a special corruption message to all parties in this set. This allows the adversary to send special messages later to control the parties (e. g., let them send arbitrary messages or send information back to the adversary).

Definition 2.16 (UC-Framework: Emulation).

A protocol Π UC-emulates another protocol Π' if

$$\forall A \exists S \forall E : \quad \left| \Pr[(E \parallel A \parallel \Pi)(1^n) = \text{real}] - \Pr[(E \parallel S \parallel \Pi')(1^n) = \text{real}] \right|$$

is negligible with PPT A , S , and E . A is an adversary on Π and S is an adversary on Π' , while E is an environment that is supposed to distinguish the protocol runs. \diamond

Remark 1. S is called a simulator and is subject to some restrictions. Most noteworthy is that S cannot rewind. Especially, no messages sent to E can be “taken back”.

Remark 2. It is sufficient to show that the probabilities are indistinguishable for the dummy adversary D instead of all adversaries A [30]. real and ideal are used as we are mainly concerned with UC-realization of ideal functionalities (see Definition 2.17).

This corresponds to the definition of UC-emulation by Canetti in the “restricted model” [30]. In the following, we describe the definition from the same work regarding the ability of protocols to realize ideal functionalities.

Definition 2.17 (UC-Framework: Realizing Ideal Functionalities).

A protocol Π UC-realizes a functionality \mathcal{F} if Π UC-emulates Π^{ideal} . \diamond

Remark 1. If we want to model corruption, we require that Π^{wrap} UC-emulates Π^{ideal} .

Remark 2. Generally, an ideal functionality cannot only model the correct output as we have to be able to simulate it (especially in presence of malicious parties).

A visual example for a protocol and a functionality that have to be distinguished by an environment can be found in Figure 4.7. We call the setting, where the environment E interacts with a protocol Π and an adversary A , real world. If E interacts with a functionality \mathcal{F} (in form of the corresponding ideal protocol Π^{ideal}) and a simulator S , we call it ideal world.

2.5.4 Composability

For composability, we want to replace a sub-protocol in a larger protocol. With ideal functionalities in mind, this is exactly what we want to achieve in many cases: We model a protocol Π that uses an ideal functionality \mathcal{G} and we prove that Π realizes a functionality \mathcal{F} . With composability, we can replace \mathcal{G} by a protocol that realizes it (e. g., a protocol Π'). This allows us to implement \mathcal{F} as the new protocol consisting of interactive machines that implement a multi-party protocol instead of an abstract functionality that acts as a trusted party. Additionally, the new protocol is just as secure as \mathcal{F} .

Formally, we have the following situation: We have protocols Π , Π' , and Π'' . Π' is a subset of Π (but on its own it is still a valid protocol), no machine of Π'' appears in Π when removing machines from Π' , and Π'' is “compatible” with Π' . The last point means, we can map main machines of Π' to the ones of Π'' and all callers that are not part of Π'' (external callers) are also external callers of Π' . This means, we can replace Π' with Π'' and get $\Pi^{\Pi' \rightarrow \Pi''}$. The following theorem allows us to use composition.

Theorem 2.8 (UC-Framework: Composition).

For Π , Π' , Π'' as described above, if Π' UC-emulates Π'' then $\Pi^{\Pi' \rightarrow \Pi''}$ UC-emulates Π .

From this theorem, the respective case for functionalities can be deduced.

Corollary 2.9 (UC-Framework: Composition of Ideal Functionalities).

If protocol Π UC-realizes functionality \mathcal{F} , protocol Π' UC-realizes functionality \mathcal{G} , and Π uses \mathcal{G} , we have that $\Pi^{\mathcal{G} \rightarrow \Pi'}$ UC-realizes \mathcal{F} .

As before, these are theorems that are for the restricted model, but the corresponding theorems exists also in the full universal composability model [30]. Finally, we call (composed) protocols \mathcal{F} -hybrid if we allow the sub-protocols (that might be introduced by replacing functionalities) to share \mathcal{F} with the protocol. This sharing does not carry over to different runs (for example through parallel execution) of the protocol.

2.6 Zero-Knowledge

Zero-knowledge enables many protocols to keep vital information secret while giving certain guarantees. For example, in our protocols, it allows us to make statements about accountability: Parties have to prove that they follow the protocol at certain steps (i. e., by proving that the data they send was computed in a certain way). Honest parties (that follow the protocol) can easily give these proofs, while malicious (misbehaving) parties cannot. Usually, the “proofs” are not proofs in the sense that they guarantee that a certain fact is actually true, but it can be guaranteed that the statement is true with a certain (very high) probability. The “zero-knowledge” part of “zero-knowledge proofs” implies that nobody observing the proof (including the person that something is proven to) learns any information about the secret for which a fact is proven.

2.6.1 Interactive Proof System

Interactive proof systems are protocols where one party (the prover) tries to prove to another party (the verifier) that a statement is true for a value e (e. g., proof that $e \in \mathcal{L}$ for a suitable language \mathcal{L}). The prover usually does not prove that the statement is true in all cases. Rather, they prove that the statement is true with a certain probability. The prover and verifier are modeled as interactive Turing machines (see Section 2.5.1 for a more detailed description). A general depiction of an interactive proof system is given in Figure 2.7a.

Definition 2.18 (Interactive Proof Systems).

$(\mathcal{P}, \mathcal{V})$ is an interactive proof system (IPS) for a language \mathcal{L} if \mathcal{P} and \mathcal{V} are connected interactive Turing machines with the following properties.

Completeness.

$$\forall e \in \mathcal{L} : \quad \Pr[(\mathcal{P} \parallel \mathcal{V})(e) = \text{accept}] \geq \tau_{\text{completeness}}$$

Soundness.

$$\forall e \notin \mathcal{L} \forall \mathcal{P}' : \quad \Pr[(\mathcal{P}' \parallel \mathcal{V})(e) = \text{accept}] \leq \tau_{\text{soundness}}$$

Note that the runtime of \mathcal{V} has to be bounded by a polynomial, while \mathcal{P} and \mathcal{P}' simply require bounded runtime. We call $\tau_{\text{completeness}}$ the completeness bound of the system and $\tau_{\text{soundness}}$ the soundness bound. We require $0 \leq \tau_{\text{soundness}} < \tau_{\text{completeness}} \leq 1$. \diamond

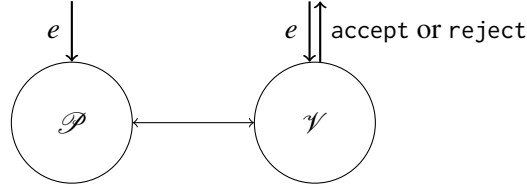
Remark. An “improved” proof system can be constructed by iterating the proof and letting the new verifier output accept iff. \mathcal{V} returned accept in all iterations. Obviously, this increases the completeness bound and lowers the soundness bound compared to the original proof system.

Corollary 2.10 (IPSs with Better Completeness and Soundness Bounds).

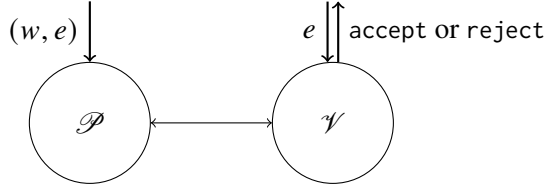
For every IPS $(\mathcal{P}, \mathcal{V})$ exists an IPS with overwhelming completeness bound and negligible soundness bound.

Proof (Sketch). This follows from the previous remark and using a polynomial number of iterations. \square

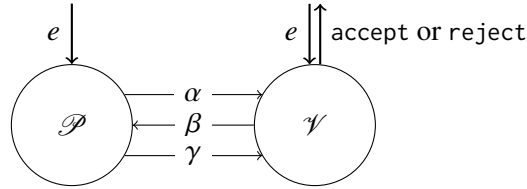
Remark. The runtime of the new verifier is still polynomial.



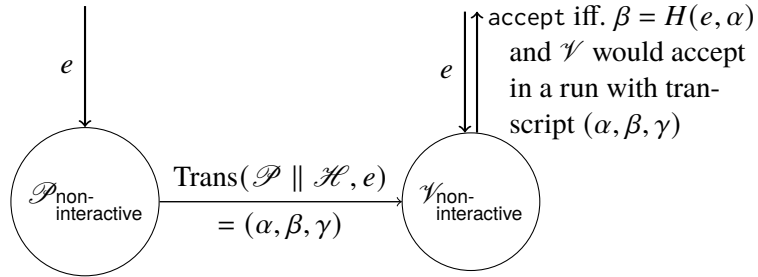
(a) Depiction of $(\mathcal{P} \parallel \mathcal{V})(e)$.



(b) Depiction of $(\mathcal{P}^w \parallel \mathcal{V})(e)$.



(c) Depiction of $(\mathcal{P} \parallel \mathcal{V})(e)$ for a Σ -protocol.



(d) Depiction of $(\mathcal{P}^{\text{non-interactive}} \parallel \mathcal{V}^{\text{non-interactive}})(e)$ – the non-interactive ZK proof constructed from the Σ -protocol $(\mathcal{P}, \mathcal{V})$ with hash function H and hashing verifier \mathcal{H} .

Figure 2.7: Interactive proof systems.

2.6.2 Zero-Knowledge Proofs

An important property of IPSs – more specifically of the prover – is zero-knowledge (ZK). Intuitively, this means that no verifier – even malicious ones that try to gain information – learns anything when interacting with a prover.

Definition 2.19 (Computational Zero-Knowledge).

Let $(\mathcal{P}, \mathcal{V})$ be an IPS for \mathcal{L} . \mathcal{P} is called computational zero-knowledge if

$$\forall \mathcal{V}' \exists S : \quad \{(\mathcal{P} \parallel \mathcal{V}')(e)\}_{e \in \mathcal{L}} \text{ and } \{S(e)\}_{e \in \mathcal{L}}$$

are computationally indistinguishable. Note that \mathcal{V}' is a PPT ITM and S is PPT. ◇

Definition 2.20 (Perfect Zero-Knowledge).

Let $(\mathcal{P}, \mathcal{V})$ be an IPS for \mathcal{L} . \mathcal{P} is called perfect zero-knowledge if

$$\forall \mathcal{V}' \exists S \forall e \in \mathcal{L} : \quad \Pr[S(e) = \perp] \leq \frac{1}{2}$$

and

$$\forall y : \Pr[(\mathcal{P} \parallel \mathcal{V}')(e) = y] = \Pr[S(e) = y \mid S(e) \neq \perp].$$

Note that \mathcal{V}' is a PPT ITM and S is PPT. ◇

2.6.3 Proofs for Languages in NP

For some of the following definitions in the area of zero-knowledge, we need the notion of NP-languages. These are languages that are decidable with a PPT Turing machine – more formally:

Definition 2.21 (NP-Languages).

A language \mathcal{L} is in (the class of) NP if there is a PPT TM M such that $e \in \mathcal{L}$ iff. there is a run in which M accepts (e. g., $M(e) = \text{accept}$). ◇

We can transform this canonical definition to a more suitable one.

Corollary 2.11 (NP-Languages).

A language \mathcal{L} is in (the class of) NP if there is a relation \mathcal{R} that is decidable in polynomial time with a deterministic Turing machine such that

$$\mathcal{L} = \{e \mid \exists w : (w, e) \in \mathcal{R}\}.$$

We call w the witness for \mathcal{L} .

Proof. If we use the same Turing machine M as in Definition 2.21, we define

$$\mathcal{R} = \{(\alpha, e) \mid M^\alpha(e) = \text{accept}\}$$

which means, in Corollary 2.11, we claim that

$$\mathcal{L} = \{e \mid \exists \alpha : M^\alpha(e) = \text{accept}\}.$$

This is equivalent to Definition 2.21. □

Theorem 2.12 (NP-Languages have Zero-Knowledge Proofs).

All languages in NP can be proven in zero-knowledge.

Proof. This was proven by Goldreich et al. [65]. □

Theorem 2.13 (Proving NP-Languages in Polynomial Time).

Let \mathcal{R} be a relation for the NP-language \mathcal{L} . Let $(\mathcal{P}, \mathcal{V})$ be an IPS for \mathcal{L} . $(\mathcal{P}', \mathcal{V})$ is an IPS for \mathcal{L} where \mathcal{P}' is a PPT ITM that gets the witness for \mathcal{R} as private input.

Proof (Sketch). If the witness is given to the prover in Theorem 2.12, all other steps can be done in polynomial time. \square

Remark. This makes it possible to use IPSs in practice as the runtime of the prover is no longer (potentially) superpolynomial.

We use \mathcal{P}^w to denote that the prover \mathcal{P} got the witness w as private input. This is depicted in Figure 2.7b.

2.6.4 Zero-Knowledge Proofs of Knowledge

Definition 2.22 (Proofs of Knowledge).

Let \mathcal{R} be a relation for the language \mathcal{L} . An IPS $(\mathcal{P}, \mathcal{V})$ for \mathcal{L} is called proof of knowledge with knowledge error $\kappa: \mathbb{N} \rightarrow [0, 1)$ if the following holds true.

Non-Triviality.

$$\forall e \in \mathcal{L} : \Pr[(\mathcal{P} \parallel \mathcal{V})(e) = \text{accept}] = 1$$

Validity.

$$\exists p \exists E \forall \mathcal{P}' \forall e \in \mathcal{L} \forall \alpha \in \text{Coins}(\mathcal{P}') : \Pr[(\mathcal{P}'^\alpha \parallel \mathcal{V})(e) = \text{accept}] > \kappa(|e|) \implies (e, E^{\mathcal{P}'e, \alpha}(e)) \in \mathcal{R}$$

for a polynomial p and a knowledge extractor E with runtime bound

$$\frac{p(|e|)}{\Pr[(\mathcal{P}'^\alpha \parallel \mathcal{V})(e) = \text{accept}] - \kappa(|e|)}.$$

Note that both E and \mathcal{P}' are ITMs. $E^{\mathcal{P}'e, \alpha}$ denotes that E gets $\mathcal{P}'e, \alpha$ (e. g., \mathcal{P}' on input e with fixed randomness α) as oracle. \diamond

Remark. A zero-knowledge proof of knowledge is then simply a proof of knowledge with a prover fulfilling the (computational or perfect) zero-knowledge property.

2.6.5 Σ -Protocols

In this section we describe so-called Σ -protocols. These can be used to produce non-interactive proofs (see Section 2.6.7). A Σ -protocol is characterized by the fact that only three messages are sent between the prover and the verifier: First, the prover sends a message – the commitment – to the verifier. The verifier answers – with the challenge – and finally gets the last message from the prover – the response. After this, the verifier has to output accept or reject. The following definition defines the properties of the prover and verifier needed to be considered a Σ -protocol.

Definition 2.23 (Σ -Protocols).

Let \mathcal{R} be a relation for the language \mathcal{L} . An IPS $(\mathcal{P}, \mathcal{V})$ is called a Σ -protocol if only the messages as described above are sent and the following properties hold true.

Completeness.

$$\forall e \in \mathcal{L} : \quad \Pr[(\mathcal{P} \parallel \mathcal{V})(e) = \text{accept}] = 1$$

Special Soundness.

$$(2.4) \quad \exists E \forall e \in \mathcal{L} \forall (\alpha, \beta, \gamma), (\alpha', \beta', \gamma') \in \{\text{Trans}(\mathcal{P} \parallel \mathcal{V}^s, e) \mid (r, s) \in \text{Coins}(\mathcal{P} \parallel \mathcal{V})\} :$$

$$\alpha = \alpha'$$

$$(2.5) \quad \wedge \beta \neq \beta' \\ \implies (e, E(\alpha, \beta, \beta', \gamma, \gamma')) \in \mathcal{R}$$

Special Honest Verifier Zero-Knowledge.

$$\exists S \forall e \in \mathcal{L} \forall \beta \forall \alpha \forall \gamma : \quad \Pr[S(e, \beta) = (\alpha, \beta, \gamma)] = \Pr[\text{Trans}(\mathcal{P} \parallel \mathcal{V}^\beta, e) = (\alpha, \beta, \gamma)]$$

where \mathcal{V}^β is \mathcal{V} that always sends the challenge β . ◇

2.6.6 Standard Σ -Protocol for Homomorphic Functions

For a homomorphic function

$$f : \quad \mathbb{W} \rightarrow \mathbb{E}$$

with

$$(2.6) \quad \forall a \in \mathbb{T} \forall x, y \in \mathbb{W} : \quad a \cdot f(x) + f(y) = f(a \cdot x + y)$$

we can define a “standard” Σ -protocol $\Sigma_{\text{std}} = (\mathcal{P}_{\text{std}}, \mathcal{V}_{\text{std}})$ for the language

$$\mathcal{L}_{\text{std}} = \{e \in \mathbb{E} \mid \exists w \in \mathbb{W} : (e, w) \in \mathcal{R}_{\text{std}}\}$$

defined by

$$\mathcal{R}_{\text{std}} = \{(e, w) \in \mathbb{E} \times \mathbb{W} \mid f(w) = e\}.$$

Note that, if \mathbb{T} are simply integers, we can use Corollary 2.1 and only require that f is a homomorphism (as in Definition 2.1). Additionally, non-zero elements (e. g., not the neutral element with respect to addition) of \mathbb{T} need to be invertible w. r. t. multiplication.

Theorem 2.14 (Σ_{std} is a Σ -Protocol).

$\Sigma_{\text{std}} = (\mathcal{P}_{\text{std}}, \mathcal{V}_{\text{std}})$ as in Figure 2.8 is a Σ -protocol for \mathcal{L}_{std} with a homomorphic function $f : \mathbb{W} \rightarrow \mathbb{E}$ (see Equation (2.6)).

Proof. E_{std} and S_{std} from Figure 2.9 can be used to show that $(\mathcal{P}_{\text{std}}, \mathcal{V}_{\text{std}})$ is a Σ -protocol.

<pre> 1: prover $\mathcal{P}_{\text{std}}(e)$ 2: let $e = f(w)$ 3: $a \xleftarrow{\\$} \mathbb{W}$ 4: $\alpha \leftarrow f(a)$ 5: send (commitment, α) 6: receive (challenge, β) 7: $\gamma \leftarrow a + \beta \cdot w$ 8: send (response, γ) 9: end prover </pre>	<pre> 1: verifier $\mathcal{V}_{\text{std}}(e)$ 2: receive (commitment, α) 3: $\beta \xleftarrow{\\$} \mathbb{T}$ 4: send (challenge, β) 5: receive (response, γ) // also check if $(\alpha, \gamma) \in \mathbb{E} \times \mathbb{W}$ 6: if $\alpha + \beta \cdot e = f(\gamma)$ then 7: return accept 8: else 9: return reject 10: end if 11: end verifier </pre>
---	--

Figure 2.8: Standard Σ -protocol for a homomorphic function f .

Completeness. \mathcal{V}_{std} accepts iff.

$$\alpha + \beta \cdot e = f(\gamma).$$

For an honest \mathcal{P}_{std} we have

$$\begin{aligned} \alpha + \beta \cdot e &= f(\gamma) \\ f(a) + \beta \cdot e &= f(a + \beta \cdot w) \end{aligned}$$

which is

$$f(a) + \beta \cdot f(w) = f(a + \beta \cdot w)$$

as $e \in \mathcal{L}_{\text{std}}$. By assumption (see Equation (2.6)), these terms are equal and \mathcal{V}_{std} accepts all $e \in \mathcal{L}_{\text{std}}$ when interacting with \mathcal{P}_{std} .

Special Soundness. We can show that E_{std} can be used to extract the witness w for e such that $(e, w) \in \mathcal{R}_{\text{std}}$. The output of E_{std} is

$$\begin{aligned} &(\gamma' - \gamma) \cdot (\beta' - \beta)^{-1} \\ &= ((a' + \beta' \cdot w) - (a + \beta \cdot w)) \cdot (\beta' - \beta)^{-1} \\ &\stackrel{(2.4)}{=} ((a + \beta' \cdot w) - (a + \beta \cdot w)) \cdot (\beta' - \beta)^{-1} \\ &= (\beta' \cdot w - \beta \cdot w) \cdot (\beta' - \beta)^{-1} \\ &\stackrel{(2.5)}{=} w, \end{aligned}$$

e. g., the witness computed by \mathcal{P}_{std} .

Special Honest Verifier Zero-Knowledge. The challenge β is fixed in both probabilities, so we can neglect it here. Now, the only randomness left in S_{std} and $(\mathcal{P}_{\text{std}} \parallel \mathcal{V}_{\text{std}}^\beta)$ is picking a random element of \mathbb{W} . Additionally, we have

$$\gamma = a + \beta \cdot w \qquad \alpha = f(\gamma) - \beta \cdot e$$

<pre> 1: procedure $E_{\text{std}}(\alpha, \beta, \beta', \gamma, \gamma')$ 2: $w \leftarrow (\gamma' - \gamma) \cdot (\beta' - \beta)^{-1}$ 3: return w 4: end procedure </pre> <p style="text-align: center;">(a) Extractor for Σ_{std}.</p>	<pre> 1: procedure $S_{\text{std}}(e, \beta)$ 2: $\gamma \xleftarrow{\\$} \mathbb{W}$ 3: $\alpha \leftarrow f(\gamma) - \beta \cdot e$ 4: return (α, β, γ) 5: end procedure </pre> <p style="text-align: center;">(b) Simulator for Σ_{std}.</p>
---	--

Figure 2.9: Extractor and simulator for Σ_{std} .

in $(\mathcal{P}_{\text{std}} \parallel \mathcal{V}_{\text{std}}^\beta)$ and S_{std} , respectively. These statements are equivalent:

$$\begin{aligned}
\gamma &= a + \beta \cdot w \\
f(\gamma) &= f(a + \beta \cdot w) \\
&\stackrel{(2.6)}{=} f(a) + \beta \cdot f(w) \\
&= \alpha + \beta \cdot e \\
\alpha &= f(\gamma) - \beta \cdot e.
\end{aligned}$$

The runs of S_{std} can be mapped to the ones of $(\mathcal{P}_{\text{std}} \parallel \mathcal{V}_{\text{std}}^\beta)$ (and vice versa) while the underlying probability space is the same. The probabilities are the same as such a bijective mapping exists. \square

Remark 1. We did not use that f is hard (or impossible) to invert. This is not needed for the zero-knowledge property. If f is easy to invert, e. g., any verifier could do so in polynomial time, the zero-knowledge property is trivially fulfilled as the verifier could do the whole proof themselves (i. e., using Theorem 2.13).

Remark 2. We instantiate this standard Σ -protocol for several functions to get zero-knowledge proofs needed for our protocols.

Remark 3. The language \mathcal{L}_{std} might be quite trivial, i. e., $\mathcal{L}_{\text{std}} = \mathbb{E}$. If f is an encryption and $\mathbb{E} = \mathbb{Y}$ where all elements of \mathbb{Y} are valid ciphers, then proving $e \in \mathcal{L}_{\text{std}}$ is not of much significance. Instead, it is essential that it is a proof of knowledge for a witness. In Section 2.6.8, we will discuss this in more detail.

2.6.7 Non-Interactive Proof Systems

For many purposes it is very convenient to have non-interactive proofs. An obvious advantage is that these require less interaction (e. g., costly communication). In this form it is also possible to generate proofs and to publish them, which allows everyone to verify these proofs for themselves. All proofs we mention in Sections 2.6.8 to 2.6.10 can be made non-interactive.

A simple and popular way of generating interactive proofs is to use Σ -protocols and the Fiat–Shamir transform [52]. This is illustrated in Figure 2.7d. To transform a Σ -protocol $(\mathcal{P}, \mathcal{V})$ into a non-interactive zero-knowledge proof, an ideal hash function H is required.⁶ We use this hash function

⁶As this is not possible in practice [102], cryptographic hash functions are used instead.

in the non-interactive prover and non-interactive verifier. Note that, while zero-knowledge proofs are not composable in general [30, 63], we can modify Σ -protocols to get protocols that are secure in the UC-framework [55].

The new prover $\mathcal{P}_{\text{non-interactive}}$ runs $(\mathcal{P} \parallel \mathcal{H})(e)$ to generate the transcript $\text{Trans}(\mathcal{P} \parallel \mathcal{H}, e) = (\alpha, \beta, \gamma)$. Here, \mathcal{H} is a new verifier that simply sends the challenge $\beta = H(e, \alpha)$ (as the output of the verifier is not part of the transcript, there is no need to simulate \mathcal{V} in \mathcal{H}). Other public parameters of the IPS or the underlying language might be included in the hash as well.

The new verifier $\mathcal{V}_{\text{non-interactive}}$ receives the transcript (α, β, γ) from $\mathcal{P}_{\text{non-interactive}}$. The verifier accepts, if and only if the following two conditions are met: 1. The challenge was computed correctly (e. g., $\mathcal{V}_{\text{non-interactive}}$ checks that $\beta = H(e, \alpha)$) and 2. \mathcal{V} would output accept in a run with transcript (α, β, γ) . The first condition makes sure that $\mathcal{P}_{\text{non-interactive}}$ actually used H for the challenge. The second condition checks that the original verifier \mathcal{V} would accept.

For non-interactive proofs of NP-languages we use the following notation:

$$\pi \stackrel{s}{\leftarrow} \text{PROVE}(w, e).$$

This runs the non-interactive proof with a witness w and a (potential) language element e and returns a representation π that includes all relevant information (especially for which language a proof is generated, e , and all messages between the prover and verifier) to verify the proof. It can then be verified as follows:

$$\phi \leftarrow \text{VERIFY}(\pi)$$

where $\phi = \text{accept}$ if the proof representation was well-defined and the verifier returns accept and $\phi = \text{reject}$ otherwise.

2.6.8 Proofs of Plaintext-Knowledge

As we consider additive homomorphic encryption schemes, we can simply instantiate Σ_{std} from Section 2.6.6 with

$$\begin{aligned} f &= \text{ENC}_k \\ \mathbb{W} &= \mathbb{X} \times \mathbb{R} \\ \mathbb{E} &= \mathbb{Y} \\ \mathbb{T} &= \mathbb{X}. \end{aligned}$$

Operations (addition and multiplication) on \mathbb{W} are then component-wise, while \oplus and \otimes can be used for \mathbb{E} .

Corollary 2.15 (Σ_{std} is a Proof of Plaintext-Knowledge).

Σ_{std} instantiated as described above is a zero-knowledge proof of plaintext-knowledge.

Proof. Σ_{std} provides a proof of knowledge of a witness $w \in \mathbb{X} \times \mathbb{R}$ such that $\text{ENC}_k(w) = e$ for a cipher e , e. g., this is a proof of knowledge for a plaintext-randomness pair that can be used to generate the cipher. This implies that it is also a proof of plaintext-knowledge. \square

```

1: adversary  $F_{\text{CPA}}(1^\eta, k)$ 
2:   return (0, 1)
3: end adversary

```

(a) Finder to win the CPA-security game.

```

1: adversary  $G_{\text{CPA}}(1^\eta, k, y)$ 
2:   let  $(\mathcal{P}, \mathcal{V})$  be an IPS for the language of  $\mathcal{R}_{\text{plaintext-knowledge}}$ 
3:   find witness  $w$  using  $k, x = 0$ , and  $y$ 
4:   if  $(\mathcal{P}^{(0,w)} \parallel \mathcal{V})((k, y)) = \text{accept}$  then
5:     return 0
6:   else
7:     return 1
8:   end if
9: end adversary

```

(b) Guesser to win the CPA-security game.

Figure 2.10: CPA-adversary using zero-knowledge proofs.

Remark 1. Alternatively, specialized proofs (i. e., for RLWE encryption [101]) can be used.

Remark 2. We write $\pi \stackrel{\$}{\leftarrow} \text{PROVE}_{\text{ENC}}((x, \tilde{x}), (k, (x)))$ for the non-interactive version of the proof.

Remark 3. The randomness used for encryption is needed as private knowledge for a prover. This observation leads to the following theorem.

Theorem 2.16 (Proofs of Plaintext-Knowledge Require Information).

A proof of plaintext-knowledge (in polynomial time) requires more information than only a plaintext x and cipher (x) .

Proof. Consider the adversary $A_{\text{CPA}} = (F_{\text{CPA}}, G_{\text{CPA}})$ in Figure 2.10. The relation for the proof is the following:

$$\mathcal{R}_{\text{plaintext-knowledge}} = \{((x, w), (k, y)) \mid \text{DEC}(w, y) = x \vee \text{ENC}(k, x, w) = y\}.$$

The proof (generated in Figure 2.10b) will be accepted with non-negligible probability if 0 was encrypted by the game because of the completeness property of interactive proof systems. If 1 was encrypted, the proof will be rejected with non-negligible probability because of soundness. This leads to a non-negligible advantage of A_{CPA} in $\mathfrak{G}_{\mathcal{E}, A_{\text{CPA}}}^{\text{CPA}}$, which is a contradiction to the assumption that the encryption scheme is CPA-secure. Which means, there is no proof for plaintext-knowledge or it is not possible to find a witness to prove plaintext-knowledge (in polynomial time or with non-negligible probability). As Theorem 2.12 and Corollary 2.15 imply that a proof of plaintext-knowledge exists, we conclude that we either need more information than only a plaintext and a cipher or superpolynomial time to prove plaintext-knowledge. \square

2.6.9 Proofs of Correct Decryption

Proving correct decryption is generally not as easy as proving plaintext-knowledge. For plaintext-knowledge we assumed that the randomness to compute the cipher is known to the proving party. For proofs of correct decryption, we do not want to assume this, as our protocols require decryptions of values that are combinations of other parties' ciphers. This implies that the decrypting party does not know the randomness used to compute the cipher that is to be decrypted as encryption-randomness is usually kept secret. The following theorem then implies that knowledge of the private key is required.

Theorem 2.17 (Proofs of Decryption Require Information).

A proof of correct decryption (in polynomial time) requires more information than only a plaintext x and cipher $\langle x \rangle$.

Proof (Sketch). Analogous to the proof of Theorem 2.16 with

$$\mathcal{R}_{\text{decryption}} = \{(w, (k, x, y)) \mid \text{DEC}(w, y) = x \vee \text{ENC}(k, x, w) = y\}$$

instead of $\mathcal{R}_{\text{plaintext-knowledge}}$ and $(\mathcal{P}^w \parallel \mathcal{V})(k, 0, y)$ used in Figure 2.10b. \square

Additionally, we do not know of a general way to prove a correct decryption in an encryption scheme agnostic way. This means, specialized proofs have to be employed for each scheme. For the classical schemes considered here, proofs of decryption exists – e. g., for ElGamal and threshold Paillier encryption [47], discrete logarithm equality proofs [33] can be used – and for RLWE-based schemes, proofs have been developed as well [101]. We write $\pi \stackrel{s}{\leftarrow} \text{PROVEDEC}(\hat{k}, (k, \langle x \rangle, x))$ for the non-interactive version of a decryption proof.

2.6.10 Proofs of Committed Ciphers

In the protocol (Section 4.3.2), we need to prove that a cipher $\langle x \rangle$ and a commitment $\llbracket x \rrbracket$ are computed for the same plaintext x . One way to do this, is to instantiate Σ_{std} with

$$\begin{aligned} f &: (x, \tilde{x}, \tilde{u}, \tilde{v}) \mapsto (\text{ENC}_k(x, \tilde{u}), \text{ENC}_k(\tilde{x}, \tilde{v}), \text{COM}_{\tilde{k}}(x, \tilde{x})) \\ \mathbb{W} &= \mathbb{X} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \\ \mathbb{E} &= \mathbb{Y} \times \mathbb{Y} \times \mathbb{C} \\ \mathbb{T} &= \mathbb{X}. \end{aligned}$$

Addition and multiplication-with-constant in Σ_{std} (see Figure 2.8) are then defined component-wise. Note that k and \tilde{k} are fixed but should be still treated as input for a proof.

Corollary 2.18 (Σ_{std} is a Proof of Committed Ciphers).

Σ_{std} instantiated as described above is a zero-knowledge proof of committed ciphers.

Proof. We can prove this like in Corollary 2.15. Let us look at $e = (\langle x \rangle, \langle \tilde{x} \rangle, \llbracket x \rrbracket)$. The relation that is proven with Σ_{std} implies that $\langle x \rangle$ and $\llbracket x \rrbracket$ are computed for the same plaintext x . This is what we wanted to achieve. \square

Remark 1. A similar zero-knowledge proof is used in the auditable version of SPDZ [12].

Remark 2. We write $\pi \stackrel{\$}{\leftarrow} \text{PROVECOM}((x, \tilde{x}, \tilde{u}, \tilde{v}), (k, \check{k}, (x), (\tilde{x}), \llbracket x \rrbracket))$ for the non-interactive version of the proof.

2.7 Multi-Party Computation

Secure multi-party computation (MPC) tries to solve the following problem: n parties want to compute a function f of their inputs (x_1, \dots, x_m) (some parties can have multiple inputs, others none, etc.). The outputs $(y_1, \dots, y_l) = f(x_1, \dots, x_m)$ should be the only thing that the parties learn (again, multiple outputs at the same party are possible, or some parties getting the same outputs). This property is called privacy. Other desirable properties are correctness, independence of inputs, guaranteed output delivery, and fairness. See Section 2.8.1 for more details on these security properties. Whether these properties can be guaranteed or not, highly depends on the capabilities of the adversary. Some real-world applications of MPC are given by Archer et al. [10].

2.7.1 Classification

Adversaries can be classified by their corruption strategy (static or dynamic), behavior (passive, active, or covert), and complexity (computational or information theoretic). For static adversaries, the corrupted parties are fixed at the start of the protocol. Dynamic (or adaptive) adversaries might corrupt honest parties during the protocol run. Passive (or semi-honest or honest-but-curious) adversaries follow the protocol like honest parties – they might try to use all information gathered during a protocol run to learn some secrets. Active (or malicious) adversaries might do anything – like sending messages of the wrong format or with wrong content etc. Covert adversaries might only deviate from the protocol if the risk of being caught is low enough for them. Computationally bounded adversaries have limited computational capabilities (authenticated channels are assumed). Information theoretic adversaries have unlimited computational power (secure channels – e. g., authenticated channels where the messages stay secret – are assumed).

Additionally, the execution model of the protocol could be stand-alone (only a single instance of the protocol is running) or concurrent (multiple instances are running at the same time). More details on the security properties of MPC protocol (also with respect to the different kinds of adversaries) are given in Section 2.8.

2.7.2 General-Purpose Protocols

While there certainly is a need for special purpose MPC protocols (such as oblivious transfer [75, 104] used as building block in some general-purpose protocols), we focus here on general-purpose protocols. There are two orthogonal concepts: the function representation and the function evaluation.

General Function Representation

Common types to represent arbitrary functions are circuits. Boolean and arithmetic circuits are capable of representing the same functions,⁷ but for certain types of functions (and certain types of function evaluation) one might be more efficient than the other.

General Function Evaluation

To evaluate circuits, there are multiple options as well. Boolean circuits can be evaluated using so-called garbled circuits [15, 117] but this technique can be extended for arithmetic circuits as well [9].

Other techniques (mainly for arithmetic circuits) include using secret sharing (see Section 2.2) and fully homomorphic encryption. When using secret sharing, we already saw that additions of shares are possible. If – in addition to that – protocols make sure that shares are generated correctly, multiplication is realized, and the reconstruction is done on the actual shares, any circuit can be evaluated. This is done in SPDZ and other protocols.

If functions should be evaluated on ciphertexts, an encryption scheme can be used, where any arithmetic circuit can be evaluated on ciphertexts. If the resulting ciphers are decrypted, the result is the same as evaluating the arithmetic circuit on plaintexts that were used to generate the input ciphertexts. This is called fully homomorphic encryption (FHE) [58].

This approach has several advantages, such as: 1. Interaction is only required for inputs and outputs [86], 2. the arithmetic circuit is not required to be public (could be the secret of a single party), and 3. it has many practical applications [11]. But still – while there is a standard [6] and several open-source implementations [69] – the main problem of fully homomorphic encryption is efficiency [11]: While it is theoretically possible to evaluate any arithmetic circuit using a FHE scheme, it is still slow.

On the other hand, somewhat homomorphic encryption (SHE) and leveled homomorphic encryption – where only certain types of circuits (e. g., circuits of a certain depth) can be correctly evaluated – are more efficient but cannot be used for all applications. In fact, those types of encryption schemes are what is used to construct fully homomorphic schemes. For our protocol (and other SPDZ-like ones), only a single ciphertext multiplication (preceded and followed by multiple ciphertext additions) is needed. Thus, a somewhat homomorphic encryption scheme is sufficient.

As we already mentioned, we can use secret sharing and fully homomorphic encryption to evaluate Boolean circuits as well. For this, the shares or encrypted values are bits. Addition and multiplication correspond then to exclusive-or (XOR) and conjunction (AND) operations on bits.

⁷A Boolean is an arithmetic circuit over \mathbb{Z}_2 . An arithmetic circuit can be represented as Boolean circuit by generating Boolean sub-circuits for the arithmetic operations.

2.8 MPC Security Properties

A protocol for (secure) multi-party computation should provide several guarantees. The classical properties describe guarantees for honest parties as well as what an adversary can achieve in the protocol.

2.8.1 Classical MPC Security Properties

Here, we present the main properties that classical MPC protocols should achieve. The description of the properties is based on the definitions of Goldwasser and Lindell [66].

Privacy

A party should only learn their own output. Everything they know about other parties' inputs or outputs should be what can be derived from this output.

Correctness

The output a party receives should be correct.

Independence of Inputs

A party cannot choose their input based on the input of other parties.

Guaranteed Output Delivery

Parties should not be able to prevent other parties from getting their designated output.

Fairness

Honest parties receive outputs iff. corrupted parties do so.

2.8.2 Previous Results on Classical MPC Security Properties

With honest majority, it is possible to compute any function while achieving the above properties using trapdoor permutations [64] or with two-thirds honest majority without any assumptions [17, 32].

Notation	Explanation
I_Π	A protocol instance of protocol Π .
R_Π	A run of a protocol Π .
$I_\Pi \models \phi$	In the context of accountability: A true verdict ϕ in a protocol instance I_Π of protocol Π .
	In the context of verifiability: A true verifiability assumption φ in a protocol instance I_Π of protocol Π .
$\gamma \Rightarrow \phi_1 \mid \dots \mid \phi_v$	An alternative notation for an accountability constraint.
$\phi \Rightarrow C$	An ensured accountability constraint. Note that the protocol run R_Π (or the protocol Π) are not part of this (shortened) notation but have to be considered to determine if the accountability constraint is ensured.
$\phi \Rightarrow \Phi$	An accountability property where all accountability constraints are ensured.

Table 2.3: Notation for accountability and verifiability in the KTV-framework.

2.8.3 The KTV-Framework

Küsters et al. developed a framework – the KTV-framework [81] – to capture accountability in protocols where some parties might be dishonest. The framework can also be used to model verifiability (as it is a “weak form of accountability” [81]). They give symbolic and computational models of their definitions – we present (and use) the latter in this work. Their definitions use a notion of processes, protocols, protocol runs, and protocol instances that we transfer to our notation of ITMs and protocols (as in Section 2.5) to use only a single notation throughout this work.

We model a protocol instances as a combination of a protocol $\Pi = (M_1 \parallel \dots \parallel M_n)$ with an adversary A , e. g., an instance I_Π is $(\Pi \parallel A)$, that always corrupts the same parties.⁸ A run R_Π of the protocol Π is a run of some protocol instance (with some adversary A). A ITM M_i is honest in a run if A did not corrupt M_i in this run (as the corrupted machines are the same for all runs of an instance, we can also say if a machine is honest for some fixed instance of the protocol). A property γ is a subset of the set of all runs of the protocol Π .

Central to the KTV-framework is a special party – the so-called judge J . Verifiability and accountability are defined from the viewpoint of the judge. If the judge only uses public information, everyone (every party of the protocol or an external auditor) can play the (abstract) role of the judge. In some protocols J could be an actual party that might get information from (potentially corrupt) parties and has to give a verdict (the actual verdict is different for accountability and verifiability). In the following we describe the definitions of accountability and verifiability in the KTV-framework together with the necessary terminology for these two security properties. See Table 2.3 for some of the notation used in this work regarding the framework.

⁸For example, A only corrupts M_1 in the protocol instance $(M_1 \parallel M_2 \parallel A)$, then M_1 has to be corrupted in every run of this instance. Another adversary A' could corrupt M_2 . If all adversaries are considered, all possible corruption scenarios are covered.

Accountability

In the following, we introduce the necessary terminology and notation for accountability: A verdict is a positive Boolean formula $\phi \in F_{\text{dis}}$ with propositions of the form $\text{DIS}(M_i)$. F_{dis} is the set of all such formulas. A verdict ϕ is true in a protocol instance I_Π if the formula ϕ evaluates to true when we set $\text{DIS}(M_i)$ to false iff. M_i is honest in this protocol instance. An accountability constraint $C = (\gamma, \phi_1, \dots, \phi_v)$ is a tuple of a protocol property γ and verdicts where we say that J ensures the constraint in a run R_Π if $R_\Pi \in \gamma$ or J outputs a verdict ϕ in R_Π that implies one of the ϕ_i . A set Φ of accountability constraints is called accountability property.

Definition 2.24 (Accountability).

For a protocol Π , a judge $J \in \Pi$ ensures (Φ, δ) -accountability for an accountability property Φ and a bound $\delta \in [0, 1]$ if the following properties hold true for all protocol instance I_Π .

Fairness. The judge J is fair if $\Pr[I_\Pi(1^n) \mapsto \{J : \phi \mid I_\Pi \not\models \phi\}]$ is negligible.

Completeness. The probability $\Pr[I_\Pi(1^n) \mapsto \{J : \phi \mid \phi \not\Rightarrow \Phi\}]$ is δ -bounded. \diamond

Remark 1. Fairness is independent of the accountability property and models that a fair judge (almost) never claims that honest parties cheated.

Remark 2. Completeness models that the judge rarely fails to ensure the accountability constraints. Recall that $\phi \Rightarrow \Phi$ means that for each accountability constraint $(\gamma_i \Rightarrow \phi_{1,i} \mid \dots \mid \phi_{v,i}) \in \Phi$ we have that either the goal γ_i is met or the verdict implies some combination of dishonest parties. $\phi \not\Rightarrow \Phi$ means then that for some constraint, the goal is not met but J cannot blame the dishonest parties in the way that the constraint requires it.

Remark 3. An accountability property Φ gives individual accountability if each $\phi_{j,k}$ implies a formula of the form $\text{DIS}(M_i)$, e. g., no matter what verdict is used to ensure constraint by J we can imply individual parties from it.

Now we present an example to illustrate accountability. Consider the (fictitious) MPC protocol $\Pi = (M_1 \parallel M_2 \parallel J)$. Here, M_1 and M_2 want to compute which of them picks a higher number. For this, they first broadcast a commitment to their number and in the second step broadcast the decommitment. Lastly, the decommitments are compared to find the winner. The judge J just gives a verdict but observes the protocol. An accountability constraint C would be

$$\{R_\Pi \mid \text{the higher number of the inputs is found in } R_\Pi\} \Rightarrow \text{DIS}(M_1) \mid \text{DIS}(M_2)$$

and it provides individual accountability. A judging procedure would do the following:

- (1) Receive c_1 from M_1 and c_2 from M_2 .
- (2) Receive (x_1, r_1) from M_1 and (x_2, r_2) from M_2 .
- (3) If (x_1, r_1) is not a valid decommitment for c_1 , post verdict $\phi = \text{DIS}(M_1)$.
- (4) If (x_2, r_2) is not a valid decommitment for c_2 , post verdict $\phi = \text{DIS}(M_2)$.

Note that, if the judge J gives multiple verdicts, they are combined, i. e., $\text{DIS}(M_1)$ and $\text{DIS}(M_2)$ are combined to $\text{DIS}(M_1) \wedge \text{DIS}(M_2)$.

Now we could show that the protocol Π is $(\Phi, 0)$ -accountable with $\Phi = \{C\}$ from above. For fairness, we can see that J would never blame an honest party (as honest parties do not send wrong decommitments). To not fulfill correctness, the judge would have to miss a wrong decommitment (if the decommitment is correct, the right numbers are compared, and the goal is fulfilled). If the commitment scheme is binding, this is only possible with negligible probability (or not at all if it is perfectly binding). Note that we would get a similar result with the constraint

$$C' = \{R_\Pi \mid \text{the higher number of the inputs is found in } R_\Pi\} \Rightarrow \text{DIS}(M_1) \vee \text{DIS}(M_2)$$

which does not provide individual accountability.

Verifiability

For verifiability, we have a slightly different setup. Here, a verdict is simply a binary option (accept or reject) and verifiability assumptions are made. A verifiability assumption is a positive Boolean formula $\varphi \in F_{\text{hon}}$ with propositions of the form $\text{HON}(M_i)$. F_{hon} is the set of all such formulas. An assumption φ is true in a protocol instance if the formula φ evaluates to true when we set $\text{HON}(M_i)$ to true iff. M_i is honest in this protocol instance.

Definition 2.25 (Verifiability).

For a protocol Π with property γ and a verifiability assumption $\varphi \in F_{\text{hon}}$, the goal γ is guaranteed by φ and a judge $J \in \Pi$ ensures δ -verifiability for a bound $\delta \in [0, 1]$ if the following properties hold true for all protocol instance I_Π .

Property 1. If $I_\Pi \models \varphi$, then $\Pr[I_\Pi(1^n) \mapsto \{J : \phi \mid \phi = \text{accept}\}]$ is overwhelming.

Property 2. The probability $\Pr[I_\Pi(1^n) \mapsto \{J : \phi \mid \phi = \text{accept}\}, R_\Pi \notin \gamma]$ is δ -bounded. \diamond

Remark 1. The first property implies that, if the accountability assumptions are met, the judge accepts the protocol. This is independent of the goal γ .

Remark 2. The second property models that J accepts only rarely if the goal is not met.

2.9 E-Voting

First, we cover the basics of voting protocols that are important for our proposed MPC protocol if it is to be used in the context of e-voting. After that, several techniques to implement a voting protocol are sketched.

There are several types of voting protocols that could be used for election or surveys. Classically, the protocol is divided into several phases, i. e., preparation, voting, tallying, and announcing the result. Especially in the context of (verifiable) e-voting, there are fine distinctions that can be made. This is illustrated by Table 2.4. The three examples differ by the points when (partial) results are published.

Example 1	Example 2	Example 3
1. Vote	1. Vote	1. Vote
2. Publish votes		
3. Tally	2. Tally 3. Publish tally	2. Tally
4. Compute result	4. Compute result	3. Compute result 4. Publish result

Table 2.4: Examples of voting protocol sequences.

The steps after publishing these results can then be checked by anyone by simply recomputing them. A natural conclusion would be to try to publish (partial) results as early as possible, but this could lead to privacy issues. Publishing after tallying is similar to classical elections – for example in Germany – where the tally is published and the result (i. e., how many seats each party gets in the parliament) can be recomputed by anyone. In tally-hiding voting, only the election result is revealed – e. g., especially the tally remains secret – and thus not more information than necessary is disclosed (if the result is (part of) the tally, some information about the tally will always be revealed). We shortly summarize examples from Küsters et al. [79] for situations where tally-hiding voting is desirable: 1. in elections with multiple rounds (to prevent bias in subsequent rounds), 2. in elections with a small number of voters (to preserve the privacy of individual voters), 3. to prevent “unnecessar[y] embarrass[ment]” (resulting from a low number of votes for a candidate), or 4. if revealing certain information is forbidden (i. e., by law, rules, or regulations).

Another difference is the vote-type a voter is able to give. We consider the case that there is a set of candidates $C = \{C_1, \dots, C_{\#candidates}\}$ and the vote of voter V_j is

$$v_j = (c_{1,j}, \dots, c_{\#candidates,j}) \in \mathbb{X}^{\#candidates},$$

e. g., the voter gives a plain number for each possible candidate indicating the number of votes the respective candidate gets from this voter. Additionally, a predicate

$$P: \mathbb{X}^{\#candidates} \rightarrow \{\text{false}, \text{true}\}$$

indicates if a vote is valid or not, i. e., if each voter has at most two votes,

$$P: v_j \mapsto \sum_{i=1}^{\#candidates} c_{i,j} \leq 2$$

could be such a check.⁹ Tallying is then simply summing up the valid votes of all candidates: The tally of candidate C_i is

$$c_i = \sum_{\substack{j=1 \\ P(v_j)}}^{\#voters} c_{i,j}.$$

⁹assuming $\mathbb{X} \subseteq \mathbb{N}$

The voting result should then be only a function of the candidate tallies, e. g.,

$$R = f(c_1, \dots, c_{\#candidates}).$$

Other vote-types could be rankings (i. e., each voter gives a list of their top-three candidates) or free-form values (as commonly found in surveys). As we will see later, the previously introduced type of votes (numbers for each candidate) can be easily handled by common types of e-voting protocols. Nevertheless, also these (more powerful) voting protocols can be implemented as e-voting protocols.

2.9.1 E-Voting Techniques

All techniques discussed have one thing in common: The voting phase produces encrypted votes (for each voter). This means, it is necessary to somehow prove that the encrypted votes are correct (e. g., $P(v_j) = \text{true}$ for an encrypted v_j). Typically, this is done with zero-knowledge proofs [4, 18, 79]. This also means that the “publishing” mentioned in Table 2.4 implies decryption. Verifiability (as described in Section 2.8.3) is partially given by keeping most (encrypted) information publicly available – on a so-called bulletin board. This includes the encrypted votes, (zero-knowledge) proofs, intermediate results, etc.

One powerful technique is shuffling of votes [19, 93, 115]. This enables all previously mentioned votes-types, but also publishes non-aggregated votes. The main idea is to shuffle and re-encrypt the encrypted votes (as a single step) and then decrypt these votes. The reasoning behind this is that the permutation (in combination with re-encryption) removes associations between voters and votes, so the votes can be safely decrypted. Re-encryption guarantees that the new ciphers are indistinguishable from the input ciphers (if the encryption scheme is CPA-secure) and the shuffle removes the ability to link the inputs and outputs of re-encryption.¹⁰ Meanwhile, the parties executing the shuffling, re-encryption, and decryption have to prove that these operations are done correctly, as the voting result might be manipulated in one of them. Using this technique, we could implement Example 1 from Table 2.4

Another technique is to use homomorphic properties of the used encryption scheme. This could be used to implement Example 2 and Example 3 of Table 2.4. If an additive homomorphic encryption scheme is used, tallying can be done using the additive homomorphic property of the scheme. To implement Example 2, the encrypted tally would be decrypted after summing up the encrypted votes.¹¹ For Example 3, a function f has to be evaluated on the encrypted tallies. This can be done with FHE or any other MPC protocol that supports encrypted inputs.

Other techniques are possible as well. For example, Benaloh et al. [21] combined shuffling and homomorphic sums to support ranked voting.

¹⁰Without shuffling, the first cipher after re-encryption would be the re-encryption of the first input cipher, the second one for the second, etc.

¹¹This was done – for example – in early verifiable elections [20], but also more recent approaches use this simple technique [90].

2.9.2 Security Properties of E-Voting Systems

E-voting systems should guarantee several security properties. Some are similar to MPC security properties (see Section 2.8.1), i. e., correctness and privacy should be achieved. These properties are somehow different from MPC security properties as the voters (for whom we want to guarantee privacy) are not involved in the protocol.

On the other hand, there are several notions of verifiability in the context of e-voting:

Individual Verifiability

A voter can check that their vote is published (in an encrypted way) [77].

This security property is related to so-called clash attacks [78] where multiple voters get the same receipt (i. e., information that identifies their vote on a bulletin board) so some votes can be changed or removed as long as a vote that corresponds to the receipt is published. Obviously, such wrong receipts should be recognizable as such.

Universal Verifiability

Everyone can check that the election result corresponds to the published (encrypted) votes [77].

This is the core of many e-voting systems and can be achieved by all the mentioned e-voting techniques. Many approaches reveal the tally to achieve this (mix-nets typically reveal all votes after shuffling and re-encrypting; other approaches sum up the encrypted votes and publish the aggregated votes). Then, anyone can compute the election result from the revealed information. Tally-hiding approaches that only reveal the result are rare but possible as well [79].

Eligibility Verifiability

Everyone can check that only voters (registered people / people who are allowed to vote in some way) voted and there is at most one vote per voter [77].

This can be used to prevent so-called bullet stuffing where votes are added to an election to change the result. Some kind of mechanism should prevent voters to vote multiple times (or only count one vote) or anyone to add votes of registered voters that did not vote (i. e., by requiring votes to be signed with valid signing keys).

End-to-End Verifiability

There are several notions of end-to-end verifiability. Intuitively, one could say a definition should make sure that the election result using an e-voting system is the same as the actual voting result. Cortier et al. [36] analyzed several definitions of (end-to-end) verifiability and discovered that some fail to provide this intuition.

One example of how one could define end-to-end verifiability is how Küsters et al. do this for Ordinos: There, the verifiability is equivalent to detecting manipulation of m or more honest votes with a certain probability (as one cannot make sure all voters perform checks on their votes) under certain assumptions [79].

Interestingly, privacy implies individual verifiability [37]. Individual and universal verifiability can be formalized in the KTV-framework [36], but together, they do not (always) imply end-to-end verifiability [36, 78].

3 Related Work

In this chapter, we describe existing MPC protocols and e-voting systems. The MPC protocols are tightly related to our protocol as they are all SPDZ-like protocols. Our protocol builds upon them to get a protocol that achieves accountability and is more suitable for e-voting.

The e-voting systems we describe are a small selection of recent verifiable e-voting systems. Each of them uses another technique to achieve verifiable elections (mix-nets, homomorphic aggregation of votes, and general multi-party computation). By showing how we can use our protocol with these systems in Chapter 8, we show that our protocol can be used with a variety of systems.

3.1 MPC Protocols

As all the protocols mentioned here are related to SPDZ, we start with describing the original version of SPDZ first. Then, we outline an auditable version of SPDZ where anyone (participating in the protocol or not) can check the correctness of the result. Additionally, we describe accountable MPC protocols (based on SPDZ).

Obviously, neither SPDZ nor its auditable version provide accountability. But also, the accountable versions lack some properties to be perfectly suitable for e-voting. Our protocol tries to fix that. In Chapter 7, we compare these protocols to our protocol to highlight the differences.

3.1.1 SPDZ

Damgård et al. [49] developed an efficient MPC protocol: SPDZ. It supports arithmetic circuits over finite fields and is secure for active adversaries corrupting up to $n - 1$ parties. The computation is separated in an online and offline phase, where data of the offline phase is used to make the online phase efficient. For this, the offline phase is mostly independent of the computed function (circuit), but upper bounds on the number of multiplications and the number of inputs have to be given. The online phase has linear communication complexity in the size of the circuit, e. g., the work in the online phase is independent of the number of parties.

The main idea of “SPDZ-like” protocols (SPDZ and variations based on it) is the following: An arithmetic circuit is calculated on shares using an additive homomorphic secret sharing scheme (e. g., each party does linear operation on their shares $\langle x \rangle_i$ locally without interaction). Multiplication still needs interaction, but it is done using protocols which are secure for semi-honest adversaries and are otherwise correct up to an additive error (e. g., the multiplication of x and y might yield $x \cdot y + \delta$ instead of $x \cdot y$ for a δ known to the adversary). To counteract this (and manipulating the protocol in other ways), the arithmetic circuit is also computed on message authentication code (MAC) values $\alpha \cdot x$ for every input x where α is the MAC key. Again, these are shared with the secret sharing

scheme, so every party \mathbb{P}_i holds $\langle \alpha \cdot x \rangle_i$ in addition to $\langle x \rangle_i$. The protocol involves opening values (e. g., revealing the shares) at certain points (during multiplication and for the final output). Here, only the shares are revealed but SPDZ includes a way to later check all opened shares using the MACs (without revealing the MAC key). As no single party (or proper subset of parties)¹ knows the MAC key, cheating is similarly hard as guessing the MAC key.

The protocols to generate shares for inputs and doing multiplication are quite similar to what we use in our protocol (in fact, our protocol includes generating shares and doing multiplication on shares just like in SPDZ). Therefore, we leave them out here. To check opened values (using the MAC), they do the following (actually, this is the more refined MAC checking of Damgård et al. [48] without revealing the MAC key). For m opened values (e. g., $m \cdot n$ shares)

$$x_j' = \text{REC}(\langle x_j' \rangle_1, \dots, \langle x_j' \rangle_n)$$

with $j \in \{1, \dots, m\}$ (note that x_j' might be wrong if some shares are wrong, e. g., $x_j' \neq x_j$ if some $\langle x_j' \rangle_i \neq \langle x_j \rangle_i$), m random values r_j are sampled. Then, every party is supposed to reveal²

$$\langle z \rangle_i = \sum_{j=1}^m r_j \cdot \langle \alpha \cdot x_j \rangle_i - \langle \alpha \rangle_i \cdot \sum_{j=1}^m r_j \cdot x_j'$$

If all values are correct, this reconstructs to $z = 0$. If some value x_j or MAC $\alpha \cdot x_j$ is wrong, $z = 0$ with probability $2/q$ for fields of order q (e. g., $\mathbb{X} = \mathbb{F}_q$) [48]. This means, there is a good way of checking opened values using the MACs without revealing the key.³ A MAC check like this enables them to detect misbehavior when generating input shares, doing multiplication, or opening values.

There has been a lot of work in improving SPDZ (or building on top of it), such as

- covertly and actively secure preprocessing phase and reusable preprocessing data [48],
- faster additive-only implementation with improved performance for the used proof of knowledge [72],
- implementation over power-of-two-integers instead of a field [38],
- application to honest-majority scenarios and optimizations for small fields [34], and
- doubled online phase performance if the function is known in the preprocessing phase [16].

Furthermore, there have been attempts to make SPDZ verifiable or accountable. These are discussed in the following sections.

¹As full-threshold secret sharing is used, all shares have to be known to reconstruct a secret.

²First, each party has to commit to $\langle z \rangle_i$. Otherwise, some party could adjust their share such that all shares reconstruct to 0. This is why Table 7.2 includes communicating a commitment.

³Not revealing the MAC key when checking correct openings is what enables the use of unused preprocessing data for other computations.

3.1.2 Auditable SPDZ

Baum et al. [12] built upon SPDZ to develop a publicly auditable multi-party computation protocol. For this, they augment every shared value x with a commitment $\llbracket x \rrbracket$. Their protocol is structurally similar to SPDZ and can be seen as an augmentation. Here, every party \mathbb{P}_i not only holds a share $\langle x \rangle_i$ and a MAC share $\langle \alpha \cdot x \rangle_i$ but also $\langle \tilde{x} \rangle_i$ and $\langle \alpha \cdot \tilde{x} \rangle_i$ with

$$\begin{aligned} x &= \text{REC}(\langle x \rangle_1, \dots, \langle x \rangle_n) \\ \tilde{x} &= \text{REC}(\langle \tilde{x} \rangle_1, \dots, \langle \tilde{x} \rangle_n) \\ \llbracket x \rrbracket &= \text{COM}_{\tilde{k}}(x, \tilde{x}), \end{aligned}$$

e. g., each party holds a share of the plaintext and randomness of a commitment to the secret x , and a MAC share for both x and \tilde{x} , too. At openings, $\langle x \rangle_i$ and $\langle \tilde{x} \rangle_i$ are revealed instead, while a MAC check is done for $2 \cdot m$ (instead of m) values: x_j' and \tilde{x}_j' . If this check succeeds, it implies that the secrets and randomness for the commitments can be reconstructed. The commitments are only generated for the inputs⁴ but never actually used in the compute phase. Instead, an (external) auditor could get the publicly accessible commitments for the inputs and recompute the circuit on them. Whenever an opening gate is encountered, the publicly opened shares $\langle x \rangle_i$ and $\langle \tilde{x} \rangle_i$ are used to check if they reconstruct to the decommitment of the commitment that the auditor computed themselves.

The compute parties perform (fast) computations on shares, while auditors compute on commitments (which is potentially slower) to check that all opened values are correct. By the binding property of the commitment scheme,⁵ the protocol stays publicly verifiable as long as the compute parties cannot influence the parameter generation of the commitment scheme to learn a backdoor – even if all compute parties are malicious. A (very) similar approach is followed by the next protocol to achieve accountability.

3.1.3 Accountable SPDZ

Cunningham et al. [44] modified SPDZ (or its auditable version) even further to achieve individual abort / individual accountability. To do this, they use commitments for each share instead. This means, each party \mathbb{P}_i now holds $\langle x \rangle_i$, a MAC share $\langle \alpha \cdot x \rangle_i$ and randomness \tilde{x}_i . Now the (Pedersen) commitments are

$$\llbracket \langle x \rangle_i \rrbracket = \text{COM}_{\tilde{k}}(\langle x \rangle_i, \tilde{x}_i)$$

and are published for all inputs. This also means, the MAC check can only be done on the shares and, if it fails, the commitments are used to find cheaters. Recall that, when opening a value, $\langle x \rangle_i$ and \tilde{x}_i are revealed, so everyone can check if a share (together with the randomness) is a decommitment for a commitment. This means, the compute parties do similar work as in the auditable version of SPDZ but if the MAC check fails, they do the same as an auditor in auditable SPDZ, only n times –

⁴Actually, the protocol includes the generation of commitments for each party. These are then combined together with an input to get a commitment for the input. If the commitments are instead combined with the respective shares, we get commitments for each share as in the protocol shown in Section 3.1.3 or in our protocol.

⁵The additive homomorphic property is needed for correctness as well.

for each share instead of once for a shared value. Additionally, they require the encryption scheme used in the protocol to have verifiable decryption (with identifiable abort). This is not a requirement for the above protocols.

3.1.4 Other Accountable MPC Protocols

In the following we shortly describe further MPC protocols that provide identifiable abort, e. g., if the protocol aborts, the party (or parties) that were responsible for the abort (due to some misbehavior) can be individually identified. This is equivalent to individual accountability where the protocol either finishes (the goal is then all protocol runs that do not abort and compute the correct result) or all dishonest parties are detected by some judge.

The protocol of Ishai et al. [70] uses commitments as well. They developed a “semi-honest to malicious with abort” compiler that – as the name implies – converts protocols that are secure with semi-honest adversaries to protocols that work with malicious adversaries but might abort while identifying malicious parties. A disadvantage (for certain use-cases) of their construction is the information theoretic commitment scheme they use. This requires each verifier to hold a different (secret) verification key that corresponds to a single signing key used by the party that commits to a message. Additionally, the verification key cannot be public. This seems to rule out their approach for constructing an MPC protocol that can be publicly verified (and achieves individual accountability) by any party that does not participate in the multi-party computation – even if all parties are corrupted.

Similarly, the protocol of Baum et al. [13] uses information theoretic signatures with verification keys for each party that cannot be public. Their protocol is based on BDOZ [22] and SPDZ [49]. However, they suggest to generate n additional verification keys and publish commitments for them to a bulletin board. The i -th commitment on the bulletin board can be opened by \mathbb{P}_i (by revealing the necessary decommitment). With this adaption, an auditor should come to the same conclusion as \mathbb{P}_i if \mathbb{P}_i detected a misbehaving party and opened their additional verification key. This construction seems to fail if all parties are corrupted. Then, the commitment scheme fails to work (as all signing and verification keys are then known to an adversary). The same is true for the protocol of Ishai et al. [70].

3.2 E-Voting Systems

In addition to the foundational research on e-voting systems (outlined in Section 2.9.1), there are also a lot of complete e-voting systems that try to capture every aspect of e-voting. Examples range from research [4, 79] and open source projects [4, 90] to commercial systems.

One popular verifiable e-voting system is Helios [4]. Küsters et al. developed a tally-hiding system – Ordinos – that can be seen as an extension of Helios [79]. This is why we will focus on presenting Ordinos here.

While there are (very few) other tally-hiding e-voting systems, most of them do not seem to generalize well [79]. Even Ordinos is limited by the MPC protocol they use (it is quite specialized to sorting the total votes of candidates), which is why we propose a general-purpose MPC protocol that fulfills the requirements to be used by Ordinos.

3.2.1 Helios

Helios [4] was designed to be simple (to understand) and publicly auditable. The correctness of the result is designed to be verifiable even if Helios (the server implementing the Helios protocol) is corrupt. At the same time, privacy is not guaranteed if Helios is corrupt. Also, they stress that Helios should only be used in elections where the risk of coercion is low as the system does not solve the problem of coercion. Instead, they recommend it for “student government, local clubs, online groups such as open-source software communities” [4]. Helios includes the (software) design [4], an open-source implementation [2], and a website to manage elections [3].

In the voting phase, a voter can generate a ballot and encrypt it using ElGamal encryption (see Section 2.3.3). This includes the option for the voter to audit the ballot by also receiving the randomness to check if the vote (encrypted with the shown randomness) results in the shown ciphertext. A hash of the ciphertext is shown as well.

The (encrypted) ballots are published on a bulletin board (ballots are associated with identifiers of the voters). A mix-net is used to shuffle and re-encrypt the ballots (see Section 2.9.1). Then, the shuffled ballots are decrypted. For both mixing and decryption, non-interactive zero-knowledge proofs are provided. After this, the tallying is done.

Everyone can obtain the whole election data. This includes the bulletin board and ZK proofs. Voters can check if their vote is on the bulletin board and the encrypted vote was the same as the one they published. The proofs can be verified and the tally can be re-done to check the result.

3.2.2 Ordinos

Ordinos [79] takes inspiration from Helios but extends its capabilities in several ways. Firstly, tally-hiding voting is supported, whereas Helios always reveals the full tally. Secondly, Ordinos is not only verifiable but it provides accountability (see Section 2.8.3 for more details).

The security properties of Ordinos are proven by Küsters et al. in their specification of Ordinos [79]. Primitives (such as the encryption scheme or zero-knowledge proofs) can be replaced by others as long as they provide the same properties.

The election process is straight forward: 1. Voters publish their encrypted votes (on a bulletin board). 2. The votes are summed up (using the additive homomorphic property of the encryption scheme). 3. An MPC protocol computes the election result (using the aggregated ciphers as input). 4. The result is published. The following parties participate in the process:

- #voters voters, each with a voter support device and a voter verification device,
- n trustees,
- one authentication server, and
- one bulletin board.

The voters use the support devices to vote (to generate, encrypt, and submit their vote) and the verification device to verify the actions of their support device (e. g., check if it acted as expected). The trustees do the tallying and compute the election result. The authentication server makes sure that only voters (not anyone else) submit their votes. All information that is used to verify the election is handled by the bulletin board. For more details on the election process and the actions each party has to perform, see the Ordinos specification [79].

Küstern et al. analyzed Ordinos with the KTV-framework [36, 81] (see Section 2.8.3 for more details). They proved end-to-end verifiability, accountability, and privacy for Ordinos. Descriptions of these properties can be found in Section 2.9.2.

In their implementation, Küstern et al. use a threshold-variant of the Paillier cryptosystem [96], non-interactive zero-knowledge proofs [7, 41, 109], a signature scheme (unspecified; needs to be EUF-CMA-secure [67]), and a greater-than MPC protocol [85]. They measured the runtime of the tallying phase for different systems configurations (number of voters, number of candidates, number of trustees, different communication channels (same machine, local, internet)).

The runtime of the tallying seems independent of the number of voters, as the votes are aggregated and the MPC protocol runs on the aggregated values. The numbers of trustees (tested for 2 to 8 trustees on a local network) influences the timing only in a minimal way – they state the logarithmic complexity of the MPC protocol as the reason for this. Their evaluation of the election result uses the MPC protocol a quadratic number of times (in the number of candidates). This can be seen in the runtime of the system when changing the number of candidates. They assess their implementation as “quite independent of the specific networks”, where running on a single machine (with 2 to 8 trustees) or in a local network (with 3 trustees) differ only little. The difference in the used network (local vs. over the internet) is more pronounced with larger number of candidates (evidently, it also increases quadratically).

3.2.3 Other E-Voting Systems

In contrast to Helios, Microsoft’s ElectionGuard [18, 90] is designed to be used in a distributed way, e. g., multiple trustees perform an election. For this, they use a threshold variant of ElGamal encryption. Similarly to the above, ballots are encrypted, proofs (that the vote is valid) and tracking codes (for voters to confirm that votes are included in the set of ballots) are generated, and encrypted votes are published.

They use the additive homomorphic property of ElGamal encryption to tally the votes and the trustees decrypted the sum of the votes for each choice / candidate. Here, an auditor can again check the proofs and check that the right aggregation of votes was decrypted.

In contrast to Helios (and Ordinos), ElectionGuard is not seen as web-only system. They propose to use their system also in addition to physical elections. There, they one could decrypt a random choice of ballots and compare them to the physical ones.⁶ This enables risk-limiting audits [18, 84].

⁶In this case, the voters would not get the tracking codes to preserve privacy.

Certainly, there might be more e-voting systems worth mentioning here⁷ but to our knowledge, they all either use mix-nets or compute a partial result using the encrypted votes [35, 61, 80, 114]. This might be the sum of votes for each candidate or something more sophisticated. Just like most e-voting systems do not seem to be tally-hiding, only few are designed with accountability in mind [92].

⁷for example, the criticized [87] e-voting in Switzerland [112]

4 Formal MPC Protocol

In this chapter, we present an MPC protocol (Section 4.2) together with all necessary functionalities (Section 4.1) and protocols (Section 4.3). We also prove that the protocols realize the respective functionalities in the UC-framework.

All functionalities assume static corruption by an adversary and provide an auditing interface. For all functionalities, auditing reveals all malicious parties (a subset of corrupted parties that tried to manipulate the execution).

There are a few restrictions to what the environment can do in the protocols:

1. Parties $\mathbb{P}_i \in \mathbb{P}$ can only **output** values once in each phase.
2. All corrupted parties $\mathbb{P}_i \in \mathbb{P}$ query their **output** in the compute phase from the adversary before outputting it.
3. All parties can only receive an **input** as in the protocol (especially only once).
4. The protocols have an inherent sequence that cannot be deviated from. For example in our MPC to evaluate arbitrary arithmetic circuits (Figure 4.5), first, the INITIALIZE sub-protocol has to be executed. Then, all #inputs inputs have to be given – each with the INPUT sub-protocol. After this, COMPUTE evaluates the circuit. Only after that, the protocol can be audited with the AUDIT sub-protocol.

The same is true for the functionality that corresponds to the protocol. Also, all other protocols and functionalities have to run their sub-protocols or sub-functionalities in order. The order for the respective protocol / functionality should be clearly recognizable. In a real protocol, it should be checked that the sub-phases are executed in order and auditing should also handle these cases.

5. The auditing party $\mathbb{P}_{\text{audit}}$ cannot be corrupted with the same reasoning as Baum et al. [12]: As long as anyone is interested in auditing the protocol, it can be successfully audited using only publicly available information, e. g., anyone can act as $\mathbb{P}_{\text{audit}}$. Furthermore, the computation cannot be influenced by malicious auditors.

These restrictions are mostly for more convenient modelling and are enforced by the static corruption wrapper. For example, Item 1 can be left out, but all functionality have to be changed in the following way: The adversary is always allowed to instruct the functionality to send certain outputs to corrupted parties. The reason for this restriction is to remove the possibility to distinguish the real and ideal worlds by instructing corrupted parties to output messages that are not part of the protocol. This would be only possible in the real world, e. g., the environment could distinguish based on whether these messages are received from the corrupted parties or not.

Additionally, all functionalities assume that public keys (the public key for the encryption scheme and the public parameters for the commitment scheme) are available after the initialization phase. This models that the parties agree on these keys during initialization and it is part of \mathcal{F}_{key} which is used by the protocols.¹ How the functionalities get to know the keys is left out for simplicity, but it could be modelled in the following way: The public keys have to be given in **input** messages for the functionalities (in every phase after initialization), i. e., $(\text{input}, \mathbb{P}_{\text{input},i}, v, k, \check{k})$ instead of $(\text{input}, \mathbb{P}_{\text{input},i}, v)$ in the **INPUT** phase of the functionality shown in Figure 4.4.

4.1 Prerequisite Functionalities

Unlike SPDZ and similar protocols, we do not separate our protocols explicitly in online and offline parts. Instead, we separate them by their intended function to avoid getting one big offline functionality with many unrelated functions. How preprocessing (e. g., moving some parts of the protocol in an offline phase) is handled is shown in Section 5.6. The main protocol – discussed in Section 4.2 – requires functionalities that will be used by the compute parties. The functionalities are: $\mathcal{F}_{\text{rand-cipher}}$, $\mathcal{F}_{\text{com-cipher}}$, \mathcal{F}_{key} , and $\mathcal{F}_{\text{bulletin}}$.

The following, we discuss the first three – $\mathcal{F}_{\text{bulletin}}$ is only described on an intuitive level: 1. Parties send messages to $\mathcal{F}_{\text{bulletin}}$ and it relays it to all other parties. 2. Parties can later recover any message that was sent in such a way (using a unique identifier). We write “**bulletin-store** x ” for deterministically computing the identifier for x (e. g., using a function $\text{id}(x)$) based on the step in the protocol, sending x with the identifier to $\mathcal{F}_{\text{bulletin}}$. “**Bulletin-load** x ” is defined analogously: The receiving party computes the identifier, sends a request with the identifier to $\mathcal{F}_{\text{bulletin}}$, and $\mathcal{F}_{\text{bulletin}}$ answers with the respective value (as soon as it is available). See Section 5.1 and Table 5.1 for more details.

$\mathcal{F}_{\text{rand-cipher}}$ makes the parties “agree” on a ciphertext. The corresponding plaintext has the following properties: 1. It is distributed uniformly at random and 2. it is unknown to all parties. These properties hold only if less than t parties are corrupted. Otherwise, the plaintext is effectively chosen by the adversary, e. g., it is known and not necessarily random. If at least t parties act maliciously, honest parties also know that $|\mathbb{P}_{\text{corrupt}}| \geq t$ as $\mathbb{P}_{\text{malicious}} \subseteq \mathbb{P}_{\text{corrupt}}$. In this case, honest parties output \perp and audits detect that honest parties would abort. $\mathcal{F}_{\text{rand-cipher}}$ can be found in Figure 4.1.

$\mathcal{F}_{\text{com-cipher}}$ generates a random value for each honest party. A cipher and a commitment for this value are computed and the random value is only given to the party it was generated for. For corrupted parties, this value is chosen by the adversary. A cipher and commitment are selected by the adversary as well, but the commitment has to correspond to the cipher, if the cipher and commitment are not generated for the same plaintext, the corrupted party is detected and classified as malicious. Finally, each party gets the ciphers and commitments of all parties. $\mathcal{F}_{\text{com-cipher}}$ can be found in Figure 4.2.

\mathcal{F}_{key} models key generation, distribution, and threshold decryption. In the initialization, each party gets the public keys and their own private key. If enough parties act honestly, the honest parties get the correct decryption of a cipher. Otherwise, the parties output \perp to signal that the decryption was not possible. The adversary always learns the result of the decryption with the following reasoning:

¹In the proof, we could assume a programmable random oracle that generates the public keys in the ideal world. The simulator should program the oracle to give the same keys as in the internal simulation.

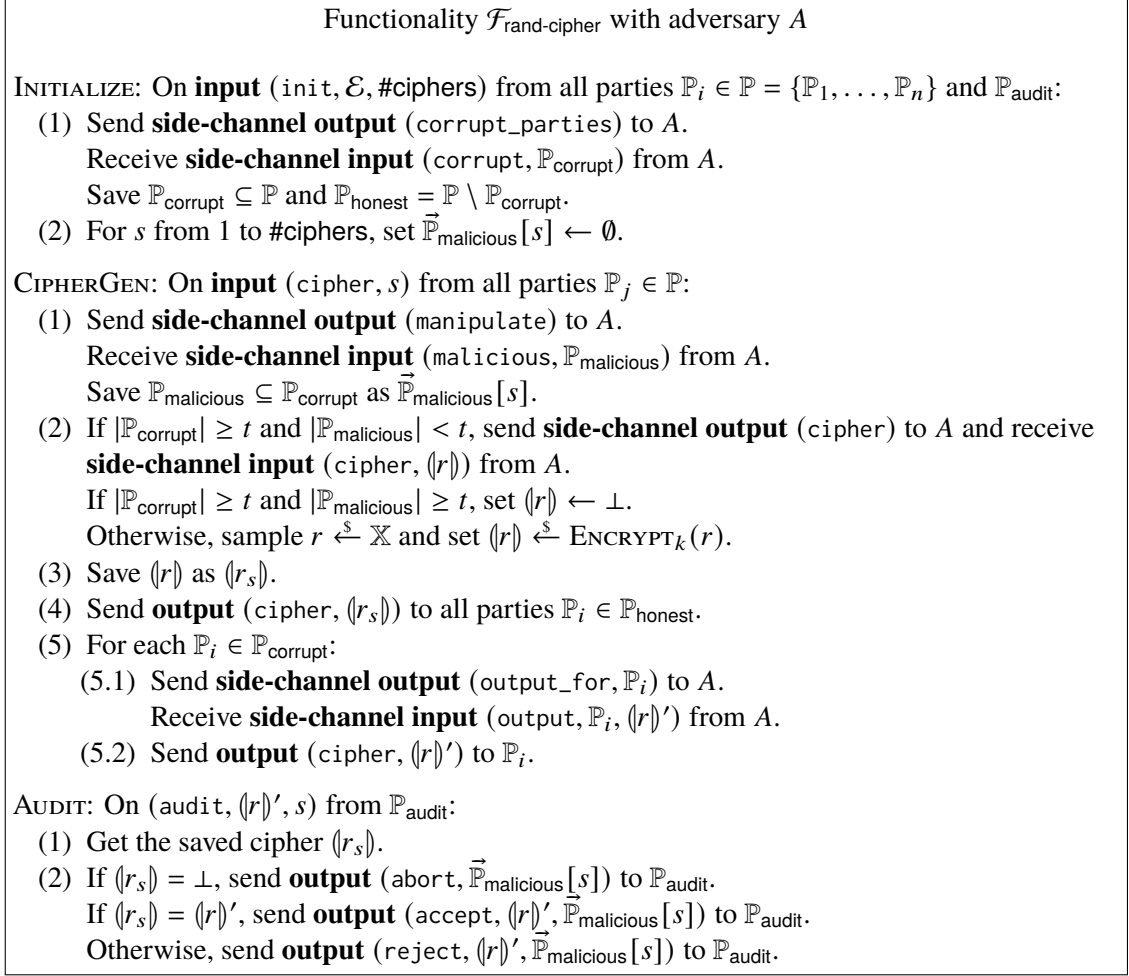


Figure 4.1: Functionality to generate a random cipher.

We assume that a real adversary can control the timing of the corrupted parties. This means, they can wait for all honest parties to broadcast their decryption share. These shares can be combined with the shares of the corrupted parties to reconstruct the plaintext. Finally, the adversary can choose to follow the protocol and send the correct shares (in this case, the honest parties learn the plaintext) or (if less than t parties act honestly) send wrong shares and make it impossible for honest parties to reconstruct the plaintext. During the decryption, misbehaving parties are revealed to anyone. This enables auditing the functionality. \mathcal{F}_{key} can be found in Figure 4.3.

The functionalities use the following inputs for a circuit with $\#\text{inputs}$ inputs and $\#\text{multiplications}$ multiplications that is computed by n (compute) parties:

$$\begin{aligned} \#\text{ciphers} &= (\#\text{inputs} + 3 \cdot \#\text{multiplications}) \cdot (t - 1) + 2 \cdot \#\text{multiplications} \\ \#\text{commitments} &= \#\text{inputs} + 3 \cdot \#\text{multiplications} \\ \#\text{decryptions} &= (\#\text{inputs} + 3 \cdot \#\text{multiplications}) \cdot n. \end{aligned}$$

SHARE (see Figure 4.5b) performs n decryptions, needs $t - 1$ random ciphers, and a commitment from each party. It is called for each input and each part of the multiplication triple, e. g., for a , b , and c for a triple (a, b, c) . Additionally, two of the tree parts (a and b) of each multiplication

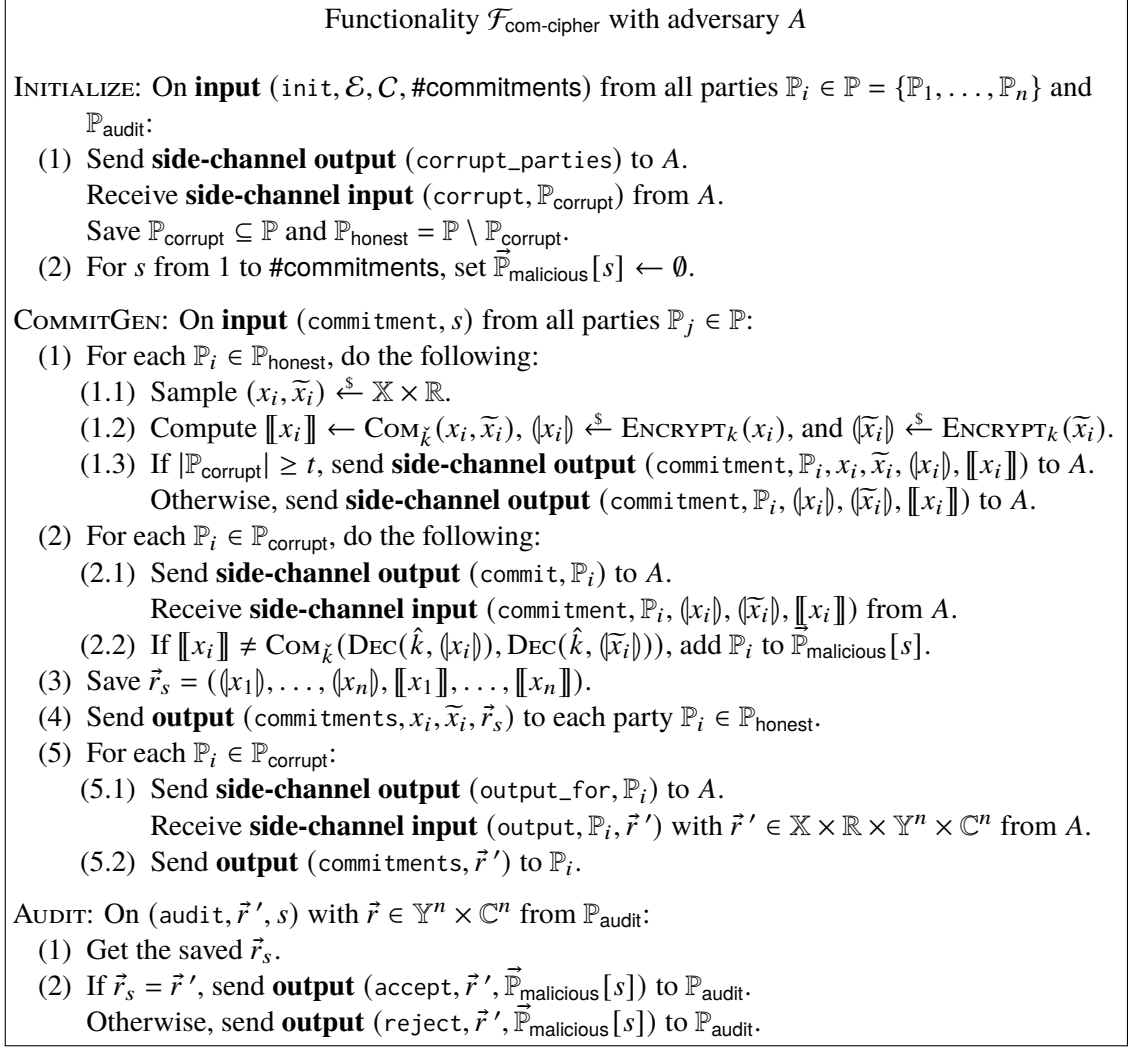


Figure 4.2: Functionality to generate a committed cipher.

triple are generated as random cipher. This gives us the number of rounds for $\mathcal{F}_{\text{rand-cipher}}$, $\mathcal{F}_{\text{com-cipher}}$, and \mathcal{F}_{key} . Now we come to our main protocol – the MPC protocol to evaluate arbitrary arithmetic circuits.

4.2 Main Protocol

We define \mathcal{F}_{MPC} (see Figure 4.4) as a functionality that allows evaluating arbitrary arithmetic circuits in an MPC way (for a more details on circuits, see Section 5.9). Here, n compute parties $\mathbb{P}_i \in \mathbb{P}$ evaluate a circuit \mathcal{C} with inputs from $\#\text{inputs}$ input parties $\mathbb{P}_{\text{input}, i} \in \mathbb{P}_{\text{input}}$. The gates of the circuit are arithmetic addition and multiplication operations (i. e., **ADD** and **MUL** gates) in addition to public output (**OPEN**) gates. The adversary will always learn the result of the computation (similarly to learning the plaintext in \mathcal{F}_{key}) and can try to manipulate the result for the other parties. Honest parties will always detect this and compensate (if at least t parties do not misbehave, the result can

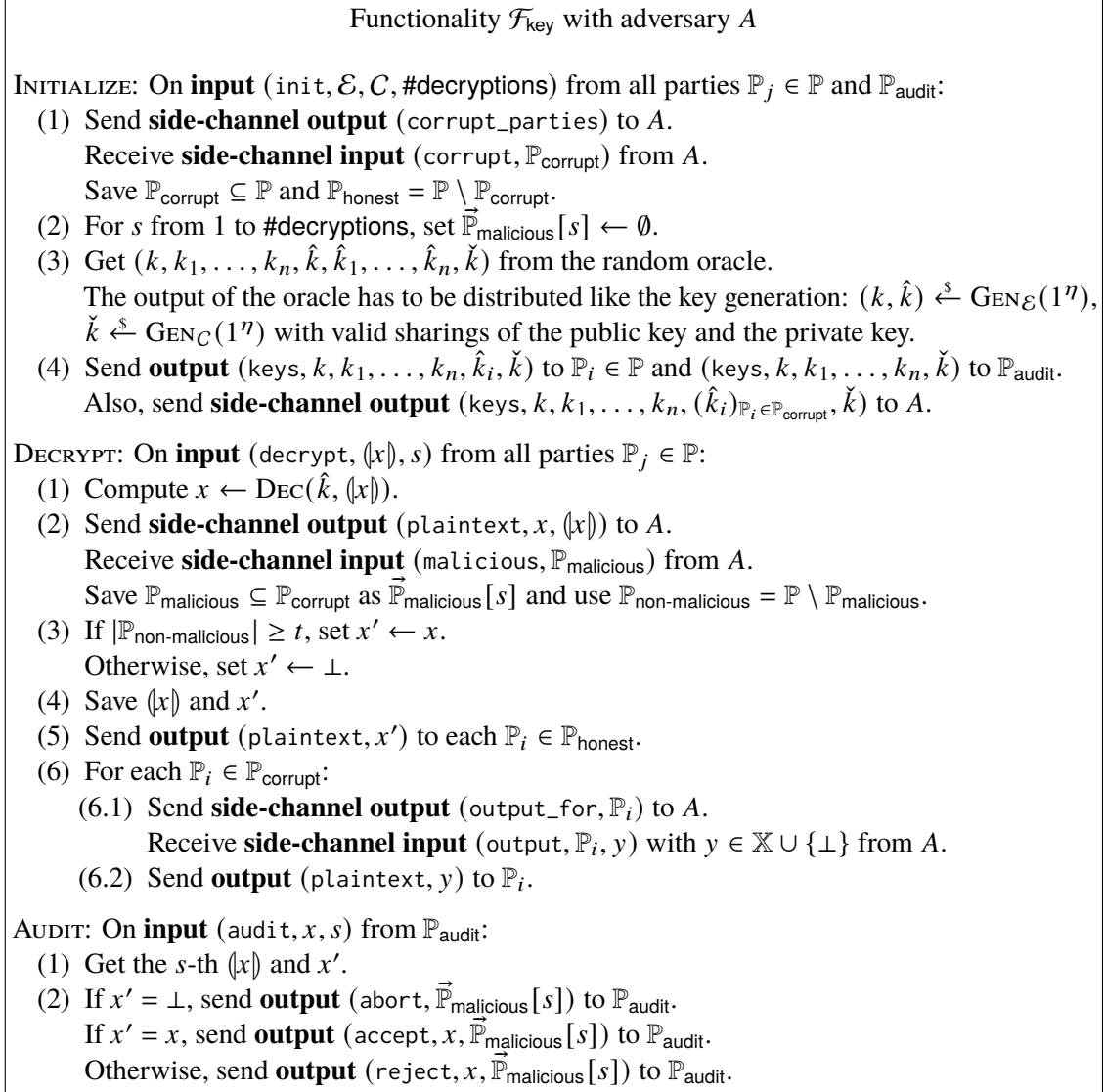


Figure 4.3: Functionality for threshold decryption.

still be computed) or abort (if less than t parties do not misbehave). The inputs stay hidden from the adversary, unless at least t compute parties are corrupted, or the corresponding input party is corrupted. For honest input parties (with $|\mathbb{P}_{\text{corrupt}}| < t$), the functionality sends the encrypted input to the adversary. If we compare this to the functionality for auditable SPDZ [12], we see that they use a different strategy: They do not reveal anything about the inputs. They can do this, as they use dummy inputs in the simulation for honest parties (i. e., the value 0) and adjust the commitments later, so they match with the output of the ideal functionality. We cannot do something like this if we want to support perfectly binding commitment schemes. Instead, we use this “hack” of sending the encrypted input to the simulator.

Π_{MPC} is our proposed protocol to realize this auditable multi-party computation. The description of each parties’ view on the protocol is found in Figure 4.5: Figure 4.5a for input parties, Figures 4.5b and 4.5c for compute parties, and Figure 4.5d for auditing parties. The protocol uses Shamir secret

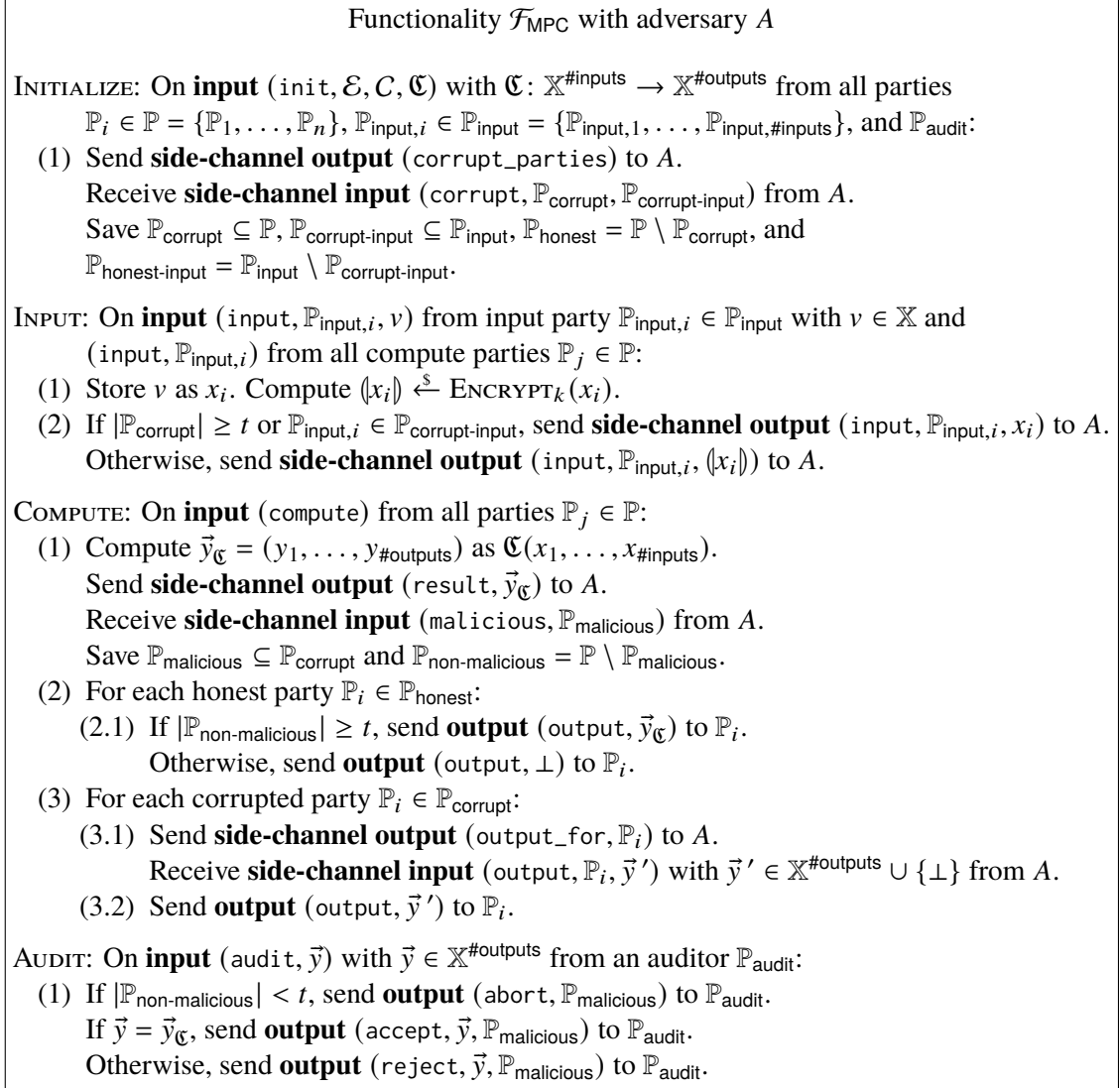


Figure 4.4: Functionality for an accountable multi-party computation.

sharing (unlike most SPDZ-like protocols). We do this because it is more flexible and enables the computation of the result, even if (a few) parties are misbehaving. A version with full-threshold secret sharing is given in Section 5.8.

4.2.1 Design Rationale

Our protocol is heavily inspired by SPDZ [49] and some of its variants [12, 44]. One reason for this is the low complexity and high performance of SPDZ (in the online phase). In fact, we manage to add accountability with the same communication complexity of the auditable version of SPDZ (see Table 7.2). Doing so increases the computational complexity. We evaluate the arithmetic circuit n additional times on commitments (Sections 5.7 and 5.8 include some improvements on that). It seems that we cannot achieve individual accountability without checking the computation for each

<p>Protocol Π_{MPC} from the view of $\mathbb{P}_{\text{input},i}$</p> <p>INITIALIZE: On input $(\text{init}, \mathcal{E}, C, \mathfrak{C})$ from E:</p> <p>(1) <i>Static corruption wrapper is inserted here.</i></p> <p>INPUT: On input $(\text{input}, \mathbb{P}_{\text{input},i}, v)$ from E:</p> <p>(1) Compute $\langle x_i \rangle \xleftarrow{\\$} \text{ENCRYPT}_k(v)$.</p> <p>(2) Bulletin-store $\langle x_i \rangle$.</p>
--

(a) Protocol from the view of input parties.

Figure 4.5: Protocol for an accountable multi-party computation.

party (in case a misbehaving party has to be identified; a less expensive check can verify that the result was correct or not). This leads to $O(n \cdot |\mathfrak{C}|)$ extra work (in the worst case). Our proposed protocol does this work in any case (to be conceptually simpler), but (the already mentioned) improvements can be used to speed up the computation if no party misbehaves.

To allow more use-cases, we employ Shamir secret sharing and support any (somewhat homomorphic) encryption scheme and any (additive homomorphic) commitment scheme. If full corruption (or a dishonest majority) is to be expected, one can use an optimized version with full-threshold secret sharing as described in Section 5.8. In that case, our protocol is almost the same as the accountable variant of SPDZ [44] (described in Section 3.1.3; we compare our protocol to it in Section 7.2).

4.2.2 Overview

Now, we will give a short overview of our protocol. As already hinted by the input phase of \mathcal{F}_{MPC} , input parties encrypt their inputs and publish the encrypted inputs. The compute parties convert these ciphers then to what we call views of shared values. A view

$$[x]_i = (\langle x \rangle_i, \tilde{x}_i, \llbracket \langle x \rangle_1 \rrbracket, \dots, \llbracket \langle x \rangle_n \rrbracket)$$

of compute party \mathbb{P}_i for the shared value x consists of $n + 2$ values: the share $\langle x \rangle_i$ of the party, a commitment-randomness \tilde{x}_i , and n commitments where each party's share and randomness are the corresponding decommitment, e. g.,

$$(4.1) \quad \llbracket \langle x \rangle_j \rrbracket = \text{COM}_{\tilde{x}}(\langle x \rangle_j, \tilde{x}_j)$$

for each \mathbb{P}_j . The views are computed with the `SHARE` sub-protocol (see Figure 4.5b). Here, a ciphertext-polynomial p is constructed for secret sharing. The polynomial is evaluated for each party \mathbb{P}_j to get a cipher $\langle \langle x \rangle_j \rangle$ for their share $\langle x \rangle_j$. This is then masked with a cipher $\langle r_j \rangle$. Special about this cipher is that only \mathbb{P}_j knows the corresponding plaintext r_j , but everyone knows a commitment $\llbracket r_j \rrbracket$ for the same plaintext – this is ensured by $\mathcal{F}_{\text{com-cipher}}$. This means, everyone can compute a commitment for each party's share. The views are correct if

$$(4.2) \quad \llbracket r_j \rrbracket = \text{COM}_{\tilde{x}}(r_j, \tilde{r}_j)$$

Protocol Π_{MPC} from the view of \mathbb{P}_i

subprotocol SHARE: $\langle x \rangle \mapsto [x]_i$:

- (1) Get the next $(r_i, \tilde{r}_i, \langle r_1 \rangle, \dots, \langle r_n \rangle, \llbracket r_1 \rrbracket, \dots, \llbracket r_n \rrbracket)$ generated by $\mathcal{F}_{\text{com-cipher}}$.
- (2) Get the next $t - 1$ ciphers $\langle a_s \rangle$ generated by $\mathcal{F}_{\text{rand-cipher}}$.
- (3) Let $p: l \mapsto \langle x \rangle \oplus \bigoplus_{s=1}^{t-1} l^s \otimes \langle a_s \rangle$.
- (4) For $\mathbb{P}_j \in \mathbb{P}$, compute the encrypted shares and commitments:
 - (4.1) $\langle \langle x \rangle_j \rangle \leftarrow p(j)$ and $\langle m_j \rangle \leftarrow \langle \langle x \rangle_j \rangle \oplus \langle r_j \rangle$.
 - (4.2) Decrypt $\langle m_j \rangle$ to m_j using \mathcal{F}_{key} , audit it, and remove cheaters from \mathbb{P}_{curr} .
 - (4.3) $\llbracket \langle x \rangle_j \rrbracket \leftarrow \text{COM}_{\tilde{\kappa}}(m_j, \tilde{0}) \boxplus \llbracket r_j \rrbracket$.
- (5) Compute share and randomness: $\langle x \rangle_i \leftarrow m_i - r_i$ and $\tilde{x}_i \leftarrow -\tilde{r}_i$.
- (6) Return $[x]_i = (\langle x \rangle_i, \tilde{x}_i, \llbracket \langle x \rangle_1 \rrbracket, \dots, \llbracket \langle x \rangle_n \rrbracket)$.

INITIALIZE: On **input** $(\text{init}, \mathcal{E}, \mathcal{C}, \mathfrak{C})$ from E :

- (1) *Static corruption wrapper is inserted here.*
- (2) Set $\mathbb{P}_{\text{curr}} \leftarrow \mathbb{P}$. On updates of \mathbb{P}_{curr} : If $|\mathbb{P}_{\text{curr}}| < t$, do not modify it.
- (3) Initialize $\mathcal{F}_{\text{bulletin}}$, $\mathcal{F}_{\text{rand-cipher}}$, $\mathcal{F}_{\text{com-cipher}}$, and \mathcal{F}_{key} .
- (4) For s from 1 to #ciphers, do the following:
 - (4.1) Send (cipher, s) as **input** to $\mathcal{F}_{\text{rand-cipher}}$, receive **output** $(\text{cipher}, \langle r \rangle)$, audit it, and remove cheaters from \mathbb{P}_{curr} .
- (5) For s from 1 to #commitments, do the following:
 - (5.1) Send $(\text{commitment}, s)$ as **input** to $\mathcal{F}_{\text{com-cipher}}$, receive **output** $(\text{commitments}, r_i, \tilde{r}_i, \langle r_1 \rangle, \dots, \langle r_n \rangle, \llbracket r_1 \rrbracket, \dots, \llbracket r_n \rrbracket)$, audit it, and remove cheaters from \mathbb{P}_{curr} .
- (6) For s from 1 to #multiplications, do the following:
 - (6.1) Get the next two cipher $\langle a \rangle$ and $\langle b \rangle$ generated by $\mathcal{F}_{\text{rand-cipher}}$.
 - (6.2) Compute $\langle c \rangle \leftarrow \langle a \rangle \odot \langle b \rangle$.
 - (6.3) Run SHARE with $\langle a \rangle$, $\langle b \rangle$, and $\langle c \rangle$ to get $[a]_i$, $[b]_i$, and $[c]_i$. Save the triple $([a]_i, [b]_i, [c]_i)$.

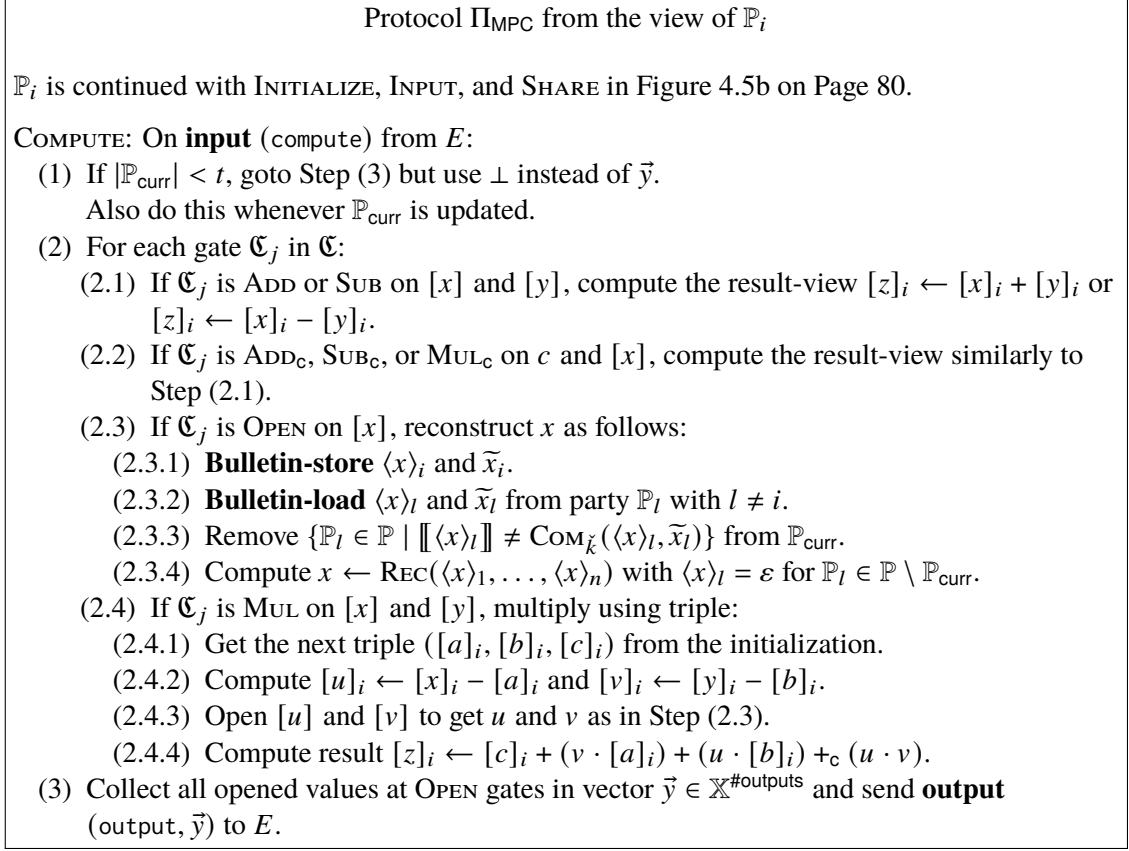
INPUT: On **input** $(\text{input}, \mathbb{P}_{\text{input}, j})$ from E :

- (1) **Bulletin-load** $\langle x_j \rangle$ from $\mathbb{P}_{\text{input}, j}$.
- (2) Run SHARE with $\langle x_j \rangle$ to get $[x_j]_i$ and save $[x_j]_i$.

\mathbb{P}_i is continued with COMPUTE in Figure 4.5c on Page 81.

(b) Protocol from the view of compute parties: initialization and input.

Figure 4.5: Protocol for an accountable multi-party computation.



(c) Protocol from the view of compute parties: evaluation of the arithmetic circuit.

Figure 4.5: Protocol for an accountable multi-party computation.

and

$$(4.3) \quad r_j = \text{DEC}(\hat{k}, \langle r_j \rangle).$$

The first property is ensured by $\mathcal{F}_{\text{com-cipher}}$, the second one by $\mathcal{F}_{\text{com-cipher}}$ together with \mathcal{F}_{key} . This means the commitments are correct:

$$\begin{aligned} \llbracket \langle x \rangle_j \rrbracket &= \text{COM}_{\tilde{k}}(m_j, \tilde{0}) \boxplus \llbracket r_j \rrbracket \\ &\stackrel{(4.2)}{=} \text{COM}_{\tilde{k}}(\text{DEC}(\hat{k}, \langle \langle x \rangle_j \rangle) \oplus \langle r_j \rangle) - r_j, -\tilde{r}_j) \\ &\stackrel{(4.3)}{=} \text{COM}_{\tilde{k}}(\langle x \rangle_j + r_j - r_j, -\tilde{r}_j) \\ &= \text{COM}_{\tilde{k}}(\langle x \rangle_j, -\tilde{r}_j). \end{aligned}$$

To ensure Equation (4.1), SHARE correctly lets each \mathbb{P}_i set their randomness \tilde{x}_i to $-\tilde{r}_i$.

To evaluate an arithmetic circuit, we have to be able to consistently operate on the (correctly computed) input shares. In particular, for valid views

$$[x]_i = (\langle x \rangle_i, \tilde{x}_i, \llbracket \langle x \rangle_1 \rrbracket, \dots, \llbracket \langle x \rangle_n \rrbracket) \quad [y]_i = (\langle y \rangle_i, \tilde{y}_i, \llbracket \langle y \rangle_1 \rrbracket, \dots, \llbracket \langle y \rangle_n \rrbracket)$$

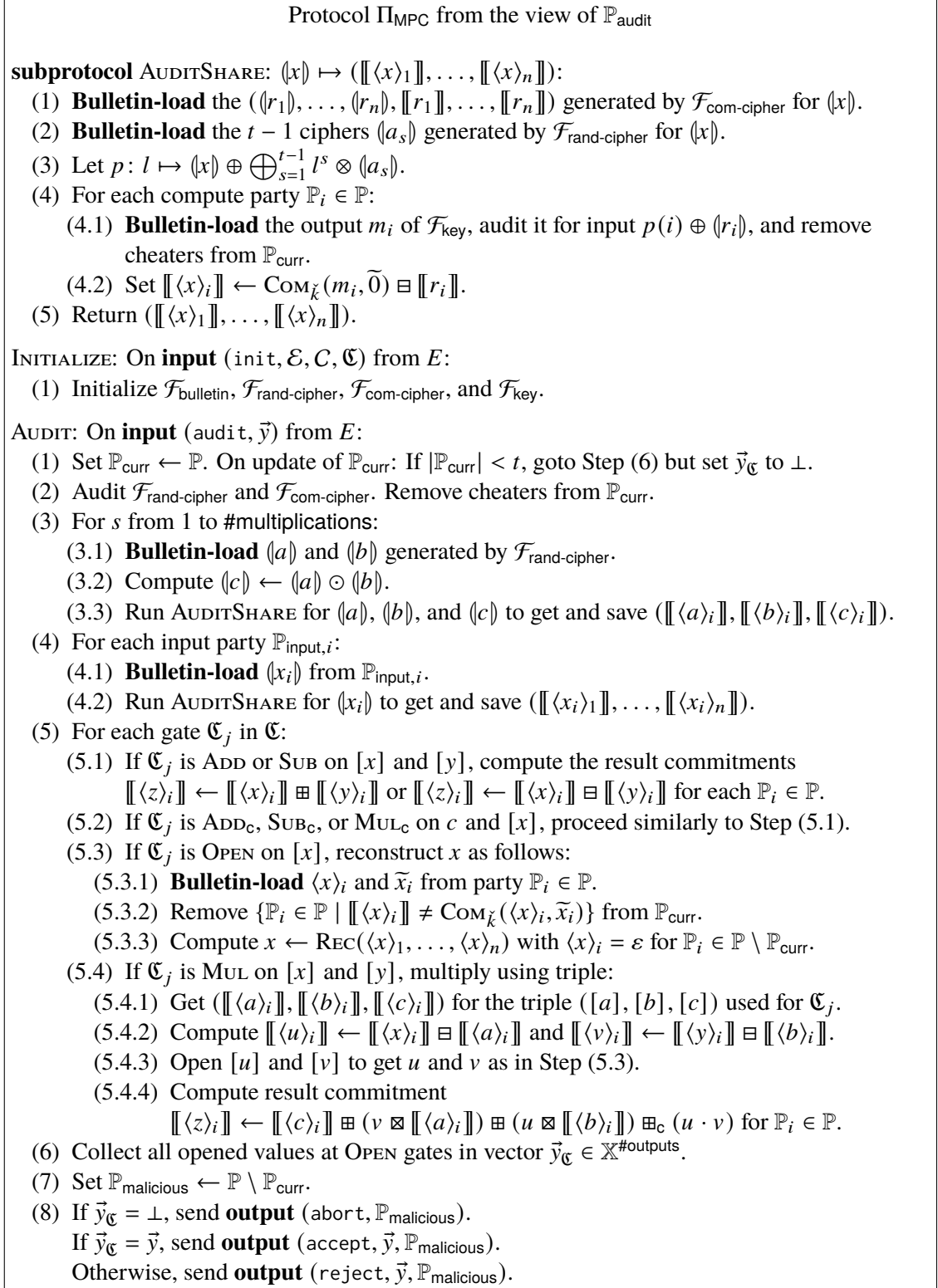


Figure 4.5: Protocol for an accountable multi-party computation.

and constants $a \in \mathbb{X}$, we need means to do additive homomorphic operations:

$$(4.4) \quad [x]_i + [y]_i = [x + y]_i$$

$$(4.5) \quad a \cdot [x]_i = [a \cdot x]_i$$

$$(4.6) \quad [x]_i +_c a = [x + a]_i.$$

Addition (Equation (4.4)) and multiplication-with-constants (Equation (4.5)) are straight forward:

$$\begin{aligned} [x]_i + [y]_i &= (\langle x \rangle_i + \langle y \rangle_i, \tilde{x}_i + \tilde{y}_i, & \llbracket \langle x \rangle_1 \rrbracket \boxplus \llbracket \langle y \rangle_1 \rrbracket, \dots, & \llbracket \langle x \rangle_n \rrbracket \boxplus \llbracket \langle y \rangle_n \rrbracket) \\ a \cdot [x]_i &= (a \cdot \langle x \rangle_i, a \times \tilde{x}_i, & a \boxtimes \llbracket \langle x \rangle_1 \rrbracket, \dots, & a \boxtimes \llbracket \langle x \rangle_n \rrbracket) \end{aligned}$$

and correctness follows from the additive homomorphic properties of the secret sharing and commitment schemes. Addition-of-constants (Equation (4.6)) is possible as well:

$$[x]_i +_c a = (\langle x \rangle_i + a, \tilde{x}_i, \llbracket \langle x \rangle_1 \rrbracket \boxplus \text{Com}_{\tilde{k}}(a, \tilde{0}), \dots, \llbracket \langle x \rangle_n \rrbracket \boxplus \text{Com}_{\tilde{k}}(a, \tilde{0}))$$

because we can add constants to shares with Shamir secret sharing. With full-threshold secret sharing, we would only add the constant to the share of a single party and only add $\text{Com}_{\tilde{k}}(a, \tilde{0})$ to the corresponding commitment. Subtractions of two views or subtractions-of-constants can be done with the corresponding subtraction operations. We perform multiplications of views with Beaver triples [14] (see Figure 5.3 for a visualization of a multiplication using such a triple). For this, we need views $[a]_i, [b]_i, [c]_i$ with

$$(4.7) \quad c = a \cdot b.$$

This is guaranteed by the way the triples are generated (we use an SHE scheme and multiply two ciphers, then we use `SHARE` to generate the views).² To multiply views, we proceed as in Figure 4.5c. Then we have

$$[z]_i = (v \cdot [a]_i) + (u \cdot [b]_i) + [c]_i +_c (u \cdot v)$$

and – assuming that opening values is correct (we discuss this next) – we can use the additive homomorphic operations on views to get

$$\begin{aligned} [z]_i &= ((y - b) \cdot [a]_i) + ((x - a) \cdot [b]_i) + [c]_i +_c ((x - a) \cdot (y - b)) \\ &\stackrel{(4.6)}{=} ((y - b) \cdot [a]_i) + ((x - a) \cdot [b]_i) + [c + (x \cdot y) - (x \cdot b) - (y \cdot a) + (a \cdot b)]_i \\ &\stackrel{(4.5)}{=} [(y \cdot a) - (b \cdot a)]_i + [(x \cdot b) - (a \cdot b)]_i + [c + (x \cdot y) - (x \cdot b) - (y \cdot a) + (a \cdot b)]_i \\ &\stackrel{(4.4)}{=} [-(b \cdot a) + c + (x \cdot y)]_i \\ &\stackrel{(4.7)}{=} [x \cdot y]_i. \end{aligned}$$

“Opening” comes into play when an output should be revealed and in multiplication (see above and Figure 4.5c). For this, each party publishes their private part of the view, e. g., $\langle x \rangle_i$ and \tilde{x}_i . Every party can then reconstruct

$$x = \text{REC}(\langle x \rangle_1, \dots, \langle x \rangle_n)$$

²Views for a and b might be generated more efficiently as in SPDZ or similar protocols [12, 49].

but published shares might be wrong. For this, the commitments come into play. The justification of commitments is that, when doing operations on views (as described above), everyone also has a commitment for the other parties' shares. When their shares are now revealed at an opening gate in the arithmetic circuit, everyone can check if the revealed share fits to the locally computed commitment. With the binding property of commitment schemes, nobody should be able to convince parties of wrong shares. In turn, if wrong shares are detected, they are not used for reconstruction.

Finally, we can audit the circuit only using the commitment – the commitments could be considered the public part of a view $[x]_i$ – and the auditing capabilities of the used functionalities like \mathcal{F}_{key} (which are implemented with zero-knowledge proofs in Section 4.3).

4.2.3 UC-Security

To prove security properties later in Chapter 6, we first prove that our protocol and the functionality to evaluate arithmetic circuits are equivalent. More precisely, we prove the following.

Theorem 4.1 (Π_{MPC} Realizes \mathcal{F}_{MPC}).

Π_{MPC} UC-realizes \mathcal{F}_{MPC} in the $(\mathcal{F}_{\text{rand-cipher}}, \mathcal{F}_{\text{com-cipher}}, \mathcal{F}_{\text{key}}, \mathcal{F}_{\text{bulletin}})$ -hybrid (simplified) model with static corruption.

Proof. To show this, we show that

$$|\Pr[(E \parallel D \parallel \Pi_{\text{MPC}}^{\text{ideal}})(1^n) = \text{real}] - \Pr[(E \parallel S_{\text{MPC}} \parallel \Pi_{\text{MPC}}^{\text{wrap}})(1^n) = \text{real}]|$$

is negligible for all PPT environments E . S_{MPC} is described in Figure 4.6 and the interactions that E has to distinguish are sketched in Figure 4.7. There, an example of an interaction with three compute parties and two input parties is depicted. Dummy parties (e. g., parties that only forward their messages) are indicated by dashed circles.

The intuition, why the probabilities are indistinguishable, is the following: S_{MPC} runs an actual instance of Π_{MPC} with only small modifications. This means, interactions from the point of view of E with S_{MPC} should be almost identical to interactions with D and $\Pi_{\text{MPC}}^{\text{wrap}}$. The cases, when E gives inputs or receives outputs from the protocol have to be considered carefully, as S_{MPC} is unable to receive these inputs or send outputs directly. In the following, each part of the online phase is discussed separately.

INITIALIZE. Initialization begins with E sending $(\text{init}, \mathcal{E}, C, \mathfrak{C})$ to all parties. Next, the corrupted parties are set. This is the same for $(E \parallel D \parallel \Pi_{\text{MPC}}^{\text{ideal}})$ and $(E \parallel S_{\text{MPC}} \parallel \Pi_{\text{MPC}}^{\text{wrap}})$ as this is enforced by the static corruption wrapper. The rest of the initialization is the same as well, because S_{MPC} just runs the initialization of Π_{MPC} .

INPUT. Here, E sends a message to an input party $\mathbb{P}_{\text{input},i}$ that should give their input to the protocol and all compute parties to instruct them to generate shares for it. This is repeated for all inputs of the arithmetic circuit. For a dishonest $\mathbb{P}_{\text{input},i}$ or in the case of at least t corrupted compute parties, S_{MPC} can perfectly simulate the protocol as the plain input value for $\mathbb{P}_{\text{input},i}$ is revealed through \mathcal{F}_{MPC} . Otherwise it only gets the cipher (x_i) for the input. However – as $\mathbb{P}_{\text{input},i}$ is honest in this

case – the cipher (x_i) is indistinguishable from a cipher that would be computed by $\mathbb{P}_{\text{input},i}$ as the way of computing them is the same (cf. Step (1) of INPUT in Figure 4.4 and Step (1) of INPUT in Figure 4.5a).

COMPUTE. In the computation step, E sends (compute) to all compute parties to instruct them to compute the circuit. Π_{MPC} gets the result of the computation from \mathcal{F}_{MPC} and simulates the computation of Π_{MPC} . At this point, it is important to correctly identify the malicious parties in the simulation (the importance for auditing will be discussed later). If the malicious parties were identified correctly, the output of the honest parties is the same: They send \perp if too many parties act maliciously. Otherwise, enough shares are available to compute the circuit correctly (and wrong shares are detected, so the result cannot differ from the one obtained by using the functionality). The static corruption wrapper ensures that all corrupted parties ask for their outputs. These requests can be forwarded and S_{MPC} acts like D .

AUDIT. \mathcal{F}_{MPC} allows no interaction with an adversary, so S_{MPC} cannot change the output of auditing (but $\mathbb{P}_{\text{malicious}}$ will be returned by \mathcal{F}_{MPC} as given by S_{MPC} in COMPUTE). This means, the output of \mathcal{F}_{MPC} and Π_{MPC} for AUDIT have to be indistinguishable, which is the case if the malicious parties are correctly identified with overwhelming probability. As discussed before, this is also important for an indistinguishable output of COMPUTE. The functionalities $\mathcal{F}_{\text{rand-cipher}}$, $\mathcal{F}_{\text{com-cipher}}$, and \mathcal{F}_{key} are correctly auditable. This leaves only one check for the auditor: detecting wrongly open commitments. For perfectly binding commitment schemes, this is easy: No cheating is possible without being detected. For computationally binding commitment schemes, we have to argue why cheating is detected with overwhelming probability.

Cheating would be the following: Broadcasting $\langle x \rangle'_i$ and \tilde{x}'_i with $\langle x \rangle_i \neq \langle x \rangle'_i$ such that

$$\llbracket \langle x \rangle_i \rrbracket = \text{COM}_{\tilde{k}}(\langle x \rangle_i, \tilde{x}_i) = \text{COM}_{\tilde{k}}(\langle x \rangle'_i, \tilde{x}'_i).$$

This is equivalent to winning the security game for the binding property of commitment schemes. For this, an adversary A_{binding} on $\mathfrak{G}_{C, A_{\text{binding}}}^{\text{binding}}$ for the commitment scheme C can be constructed from an environment E that can distinguish $(E \parallel D \parallel \Pi_{\text{MPC}}^{\text{ideal}})$ and $(E \parallel S_{\text{MPC}} \parallel \Pi_{\text{MPC}}^{\text{wrap}})$ (as discussed above, these are indistinguishable if cheating is detected, e. g., E is able to cheat and stay undetected): A_{binding} runs $(E \parallel S_{\text{MPC}} \parallel \Pi_{\text{MPC}}^{\text{wrap}})$ and outputs $(\llbracket \langle x \rangle_i \rrbracket, \langle x \rangle_i, \tilde{x}_i, \langle x \rangle'_i, \tilde{x}'_i)$ for which undetected cheating was possible. A_{binding} can know this, as it can compute all shares $\langle x \rangle_i$ for opened values in an honest way. $\langle x \rangle'_i$ is obtained by comparing the actually opened shares to the honestly computed ones and taking the $\langle x \rangle'_i$ with $\langle x \rangle_i \neq \langle x \rangle'_i$ but with the same commitment (this share exists as E cheated and stayed undetected).

Running $(E \parallel S_{\text{MPC}} \parallel \Pi_{\text{MPC}}^{\text{wrap}})$ to win the binding-game is possible as it can be done in polynomial time and it does not use any backdoor to the commitment scheme that would not be given in the binding-game. This is a contradiction to the assumption that C is computationally binding,³ so undetected cheating is only possible with negligible probability, which in turn implies that distinguishing $(E \parallel D \parallel \Pi_{\text{MPC}}^{\text{ideal}})$ and $(E \parallel S_{\text{MPC}} \parallel \Pi_{\text{MPC}}^{\text{wrap}})$ is also only possible with negligible probability. \square

³If the commitment scheme is perfectly binding, no cheater can stay undetected.

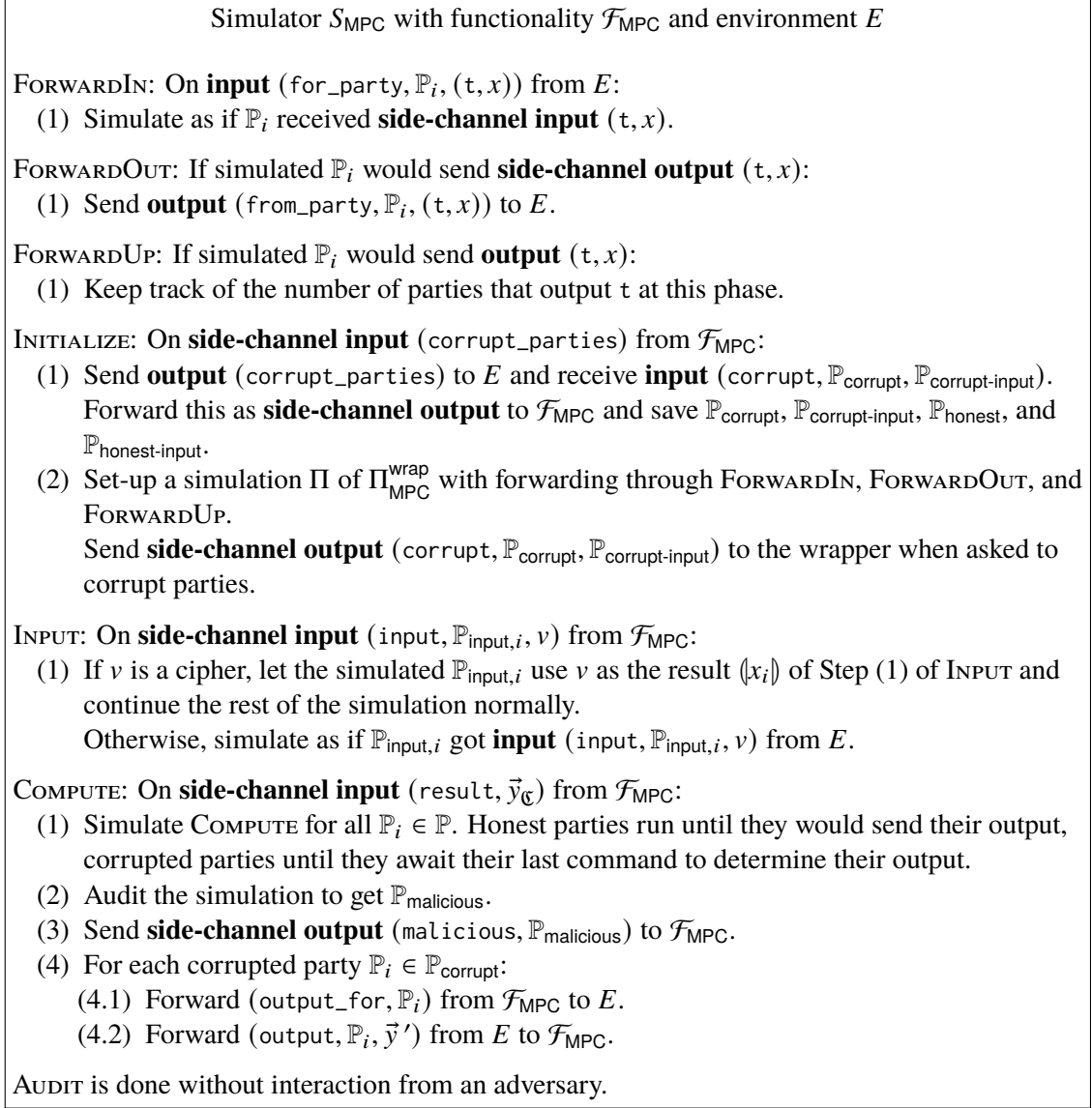


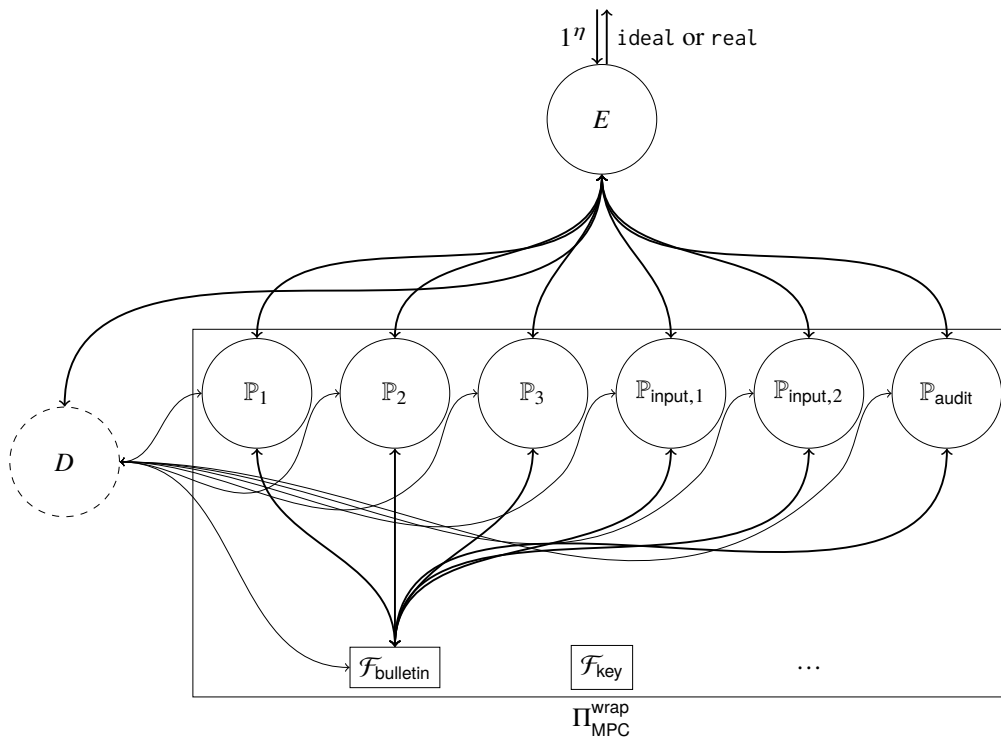
Figure 4.6: Simulator for protocol for an accountable multi-party computation.

4.3 Protocols for the Prerequisite Functionalities

In this section, we will discuss what protocols can be used to implement $\mathcal{F}_{\text{rand-cipher}}$, $\mathcal{F}_{\text{com-cipher}}$, and \mathcal{F}_{key} . As mentioned before, details on $\mathcal{F}_{\text{bulletin}}$ are left out, but we explain that it can be implemented in Section 5.1. Note that the prerequisite functionalities and protocols all work in rounds. This is done, so auditing runs can be associated with rounds, i. e., with ciphers that get decrypted in \mathcal{F}_{key} . A more sophisticated version of these would process multiple values in batches and use proofs that prove the required properties for the whole batch. This is not done here for simplicity but can be done similarly to existing protocols [12, 49].



(a) Example of the ideal world.



(b) Example of the real world.

Figure 4.7: Protocol interactions to distinguish for an environment.

4.3.1 Protocol to Generate a Random Cipher

Here, we present an alternative functionality $\mathcal{F}_{\text{rand-cipher}'}$ instead of $\mathcal{F}_{\text{rand-cipher}}$ and a protocol to implement it. The main reason is that $\mathcal{F}_{\text{rand-cipher}}$ seems impossible to simulate (for $|\mathbb{P}_{\text{corrupt}}| < t$). If we want to generate a random cipher like in SPDZ by adding ciphers from all parties up, we end up with a random cipher if less than t parties are corrupted. But we have to be sure that the ciphers of corrupted parties are not chosen based on the ones of honest parties. Independence of inputs makes sure that the corrupted parties cannot manipulate the result to a cipher with known plaintext content. To stop corrupted parties from doing so, a ZK proof of plaintext-knowledge could be employed. But this is not simulatable, as the simulator is lacking information as discussed in Theorem 2.16.

To circumvent this disadvantage of the simulator, $\mathcal{F}_{\text{rand-cipher}'}$ actually provides the simulator with a ZK proof that can simply be forwarded. What is left to show is that $\mathcal{F}_{\text{rand-cipher}'}$ and $\mathcal{F}_{\text{rand-cipher}}$ behave the same. For the case of at least t corrupted parties, they are designed to be the same. In the other case, the cipher

$$\langle r_s \rangle = \bigoplus_{\mathbb{P}_i \in \mathbb{P} \setminus \tilde{\mathbb{P}}_{\text{malicious}}[s]} \langle x_i \rangle$$

is the output of the protocol. The plaintext corresponding to $\langle r_s \rangle$ is distributed uniformly at random iff. all $\langle x_i \rangle$ for $\mathbb{P}_i \in \mathbb{P}_{\text{corrupt}} \setminus \tilde{\mathbb{P}}_{\text{malicious}}[s]$ are chosen independently of all $\langle x_j \rangle$ for $\mathbb{P}_j \in \mathbb{P}_{\text{honest}}$.⁴ The ZK proofs are used for this: If $\langle x_i \rangle$ of a corrupted party is to be chosen dependently on an honest $\langle x_j \rangle$, then x_i depends on x_j in the same way (i. e., if $\langle x_i \rangle = 2 \otimes \langle x_j \rangle \oplus \text{ENC}_k(4, \tilde{r})$, $x_i = 2 \cdot x_j + 4$). This means, $\langle x_i \rangle$ can only successfully be chosen dependently on $\langle x_j \rangle$ if x_j is known. Otherwise \mathbb{P}_i will fail to prove knowledge of the plaintext x_i . $\mathcal{F}_{\text{rand-cipher}'}$ and $\Pi_{\text{com-cipher}'}$ can be found in Figures 4.8 and 4.9.

Theorem 4.2 ($\Pi_{\text{rand-cipher}'}$ Realizes $\mathcal{F}_{\text{rand-cipher}'}$).

$\Pi_{\text{rand-cipher}'}$ UC-realizes $\mathcal{F}_{\text{rand-cipher}'}$ in the $(\mathcal{F}_{\text{bulletin}}, \mathcal{F}_{\text{key}})$ -hybrid (simplified) model with static corruption.

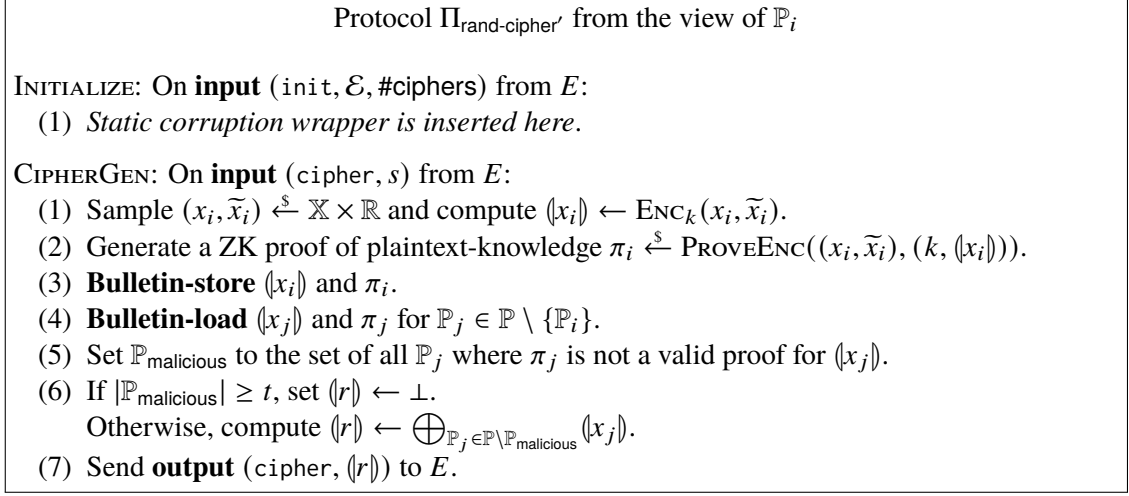
Proof (Sketch). Similarly to the simulator S_{MPC} used before, $S_{\text{rand-cipher}'}$ as in Figure 4.10 simulates a protocol instance and the real and ideal worlds differ only in a limited number of cases. These cases are 1. if a valid proof is rejected and 2. if an invalid proof is accepted. The correctness and soundness property of zero-knowledge proofs guarantee that these cases happen only with negligible probability. Both worlds are exactly the same if these cases do not happen.

INITIALIZE is perfectly simulated and AUDIT happens without interaction with the simulator in the ideal world or interaction with the adversary in the real world ($\mathbb{P}_{\text{audit}}$ cannot be corrupted). Note that $S_{\text{rand-cipher}'}$ can decrypt any cipher as \mathcal{F}_{key} is run inside the simulation where the simulator can intercept the private keys. The output for corrupted parties is not of interest as it is always the same in both world (as enforced by the static corruption wrapper). To be visibly different, the proofs would have to be validated incorrectly (i. e., if a proof π_i for a corrupted party \mathbb{P}_i wrongly yields $\text{VERIFY}(\pi_i) = \text{accept}$, \mathbb{P}_i would be detected by $\mathcal{F}_{\text{rand-cipher}'}$ and $\langle x_i \rangle$ would be excluded from the final sum in the ideal world but it is included in the sum in the real world).

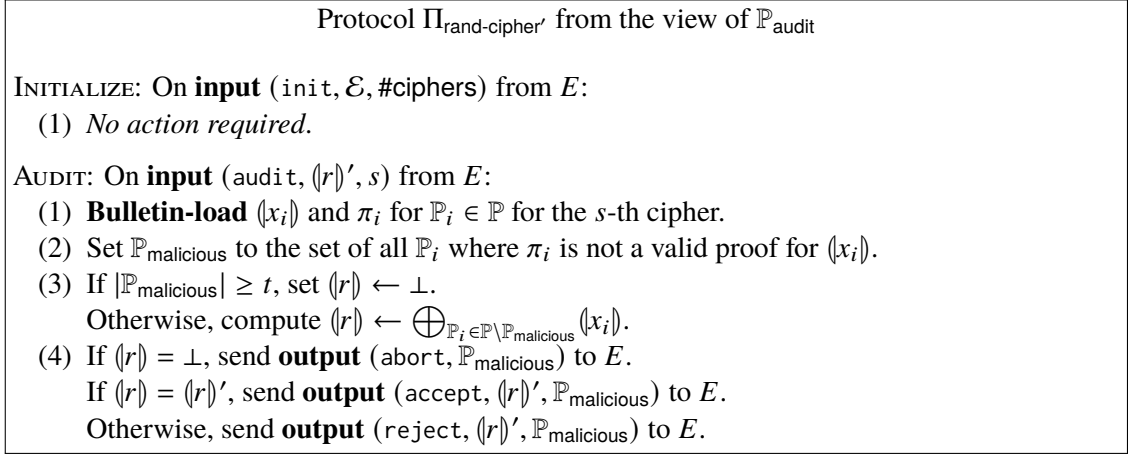
⁴The ciphers of honest parties are random, so the sum of all ciphers is random as well.

<p>Functionality $\mathcal{F}_{\text{rand-cipher}'}$ with adversary A</p> <p>INITIALIZE: On input $(\text{init}, \mathcal{E}, \#\text{ciphers})$ from all parties $\mathbb{P}_i \in \mathbb{P} = \{\mathbb{P}_1, \dots, \mathbb{P}_n\}$ and $\mathbb{P}_{\text{audit}}$:</p> <ol style="list-style-type: none"> (1) Send side-channel output (corrupt_parties) to A. Receive side-channel input $(\text{corrupt}, \mathbb{P}_{\text{corrupt}})$ from A. Save $\mathbb{P}_{\text{corrupt}} \subseteq \mathbb{P}$ and $\mathbb{P}_{\text{honest}} = \mathbb{P} \setminus \mathbb{P}_{\text{corrupt}}$. (2) For s from 1 to $\#\text{ciphers}$, set $\vec{\mathbb{P}}_{\text{malicious}}[s] \leftarrow \emptyset$. <p>CIPHERGEN: On input (cipher, s) from all parties $\mathbb{P}_j \in \mathbb{P}$:</p> <ol style="list-style-type: none"> (1) Do the following, depending on $\mathbb{P}_{\text{corrupt}}$: <ol style="list-style-type: none"> (1.a) If $\mathbb{P}_{\text{corrupt}} \geq t$, do the following: <ol style="list-style-type: none"> (1.a.1) Send side-channel output (manipulate) to A. Receive side-channel input $(\text{malicious}, \mathbb{P}_{\text{malicious}})$ from A. Save $\mathbb{P}_{\text{malicious}} \subseteq \mathbb{P}_{\text{corrupt}}$ as $\vec{\mathbb{P}}_{\text{malicious}}[s]$. (1.a.2) If $\mathbb{P}_{\text{malicious}} < t$, send side-channel output (cipher) to A and receive side-channel input $(\text{cipher}, \langle r \rangle)$ from A. Otherwise, set $\langle r \rangle \leftarrow \perp$. (1.b) Otherwise, do the following: <ol style="list-style-type: none"> (1.b.1) For each $\mathbb{P}_i \in \mathbb{P}_{\text{honest}}$, do the following: <ol style="list-style-type: none"> (1.b.1.1) Sample $(x_i, \tilde{x}_i) \xleftarrow{\\$} \mathbb{X} \times \mathbb{R}$ and compute $\langle x_i \rangle \leftarrow \text{ENC}_k(x_i, \tilde{x}_i)$. (1.b.1.2) Generate a ZK proof of plaintext-knowledge $\pi_i \xleftarrow{\\$} \text{PROVEENC}((x_i, \tilde{x}_i), (k, \langle x_i \rangle))$. (1.b.1.3) Send side-channel output $(\text{cipher}, \mathbb{P}_i, \langle x_i \rangle, \pi_i)$ to A. (1.b.2) For each $\mathbb{P}_i \in \mathbb{P}_{\text{corrupt}}$, do the following: <ol style="list-style-type: none"> (1.b.2.1) Send side-channel output $(\text{cipher}, \mathbb{P}_i)$ to A. Receive side-channel input $(\text{cipher}, \mathbb{P}_i, x_i, \langle x_i \rangle)$ from A. (1.b.2.2) If $x_i \neq \text{DEC}(\hat{k}, \langle x_i \rangle)$, add \mathbb{P}_i to $\vec{\mathbb{P}}_{\text{malicious}}[s]$. (1.b.3) Set $\langle r \rangle \leftarrow \bigoplus_{\mathbb{P}_i \in \mathbb{P} \setminus \vec{\mathbb{P}}_{\text{malicious}}[s]} \langle x_i \rangle$. (2) Save $\langle r \rangle$ as $\langle r_s \rangle$. (3) Send output $(\text{cipher}, \langle r_s \rangle)$ to all parties $\mathbb{P}_i \in \mathbb{P}_{\text{honest}}$. (4) For each $\mathbb{P}_i \in \mathbb{P}_{\text{corrupt}}$: <ol style="list-style-type: none"> (4.1) Send side-channel output $(\text{output_for}, \mathbb{P}_i)$ to A. Receive side-channel input $(\text{output}, \mathbb{P}_i, \langle r \rangle')$ from A. (4.2) Send output $(\text{cipher}, \langle r \rangle')$ to \mathbb{P}_i. <p>AUDIT: On input $(\text{audit}, \langle r \rangle', s)$ from $\mathbb{P}_{\text{audit}}$:</p> <ol style="list-style-type: none"> (1) Get the saved cipher $\langle r_s \rangle$. (2) If $\langle r_s \rangle = \perp$, send output $(\text{abort}, \vec{\mathbb{P}}_{\text{malicious}}[s])$ to $\mathbb{P}_{\text{audit}}$. If $\langle r_s \rangle = \langle r \rangle'$, send output $(\text{accept}, \langle r \rangle', \vec{\mathbb{P}}_{\text{malicious}}[s])$ to $\mathbb{P}_{\text{audit}}$. Otherwise, send output $(\text{reject}, \langle r \rangle', \vec{\mathbb{P}}_{\text{malicious}}[s])$ to $\mathbb{P}_{\text{audit}}$.
--

Figure 4.8: Alternative functionality to generate a random cipher.



(a) Protocol from the view of compute parties.



(b) Protocol from the view of auditing parties.

Figure 4.9: Protocol to generate a random cipher.

As mentioned above, this happens only with negligible probability. This leads to indistinguishability with overwhelming probability. \square

4.3.2 Protocol to Generate a Committed Cipher

Here, an alternative functionality $\mathcal{F}_{\text{com-cipher}'}$ (see Figure 4.11 for the functionality and Figure 4.12 for the protocol we want to use to realize it) is used instead of $\mathcal{F}_{\text{com-cipher}}$. The reason for this is the same as for $\mathcal{F}_{\text{rand-cipher}}$ discussed in Section 4.3.1: The simulator would have to prove something that is impossible to prove with information available to it.⁵ In this functionality, the values for

⁵The zero-knowledge proof (see Section 2.6.10) includes a proof of plaintext-knowledge which cannot be proven without the secret information (see Theorem 2.16).

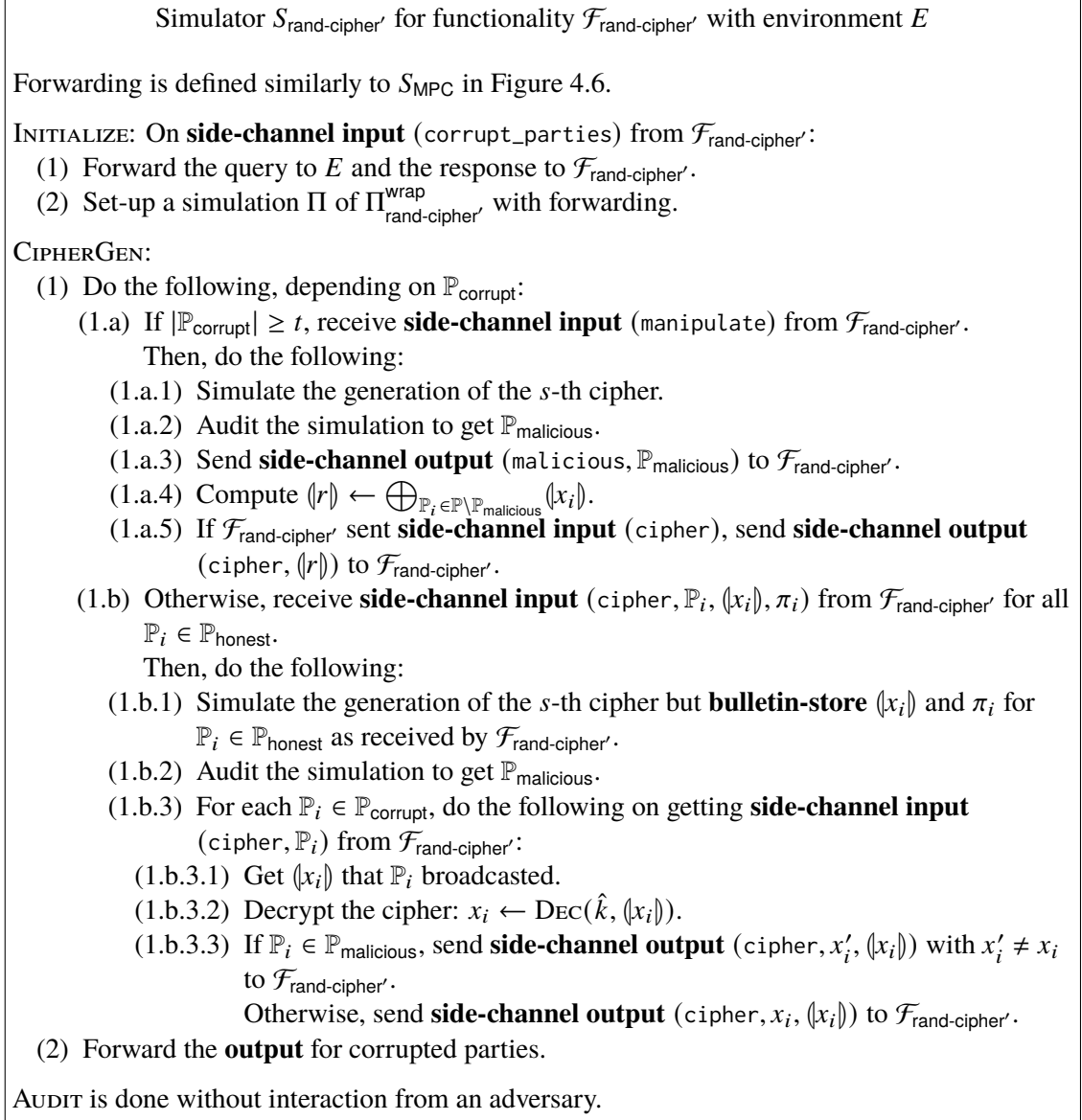


Figure 4.10: Simulator for the protocol to generate a random cipher.

honest parties are chosen by the functionality. A simulator could generate its own values with the same distribution, but the environment could detect that the **output** of $\mathbb{P}_i \in \mathbb{P}_{\text{honest}}$ is different from the **output** observed from the simulator. If the simulator uses the same ciphers and commitments as the functionality sends as **side-channel output** to it, the simulator cannot prove that ciphers and commitments are computed for the same plaintext (see Section 2.6.10) which has to be proven for all parties (without proving this, malicious parties could provide ciphers that do not correspond to the commitments).

Theorem 4.3 ($\Pi_{\text{com-cipher}'}$ **Realizes** $\mathcal{F}_{\text{com-cipher}'}$).

$\Pi_{\text{com-cipher}'}$ UC-realizes $\mathcal{F}_{\text{com-cipher}'}$ in the $(\mathcal{F}_{\text{bulletin}}, \mathcal{F}_{\text{key}})$ -hybrid (simplified) model with static corruption.

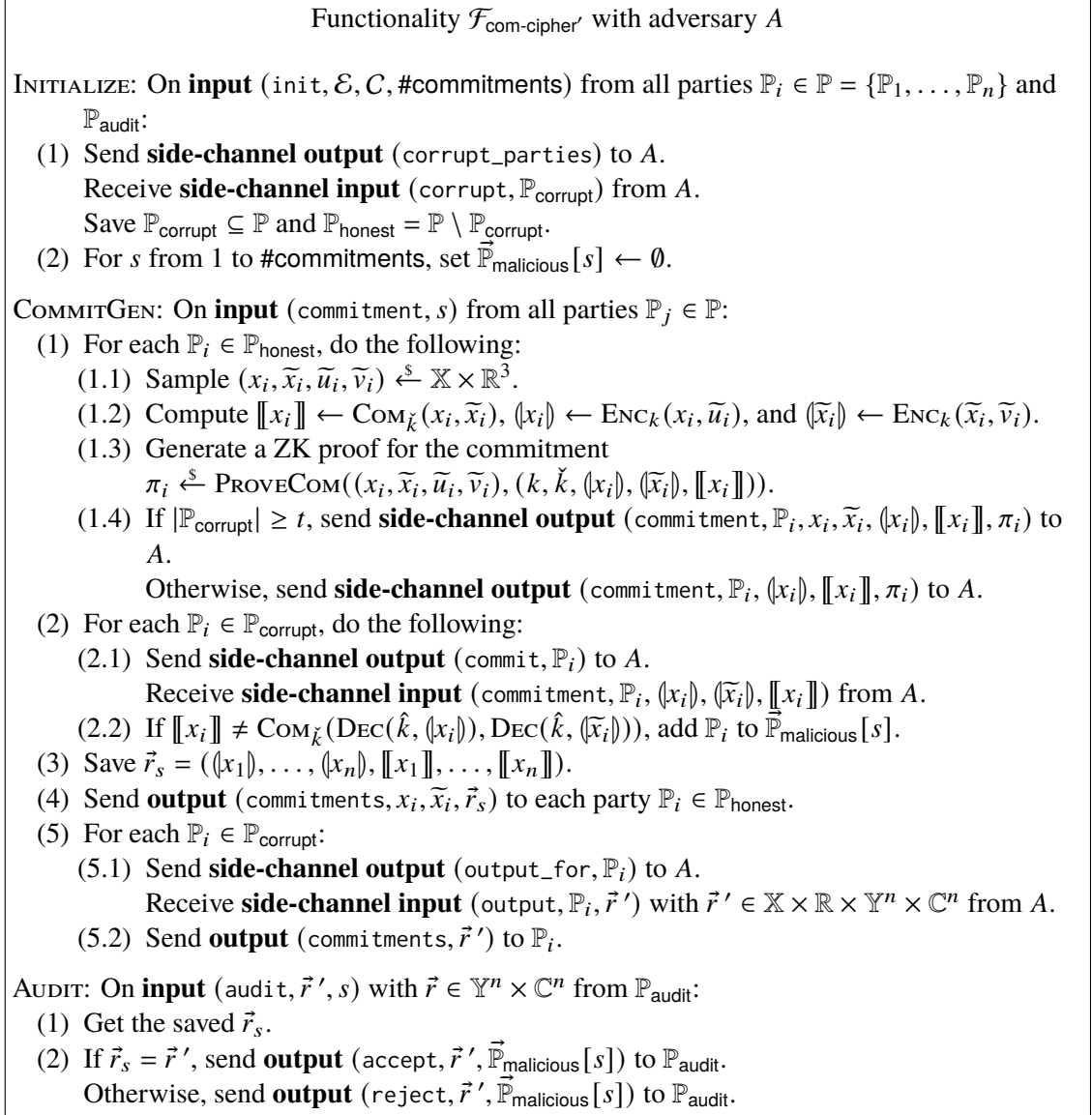


Figure 4.11: Alternative functionality to generate a committed cipher.

Proof (Sketch). Using the simulator $S_{\text{com-cipher}'}$ (see Figure 4.13), we can use the same argument as in the proof of Theorem 4.2.

Again, a simulation of INITIALIZE is not detectable. The output of corrupted parties is also not of interest. But the identified corrupted parties could be different in the real and ideal world as the method of detecting them is different (zero-knowledge proofs vs. decryption). Meaning, the worlds are indistinguishable if the proofs are correct and sound. \square

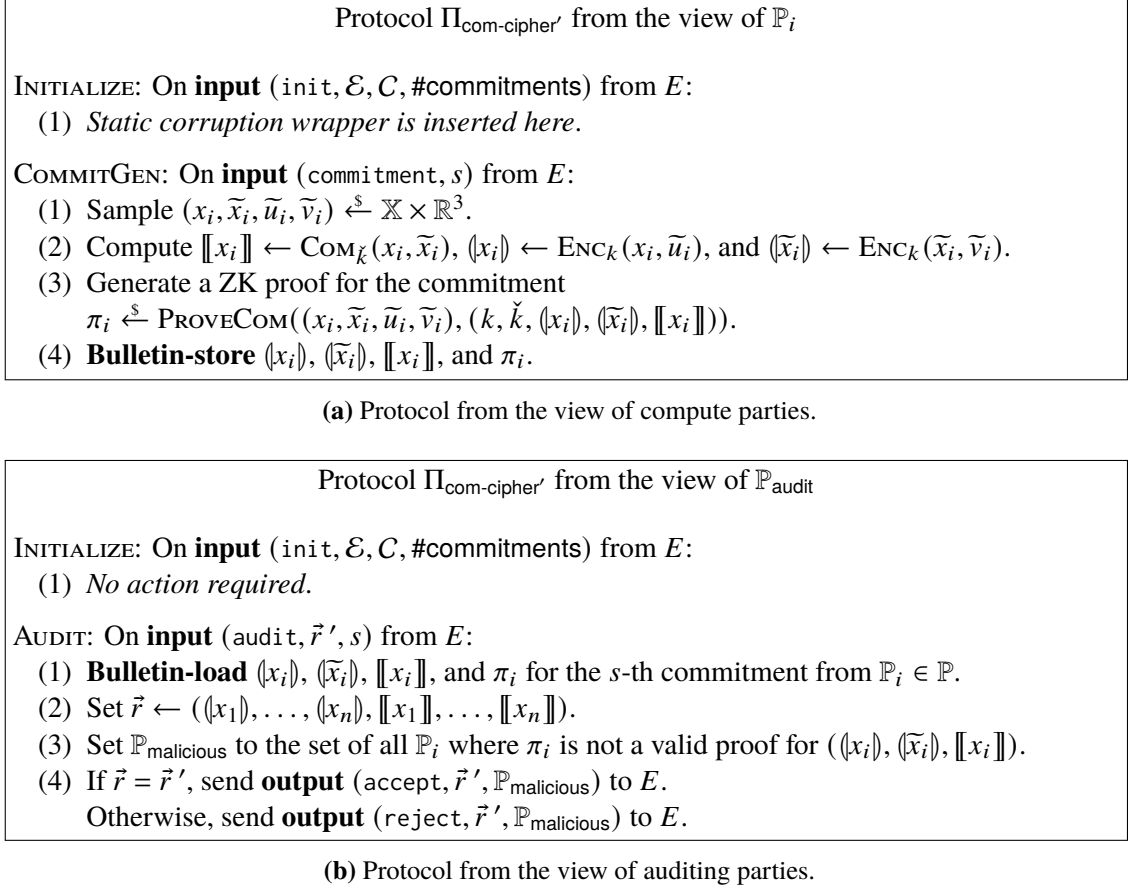


Figure 4.12: Protocol to generate a committed cipher.

4.3.3 Protocol for Threshold Decryption

The protocol Π_{key} to realize \mathcal{F}_{key} can be found in Figure 4.14. Here, we do not have to include details (i. e., zero-knowledge proofs) about the implementation in the functionality as the output of the functionality is only a single value (and not individual values per party as in $\mathcal{F}_{\text{com-cipher}'}$). The protocol should include the key generation, but details are left out here. For more on that, see Section 5.2.

Theorem 4.4 (Π_{key} Realizes \mathcal{F}_{key}).

Π_{key} UC-realizes \mathcal{F}_{key} in the $\mathcal{F}_{\text{bulletin}}$ -hybrid (simplified) model with static corruption.

Proof (Sketch). Again, we can use the same argument as for the proofs of Theorems 4.2 and 4.3, but we use the simulator S_{key} shown in Figure 4.15.

Here, the malicious parties are the same in both worlds as the simulator chooses them, but the outputs on both worlds might be different. If a wrong decryption share is not detected, a reconstruction using this share might yield a value different from the actual plaintext, but in the ideal world, only the decryption (or \perp) can be returned. Again, we argue that correct and sound proofs guarantee that this only happens with negligible probability. \square

Simulator $S_{\text{com-cipher}'}$ for functionality $\mathcal{F}_{\text{com-cipher}'}$ with environment E

Forwarding is defined similarly to S_{MPC} in Figure 4.6.

INITIALIZE: On **side-channel input** (`corrupt_parties`) from $\mathcal{F}_{\text{com-cipher}'}$:

- (1) Forward the query to E and the response to $\mathcal{F}_{\text{com-cipher}'}$.
- (2) Set-up a simulation Π of $\Pi_{\text{com-cipher}'}$ ^{wrap} with forwarding.

COMMITGEN:

- (1) For each $\mathbb{P}_i \in \mathbb{P}_{\text{honest}}$, receive **side-channel input** (`commitment`, \mathbb{P}_i , x_i , \tilde{x}_i , (x_i) , $\llbracket x_i \rrbracket$, π_i) or (`commitment`, \mathbb{P}_i , (x_i) , $\llbracket x_i \rrbracket$, π_i) from $\mathcal{F}_{\text{com-cipher}'}$.
- (2) Simulate the generation of the s -th commitment but **bulletin-store** (x_i) , $\llbracket x_i \rrbracket$, and π_i for $\mathbb{P}_i \in \mathbb{P}_{\text{honest}}$ as received by $\mathcal{F}_{\text{com-cipher}'}$.
- (3) For each $\mathbb{P}_i \in \mathbb{P}_{\text{corrupt}}$, do the following on getting **side-channel input** (`commitment`, \mathbb{P}_i) from $\mathcal{F}_{\text{com-cipher}'}$:
 - (3.1) Get (x_i) , (\tilde{x}_i) , $\llbracket x_i \rrbracket$, and π_i that \mathbb{P}_i broadcasted.
 - (3.2) Send **side-channel output** (`commitment`, \mathbb{P}_i , (x_i) , (\tilde{x}_i) , $\llbracket x_i \rrbracket$) to $\mathcal{F}_{\text{com-cipher}'}$.
- (4) Forward the **output** for corrupted parties.

AUDIT is done without interaction from an adversary.

Figure 4.13: Simulator for the protocol to generate a committed cipher.

A Note on the Security Proofs

In the UC-framework, it is essential to show that a protocol realizes a certain functionality. If it does, we can argue that the protocol is “as secure” as the functionality. To realize functionalities, we have to use a simulator. We wanted to support all secure commitment schemes (that are at least computationally hiding and binding). This includes perfectly binding ones. For these, we get into trouble when using a simulator, as we cannot decommit to different values after we see the result of the ideal functionality. This means we have to “work” on the real inputs, which should be – ideally – completely unknown to the adversary or simulator, i. e., not even encrypted inputs should be revealed (unlike in our functionality). This is the case for the functionalities of SPDZ protocols that use commitments [12, 44], but they only use a computationally binding commitment scheme (Pedersen commitments, see Section 2.4.1).

To overcome this apparent problem, our functionality “leaks” some of the implementation. This means for our functionality \mathcal{F}_{MPC} that the encryption of inputs is given to the adversary or simulator. The protocol Π_{MPC} uses encrypted inputs that are generated in the same way. So our simulator can just use what it got from the functionality and is able to just simulate a normal protocol run (e. g., without manipulating any commitments).

If we would support only computationally binding commitment schemes, we could adjust the functionality to not reveal anything about the inputs. For this, we would simply operate as the authors of the SPDZ protocols with commitments [12, 44]: We use dummy inputs for honest parties (i. e., the value 0) and adjust the decommitments of honest parties on openings. For this, we have to use an encryption scheme with the meaningless keys property [12, Definition 5] and could prove UC-security.

Protocol Π_{key} from the view of \mathbb{P}_i

INITIALIZE: On **input** $(\text{init}, \mathcal{E}, C, \#\text{decryptions})$ from E :

- (1) Use an MPC protocol to generate a public key, a private key, and public parameters for the commitment scheme. Generate shares of the keys. **Bulletin-store** the public keys and parameters to all parties. Send the private keys to the respective party only.
- (2) *Static corruption wrapper is inserted here.*

DECRYPT: On **input** $(\text{decrypt}, \langle x \rangle, s)$ from E :

- (1) **Bulletin-store** $\langle x \rangle$.
- (2) Partially decrypt: $d_i \leftarrow \text{PARTIALDEC}(\hat{k}_i, \langle x \rangle)$.
- (3) Generate a ZK proof correct partial decryption $\pi_i \stackrel{s}{\leftarrow} \text{PROVEDEC}(\hat{k}_i, (k_i, \langle x \rangle, d_i))$.
- (4) **Bulletin-store** d_i and π_i .
- (5) **Bulletin-load** d_j and π_j for $\mathbb{P}_j \in \mathbb{P} \setminus \{\mathbb{P}_i\}$.
- (6) Set $d_j \leftarrow \varepsilon$ if $\text{VERIFY}(\pi_j) = \text{reject}$.
- (7) Reconstruct $x \leftarrow \text{DECREC}(d_1, \dots, d_n)$.
- (8) Send **output** $(\text{plaintext}, x)$ to E .

(a) Protocol from the view of compute parties.

Protocol Π_{key} from the view of $\mathbb{P}_{\text{audit}}$

INITIALIZE: On **input** $(\text{init}, \mathcal{E}, C, \#\text{decryptions})$ from E :

- (1) **Bulletin-load** the public keys and parameters that the compute parties generated.

AUDIT: On **input** (audit, x', s) from E :

- (1) **Bulletin-load** the s -th $\langle x \rangle$.
- (2) **Bulletin-load** d_i and π_i for the s -th cipher from all $\mathbb{P}_i \in \mathbb{P}$.
- (3) Set $\mathbb{P}_{\text{malicious}} \leftarrow \{\mathbb{P}_i \in \mathbb{P} \mid \text{VERIFY}(\pi_i) = \text{reject}\}$ and $d_j \leftarrow \varepsilon$ if $\mathbb{P}_j \in \mathbb{P}_{\text{malicious}}$.
- (4) Reconstruct $x \leftarrow \text{DECREC}(d_1, \dots, d_n)$.
- (5) If $x = \perp$, send **output** $(\text{abort}, \mathbb{P}_{\text{malicious}})$ to E .
If $x = x'$, send **output** $(\text{accept}, x', \mathbb{P}_{\text{malicious}})$ to E .
Otherwise, send **output** $(\text{reject}, x', \mathbb{P}_{\text{malicious}})$ to E .

(b) Protocol from the view of auditing parties.

Figure 4.14: Protocol for threshold decryption.

The other works on SPDZ protocols with commitments use full-threshold secret sharing, but we use Shamir secret sharing. This should not pose a problem, but the functionality has to be adjusted (similarly to our functionality) and the proofs as well.

As we used hybrid models, we could use decryption in the simulators for all prerequisite functionality. This is actually quite similar to SPDZ and its variants. There, all required preprocessing is combined into a single offline functionality. The equivalent for our protocol would combine $\mathcal{F}_{\text{rand-cipher'}}$,

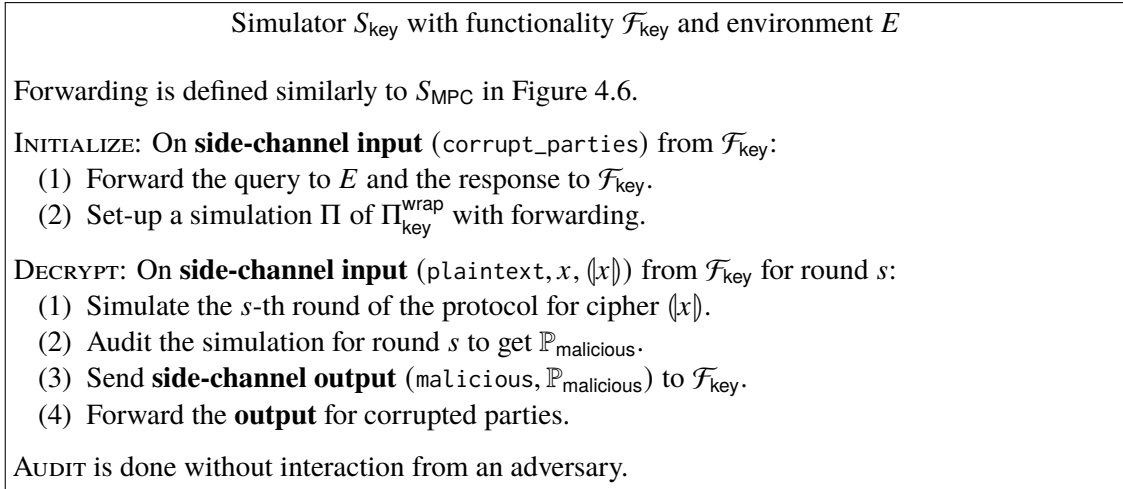


Figure 4.15: Simulator for the protocol for threshold decryption.

$\mathcal{F}_{\text{com-cipher}}$, and \mathcal{F}_{key} into one functionality. Obviously, the corresponding simulator could decrypt values in all phases that correspond to the different functionalities as it gets the private keys from the internal simulation of \mathcal{F}_{key} which contains key generation.⁶

⁶After this, the simulator would program the random oracle so the offline functionality uses the same keys.

5 Possible Implementations

In this chapter, we address several topics regarding possible implementations. First, we discuss that it is actually possible to implement an MPC protocol as described in Chapter 4. Then, we outline several changes to the proposed protocol to make it more efficient.

Obviously, Π_{MPC} cannot be used to implement \mathcal{F}_{MPC} as it includes further functionalities. Universal composability (Corollary 2.9) allows us to use the protocols for these functionalities instead. I.e., by applying the composition theorem multiple times, we get that $\Pi_{\text{MPC}}^{\mathcal{F}_{\text{rand-cipher}} \rightarrow \Pi_{\text{rand-cipher}}, \mathcal{F}_{\text{com-cipher}} \rightarrow \Pi_{\text{com-cipher}}, \mathcal{F}_{\text{key}} \rightarrow \Pi_{\text{key}}, \mathcal{F}_{\text{bulletin}} \rightarrow \Pi_{\text{bulletin}}}$ UC-realizes \mathcal{F}_{MPC} . The implementation of Π_{bulletin} and an MPC protocol to generate public and private keys (part of Π_{key}) are discussed in Sections 5.1 and 5.2. After that, more details about the SHE scheme are given in Section 5.3. Sections 5.4 and 5.5 discuss the option of using different encryption or commitment schemes at the same time in the protocol. How we would handle preprocessing is outlined in Section 5.6 and runtime improvements if parties are unlikely to cheat can be achieved with changes from Section 5.7. In Section 5.8, changes and improvements to the protocol, if full-threshold secret sharing is used, are detailed. Finally, Section 5.9 gives some ideas how to best handle arithmetic circuits.

5.1 Implementing a Bulletin Functionality

The protocols presented in Chapter 4 all use a bulletin board functionality $\mathcal{F}_{\text{bulletin}}$. While it is not strictly necessary to use a bulletin board – broadcast channels could be used instead (see Table 5.1) – bulletin boards enable more practical protocols. If the protocol should allow any party to audit a protocol run, the parties that might want to audit should be present at the time of the protocol run to receive broadcast messages that are required to audit. With a bulletin board, they would only need to request the necessary data from it. Nevertheless, we also consider broadcasts here, as both approaches are related.

Bulletin boards and broadcasting have been studied [25, 26, 31, 83, 97] – also in the context of the UC-framework [54, 66, 74] – as many protocols (like ours) assume that broadcasting channels or bulletin boards are available. A broadcast can be implemented for any number of corrupted parties if the broadcast is also allowed to abort the protocol [66]. This is not enough for our protocol with full-threshold secret sharing (see Section 5.8) as – even though there are other places where the protocol might abort – we also need accountability.

There are two classical ways to solve the problem of implementing broadcast channels: The first one requires at least two thirds of the parties to be honest, the second one requires authenticated communication (but can tolerate any number of corrupted parties) [97]. The first one is composable but the second one is not [83]. Randomized protocols can achieve composability for any number of corrupted parties (even more efficient for honest majorities) [83]. A protocol to implement a broadcast functionality was developed as well [66]. This protocol is not randomized but also

Statement	Bulletin Board	Broadcast
Bulletin-store x	Send $(\text{store}, \text{id}(x), x)$ to the bulletin board functionality.	Broadcast $(\text{store}, \text{id}(x), x)$ to all parties (using the broadcast functionality). All parties store the value x and associate it with $\text{id}(x)$.
Bulletin-load x	Send $(\text{load}, \text{id}(x))$ to the bulletin board functionality and receive (value, x) .	Load the value associated with $\text{id}(x)$.

Table 5.1: Equivalent operations when using a bulletin board or broadcast channels.

does not guarantee output delivery (thus it does not have the limitation of requiring a two-thirds honest majority), e. g., the adversary can block the broadcast (possibly indefinitely). We implicitly modelled the same in our functionality \mathcal{F}_{MPC} , too. For example – in Step (1) of the COMPUTE phase in Figure 4.4 – when the adversary is supposed to give the malicious parties to the functionality. Here, the functionality would technically block (possibly indefinitely) until the adversary supplies it with the message $(\text{malicious}, \mathbb{P}_{\text{malicious}})$. To handle these cases, a protocol (and functionality) would have to be designed differently.¹

For e-voting systems, the bulletin board is assumed to be uncorrupted (i. e., in Helios [79] or Ordinos [79]). Proposals for (distributed) implementations have been developed as well [43, 73]. If the MPC protocol is used in the context of e-voting, the bulletin board protocol used for other parts of the system can be reused for our protocol, e. g., our protocol can be nicely combined with (existing) e-voting systems. For more on this, see Chapter 8.

5.2 Implementing the Key Generation

In the protocol Π_{key} , we have to generate a public key and private key shares (for each party) for the encryption scheme. Additionally, public parameters for the commitment scheme have to be generated. This is possible for classical discrete-logarithm-based schemes [56, 57] but also for RLWE-based schemes [8, 24, 48, 100]. Secure distributed key generation protocols have been developed for other encryption schemes as well [95]. Protocols that are composable in the sense of the UC-framework have been developed, too [1, 116].

Another issue arises when we look at the key generation for the commitment scheme. As we use the commitments to make our protocol auditable, it should not be possible for an adversary to decommit to wrong values. The corrupted parties are part of the key generation, so it is important that – even in the case of full corruption – the adversary never gets trapdoor information to break the binding property of the commitment scheme. One option is to use a perfectly binding commitment scheme to circumvent this problem. If all (or at least t) parties are corrupted, they still cannot decommit to

¹A blocking of an adversary could happen in reality: A corrupted party could simply refuse to send their messages. On the other hand, a network attacker could block messages of honest parties. It might not be possible find the cause for the missing messages, so our protocol waits in case it might finally get the delayed messages.

wrong values (but they might be able to break the hiding property). Another possibility is to use some external (random) information to generate the public keys for the commitment scheme (i. e., if we assume to operate in the common reference string model). We also need some assumptions that guarantee that it should be hard (i. e., assuming discrete logarithms are hard to compute) to get the trapdoor information.

In the case of Pedersen commitments (see Section 2.4.1), this external information could be already the public key or a random group element that is added² to the key. The parties could still try to get the trapdoor information by computing the discrete logarithm of the public key (or the random group element if they already know it for the unmodified public key), but this is still as hard as in the case where only a single party is corrupted.

5.3 Somewhat Homomorphic Encryption Scheme

A somewhat homomorphic encryption scheme is needed to generate the multiplication triples. We can use the same scheme as SPDZ [49] or its auditable version [12]. This scheme is already introduced in Section 2.3.5, but some adjustments to the protocol are needed that were left out for simplicity. This mainly changes the zero-knowledge proofs to ensure that certain ciphertexts are “correct”, e. g., the noise is small enough to perform the necessary operations in the process of the protocol. Compared to the auditable SPDZ protocol, no new proofs are needed – we can just use their proof of plaintext-knowledge to implement `PROVEENC` and adjust the proof for their commitment relation³ to implement `PROVCOM`. This version of SPDZ (like the original one) does not include proofs for correct decryption. Instead, they allow errors that are then detected by the MACs (errors for inputs) or the so-called “SPDZ sacrifice” (errors in multiplication triples). Neither of these techniques gives us accountability, which is why we use verifiable decryption that is implemented in an accountable way (i. e., proofs of each party are checked, instead of checking aggregated proofs of all parties). To realize `PROVEDEC` for this encryption scheme, the general approach (described in Section 2.3.2) is used to build a verifiable threshold cryptosystem from a verifiable one (using the appropriate zero-knowledge proofs [101]). Finally, the parameters of the encryption scheme need some adjustment as we need to compute

$$\left(\bigoplus_{i=1}^n \langle a_i \rangle \right) \odot \left(\bigoplus_{i=1}^n \langle b_i \rangle \right) \oplus \left(\bigoplus_{l=1}^{t-1} j^l \otimes \left(\bigoplus_{i=1}^n \langle (m_l)_i \rangle \right) \right) \oplus \langle r_j \rangle$$

instead of

$$\left(\bigoplus_{i=1}^n \langle a_i \rangle \right) \odot \left(\bigoplus_{i=1}^n \langle b_i \rangle \right) \oplus \left(\bigoplus_{i=1}^n \langle r_i \rangle \right)$$

as the longest chain of ciphertext operations.

²or multiplied – depending on notation

³they only support Pedersen commitments, so it needs adjustments for other commitment schemes

Note that, instead of using SHE to multiply ciphertexts, one could use some MPC protocol to do the same. For this, the encryption scheme needs to be additive homomorphic only, e. g., the classical ElGamal or Paillier encryption schemes could be used (see Sections 2.3.3 and 2.3.4). This is done, for example, in a variant of SPDZ [72]. One might also use the protocol to multiply two ciphers from Cramer et al. [40] which is accountable if the threshold decryption is accountable.

5.4 Multiple Encryption Schemes

The MPC protocol – as presented until now – uses the SHE only in the preprocessing to generate multiplication triples.⁴ After the triples are generated, only views of the triples with shares, randomness, and commitments are used in the protocol. This means, to generate the views for the inputs, a different (verifiable) threshold encryption scheme could be used. This encryption scheme for the inputs would only need to support additive homomorphic operations, so one might find a more efficient scheme for this task.

5.5 Multiple Commitment Schemes

As proven in Section 4.2, any additive homomorphic commitment scheme could be used to implement the protocol – as long as it is (at least) computationally binding and (at least) computationally hiding. One might use encryption schemes (i. e., the ones discussed in Sections 2.3.3 to 2.3.5) or any other commitment scheme that fits these requirements.

Notably, the commitments of different parties are never combined (i. e., added up) at any time. This leaves the possibility of using a different commitment scheme (or different commitment parameters) for each party.

5.6 Leveraging Preprocessing

Like SPDZ and (all) related protocols, we can move many parts of the protocol to an input-independent preprocessing phase.⁵ To further take advantage of preprocessing, we can use the following technique to reduce the online (e. g., not preprocessing) complexity of the input phase: If the compute parties generate $\#inputs$ additional random ciphers $\langle r_j \rangle$ (with $\mathcal{F}_{rand-cipher}$), we can reduce the input phase to three steps. For this, the compute parties treat the $\langle r_j \rangle$ as inputs (in the preprocessing step) and views $[r_j]_i$ for each party \mathbb{P}_i are computed. We then have the following in the input phase (for a party \mathbb{P}_i and – as in Figure 4.5b – we assume that the compute parties get a cipher $\langle x_j \rangle$ for an input x_j)

- (1) Set $\langle m_j \rangle \leftarrow \langle x_j \rangle \ominus \langle r_j \rangle$.
- (2) Threshold decrypt $\langle m_j \rangle$ to get m_j .

⁴All SPDZ-like protocols that use multiplicative MACs also need ciphertext multiplications to generate MAC shares: $\langle \alpha \cdot x \rangle$ is generated from $\langle \alpha \rangle$ and $\langle x \rangle$. Then, $\langle \alpha \cdot x \rangle_i$ can be generated in a similar way to generating $\langle x \rangle_i$ from $\langle x \rangle$.

⁵We still require to know (an upper bound on) the number of inputs and the number of multiplications.

(3) Set $[x_j]_i \leftarrow [r]_j +_c m_j$.

Here, we used the general technique of preprocessing “something” random and adjusting it to fit what we want. This is used in (auditable) SPDZ and in our protocol to enable multiplication (with the multiplication triples) and to produce commitments for shares (commitments for random values are adjusted to fit the shares (see Figure 4.5b)).

With this, most of the work for compute parties can be moved to a preprocessing phase. Looking at our protocol (see mainly Figure 4.5b), the whole INITIALIZE phase can be done offline. This would include generating the mentioned random ciphers $([r_j])$ and their views.

The other phases of the protocol are the INPUT phase (can be done in three steps with one decryption per input as described above) and the COMPUTE phase (see Figure 4.5c). The latter includes the communication of one share and randomness value per opening (per party) and twice as much for each multiplication. This means, the communication cost is the same as in the auditable version of the SPDZ protocol (excluding the cost of the MAC check in that protocol, which we do not need). In the following, we describe how we can reduce the computational overhead in the COMPUTE phase (caused by the additional commitments) slightly in cases where parties do not cheat.

5.7 Speed-Up in Case of Risk-Avoiding Adversaries

If we consider risk-voiding adversaries [79, 80], we might consider optimizing our protocol to be more efficient in cases where parties do not cheat. Intuitively, such adversaries only perform manipulations that are “small” enough that they stay undetected with a high probability. Other manipulations are only performed with negligible probability. This leads to the following idea: We only check the individual shares if they do not reconstruct to a value that is expected, e. g., we also have commitments for the shared values and not only for the shares. Risk-avoiding adversaries might then manipulate their shares, but only in a way that the reconstructed value stays correct (here, we would not check their shares and they would stay undetected). Then, the result of the computation is still correct as well. It seems reasonable to not want to find all cheaters if they still compute the correct result.

Obviously, each compute party has to compute the arithmetic circuit on $n + 2$ values (a view $[x]_i$ consists of n commitments, a share, and the randomness used to commit to the share). We could just combine the share-commitments $\llbracket \langle x \rangle_i \rrbracket$ to

$$\llbracket x' \rrbracket = \bigoplus_{i=1}^n \llbracket \langle x \rangle_i \rrbracket$$

and check if

$$(5.1) \quad \llbracket x' \rrbracket = \text{COM}_{\tilde{\kappa}} \left(\sum_{i=1}^n \langle x \rangle_i, \sum_{i=1}^n \tilde{x}_i \right)$$

whenever the commitments of shares would be checked in the protocol described in Section 4.2. Note that $\llbracket x' \rrbracket$ is not a commitment to x as Shamir secret sharing is used. For the case of full-threshold secret sharing, a (even more) efficient technique is described in Section 5.8. Intuitively, it should be similarly hard to provide wrong shares or randomness to pass this test as it is for the

original commitments $\llbracket \langle x \rangle_i \rrbracket$. Additionally, successfully cheating here might not be very beneficial for an adversary, as the shares that pass this test will be used for the reconstruction with Shamir secret sharing, where each party is free to pick any t shares to reconstruct. This means, it seems to be an unsuitable way to manipulate the outcome of the multi-party computation.

If the check using $\llbracket x' \rrbracket$ fails, the individual checks with the share-commitments $\llbracket \langle x \rangle_i \rrbracket$ can be employed. This way, the wrong shares (and malicious parties) are identified and reconstruction can be done with the remaining shares (if possible). In practice, this means that the parties have to compute the commitment $\llbracket x' \rrbracket$ for each input value x (from the corresponding share-commitments $\llbracket \langle x \rangle_i \rrbracket$). Then, the circuit is evaluated on views that consist of $\langle x \rangle_i$, \tilde{x}_i , and $\llbracket x' \rrbracket$ for party \mathbb{P}_i – with one commitment instead of n commitments. If a test for an opening fails, the party has to “go back” and start computing the circuit on the n share-commitments. Obviously, this is more effort than computing the circuit in the original way (as in Figure 4.5). This is why this technique is especially suited for cases where risk-avoiding adversaries are assumed. The adversaries have to fear to be individually identified (as the protocol is accountable) and are therefore less likely to cheat, which means the expensive path of recomputing the circuit on share-commitments is often not taken.

The auditing party can use the same technique and check the circuit using the combined commitments instead of the share-commitments. Alternatively, the auditing party could always check the share-commitments to find cheaters after the fact (where they cheated successfully in the check of the compute parties with the combined commitments but the checks with the share-commitments detect them). The probability that cheaters pass the check with the combined commitments as well as the checks with the share-commitments is obviously smaller than only the passing the check of share-commitments, e. g., this technique should be even a bit more “secure”.

Now we shortly explain why commitments like $\llbracket x' \rrbracket$ are checked instead of commitments for the shared value x (e. g., instead of $\llbracket x \rrbracket$): The first problem is getting a commitment $\llbracket x \rrbracket$ for the input x . If one would want to reconstruct $\llbracket x \rrbracket$ from the share-commitments $\llbracket \langle x \rangle_i \rrbracket$, multiplication of commitments is required (if the reconstruction of the Shamir secret sharing scheme is applied to (shares of) commitments). Alternatively, the compute parties could generate a commitment of the input as follows (e. g., add the following to SHARE in Figure 4.5b):

$$\begin{aligned}
 \llbracket x \rrbracket &= \text{COM}_{\tilde{k}} \left(\text{DEC}(\hat{k}, \langle x \rangle \oplus \bigoplus_{i=1}^n \langle r_i \rangle), \tilde{0} \right) \boxplus \bigboxplus_{i=1}^n \llbracket r_i \rrbracket \\
 &= \text{COM}_{\tilde{k}} \left(x + \sum_{i=1}^n r_i, \tilde{0} \right) \boxplus \bigboxplus_{i=1}^n \text{COM}_{\tilde{k}}(r_i, \tilde{r}_i) \\
 &= \text{COM}_{\tilde{k}} \left(x, - \sum_{i=1}^n \tilde{r}_i \right) \\
 &= \text{COM}_{\tilde{k}} \left(x, \sum_{i=1}^n \tilde{x}_i \right)
 \end{aligned}$$

where decryption is implemented with threshold decryption (e. g., with \mathcal{F}_{key} as in Π_{MPC}). The checks of combined commitments (like in Equation (5.1)) are then replaced with checks of

$$\llbracket x \rrbracket = \text{COM}_{\tilde{k}} \left(\text{REC}(\langle x \rangle_1, \dots, \langle x \rangle_n), \sum_{i=1}^n \tilde{x}_i \right).$$

Functionality $\mathcal{F}_{\text{rand-cipher-share}}$ with adversary A

This Functionality behaves like $\mathcal{F}_{\text{rand-cipher}'}$ in Figure 4.8 but also outputs x_i to party \mathbb{P}_i .

Figure 5.1: Functionality to generate a random shared cipher.

As some shares are simply ignored in the reconstruction, a cheater might stay undetected, but their wrong share also did not influence the result. This is why it is important to use the reconstruction result later on (instead of reconstructing x again with different shares). With these kinds of checks, all parties might find different sets of malicious parties (as it depends on the shares used for reconstruction). One might fix that by deterministically determining the shares that should be used or simply using the method of combined commitments $\llbracket x' \rrbracket$ above.

On a final note: We would also have to adjust the functionality to compensate for the changed behavior. The above description forms the changes to the protocol. In the functionality, an auditing party would now accept an output \vec{y} , even if $|\mathbb{P}_{\text{non-malicious}}| < t$, but only if $\vec{y} = \vec{y}_{\mathcal{C}}$.

5.8 More Efficient Online Phase with Full-Threshold Secret Sharing

With full-threshold secret sharing there is no way to “recover” if at least one share of a shared value is wrong. With Shamir secret sharing and commitments, we are able to identify wrong shares and (if not too many shares are wrong) can still reconstruct the correct value from the other shares. We are still able to identify wrong shares with full-threshold secret sharing (using the commitments), but there is no way to reconstruct the value without the missing share.

This means we can use the efficient MAC checks of the auditable version of SPDZ [12] to perform a single check at the end of the protocol (or at least check a large batch of shares at once) to see if all shares and the revealed randomness for the commitments are correct. Reintroducing the MACs of SPDZ stops us from using the options mentioned in Sections 5.4 and 5.5. Multiple encryption schemes are no longer possible as input ciphers have to be multiplied with the cipher of the MAC key. If we want to guarantee that the protocol does not abort if a value x can be reconstructed from opened shares $\langle x \rangle_i$, we should also check if

$$\bigoplus_{i=1}^n \llbracket \langle x \rangle_i \rrbracket = \text{COM}_{\tilde{k}} \left(\sum_{i=1}^n \langle x \rangle_i, \sum_{i=1}^n \tilde{x}_i \right),$$

even if there is a $\langle x \rangle_i$ with

$$\llbracket \langle x \rangle_i \rrbracket \neq \text{COM}_{\tilde{k}}(\langle x \rangle_i, \tilde{x}_i).$$

This means, we cannot use different commitment schemes for each party.

We will discuss this more in Chapter 7, but this is similar to the accountable of SPDZ protocol of Cunningham et al. [44]. An obvious difference is that we also have MACs of randomness in our protocol. This way, if the MAC check is successful, a more complete check of all shares using commitments is likely to be successful as well. If not, the compute parties would abort. They do not know who is responsible from the MAC check, but we do not require compute parties to know that (if they would want to know they could run the AUDIT sub-protocol of $\mathbb{P}_{\text{audit}}$). This has then the same complexity as the mentioned version of SPDZ [44].

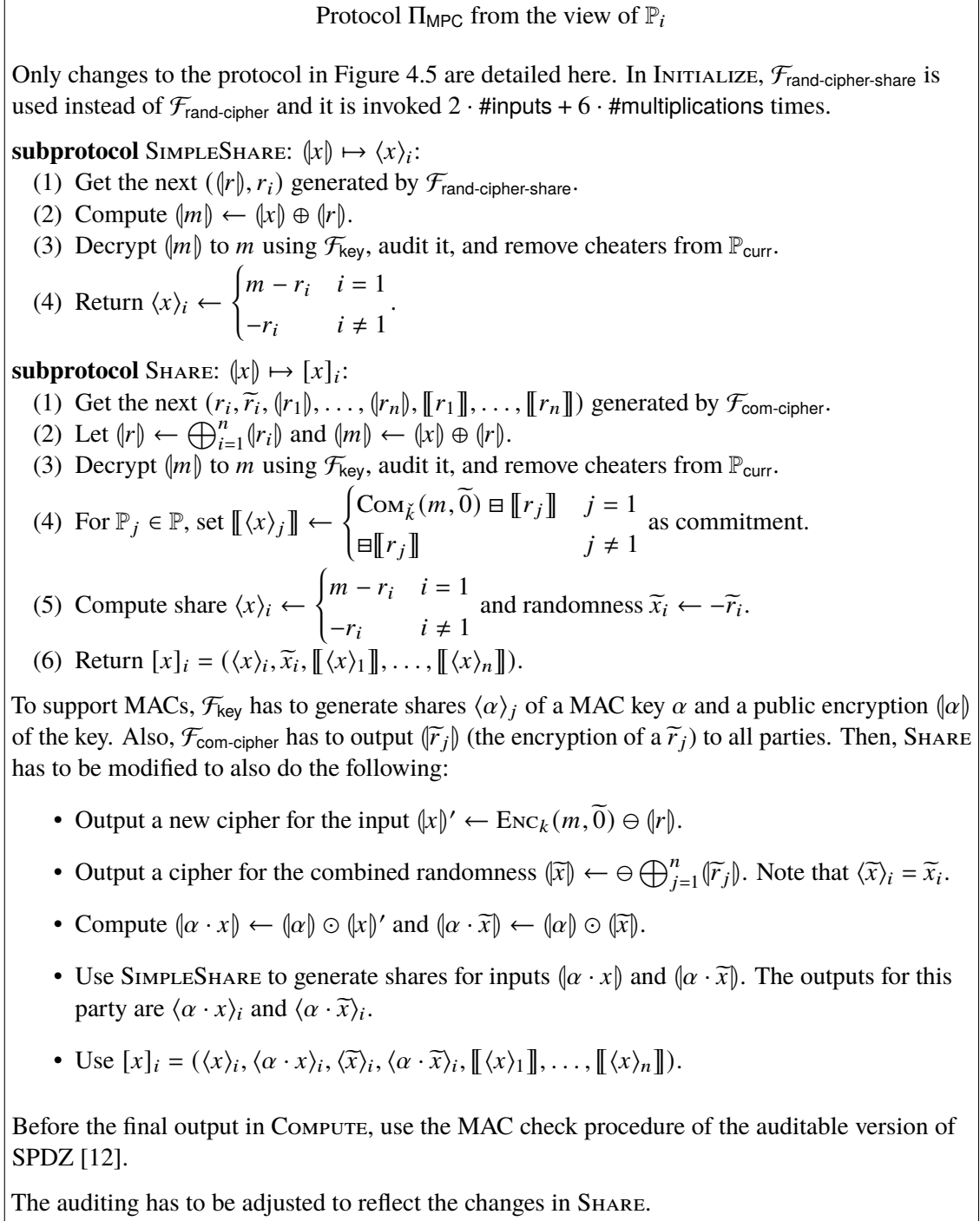


Figure 5.2: Changes to the protocol Π_{MPC} if full-threshold secret sharing is used.

5.9 On Arithmetic Circuits

We glossed over several aspects about arithmetic circuits – we will remedy that in this section. Firstly, we assumed that we just get an arithmetic circuit for a multi-party computation. In practice, this circuit has to be engineered – potentially for every use of the protocol (i. e., if some parameters change). Fortunately, there has been developments in how to describe the functionality of a circuit in a structured way (e. g., like in a programming language) and generate the circuit from the description [23, 91, 94, 105]. This can be done before starting the MPC protocol and everyone can make sure that circuit is what should be computed (i. e., by “recompiling” the circuit).

After everyone agrees on a circuit, we need to find an order in which to evaluate it. If two compute parties evaluate it in a different order – for example two OPEN gates could be evaluated in an interchangeable order – a deadlock could arise. As the circuit is a directed acyclic graph, the evaluation order could just be the topological ordering (needed for dependencies of gates in the circuit) where the algorithm to get the ordering is deterministic and defined as part of the (preprocessing of the) protocol.

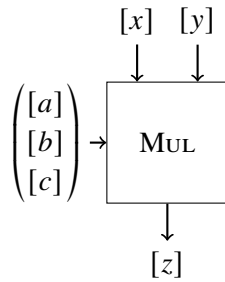
The way we described the circuits before, there is only a way to publicly output a value, e. g., everyone (compute parties and external observers) learns the output. To allow for private input, one could simply add a few more gates. If party \mathbb{P}_j should get the value of $[x]$ as private output, r is chosen randomly by \mathbb{P}_j and added as input to the circuit. This means, \mathbb{P}_j is now an additional input party that encrypts r and $[r]$ is generated for it. Then, OPEN(ADD($[x]$, $[r]$)) (e. g., an additional addition and opening gate) is added to the circuit. Lastly, \mathbb{P}_j can privately compute the value x by subtracting r from the publicly opened value. This way, everyone can still check if the right value was opened (by checking the decommitments at the opening), but only \mathbb{P}_j knows the secret value of x . Taking this to extreme means that we can publicly audit a multi-party computation (and get verifiability and accountability) without publishing any information about the outputs.⁶

Additionally, the circuit can be “compressed” if we make the multiplication with beaver triples explicit (see Figure 5.3). The resulting graph has only the following node types: 1. (secret) input gates, 2. public constants, 3. additive arithmetic gates (e. g., ADD, ADD_c, SUB, SUB_c, and MUL_c), and 4. public output gates (OPEN). Obviously, there are only OPEN gates as “sinks” in the graph (for other types of nodes, nobody would know the value if it is not reconstructed). If we combine the arithmetic gates between inputs and OPEN gates, we get sub-graphs that compute

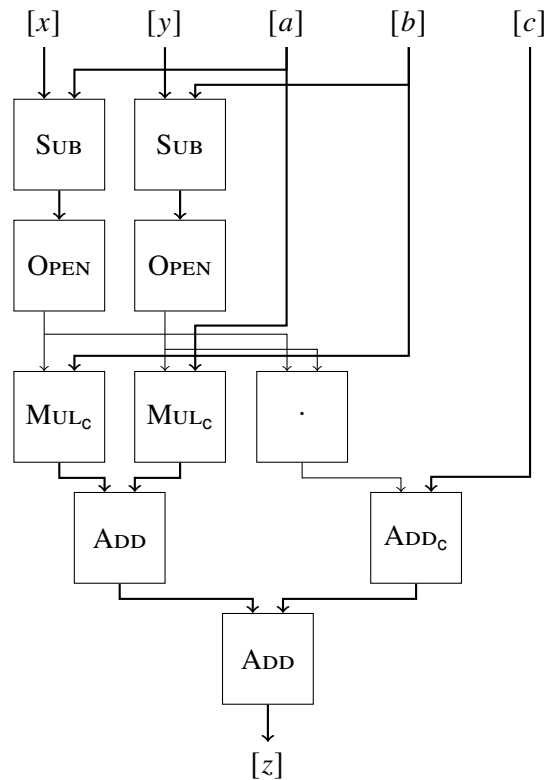
$$(5.2) \quad \sum_{j=1}^m a_j \cdot [x_j] +_c c,$$

as we can write any term with only these additive arithmetic operations in this form. As we get intermediate OPEN gates by substituting multiplication gates, it is clear that the depth of the circuit with these combined sub-terms depends on the number of multiplications in the original circuit (but not on the number of other arithmetic gates).

⁶If r is distributed uniformly at random, the masked output is distributed randomly as well, e. g., without knowing the r that was used, nobody can successfully recover the output.



(a) Multiplication gate using Beaver triples.



(b) Substitution circuit for a multiplication using Beaver triples.

Figure 5.3: Substitution of multiplication gates.

On a related topic, as we only have terms like Equation (5.2) followed by a public opening: If we do the addition of c after the opening, we never need to add (or subtract) constants from views. As described in Section 4.2, how to add constants to a view is dependent on the secret sharing scheme but with this simple swap, we do not have to worry about this (or addition-with-constants does not need to be defined).

6 Security Analysis

Now that we presented (in Chapter 4) our protocol (Figure 4.5) and proved (in Section 4.2) that it realizes the functionality \mathcal{F}_{MPC} (Figure 4.4), we can continue with our security analysis. As we analyze the protocol in the UC-framework, we can simply “read-off” the security properties from the ideal functionality (as we proved that our protocol is “as good” as the functionality).

6.1 Privacy

Let us start with privacy. Obviously, an adversary learns the following things: 1. an encryption of an input (if the input party is not corrupted and less than t compute parties are corrupted) 2. a plain input (the input party is corrupted or at least t compute parties are corrupted), and 3. the result of the computation. The result of the computation and inputs of corrupted input parties are of no special interest, as the adversary is allowed to know these. The encrypted inputs do not reveal anything as the encryption scheme is CPA-secure and no other information about the inputs is revealed (except the computation result) by the functionality. If at least t compute parties are corrupted, all inputs are known to the adversary. In total, we get that the protocol achieves privacy if less than t compute parties are corrupted.

6.2 Correctness

For correctness, we see that honest parties either get the correct result of the computation or \perp . This means our protocol is correct but might abort. This is the case if less than $n - t$ parties do not behave maliciously. In other words, if t parties (or more) follow the protocol, the output is correct, otherwise the honest parties abort.

6.3 Guaranteed Output Delivery

This leads us to the next property: guaranteed output delivery. The honest parties receive the correct outputs if at least t parties follow the protocol. As mentioned in Section 5.1, an adversary can be blocking and delay a protocol run indefinitely. This can be remedied if enough compute parties are honest or if the bulletin board is honest.

Honest Parties	Adversary-Controlled Parties	Secrecy	Correctness	Guaranteed Output Delivery	Fairness
$\geq t$	$\geq t$	no	yes [†]	yes [‡]	yes
$\geq t$	$< t$	yes [†]	yes [†]	yes [‡]	yes
$\geq t$	0	yes [†]	yes	yes	yes
$< t$	$\geq t$	no	abort ^{*†}	no	no
$< t$	$< t$	yes	abort ^{*†}	no	no
$< t$	0			n/a	
0	$\geq t$	no	no ^{**}	no	no
0	$< t$			n/a	
0	0			n/a	

[†] except with negligible probability

[‡] if broadcast channels exist or the bulletin board is not blocked

* Honest parties get either the correct output or abort.

** Any party can still verify if a result is correct or not.

Table 6.1: Overview of the security analysis.

6.4 Fairness

For fairness, we get the following: The adversary always learns the computation result. This can be split in two cases for actual implementations. If the adversary controls at least t parties, all inputs are known and the result could be computed without honest parties. Otherwise,¹ there are enough honest parties to finish the protocol and the adversary learns the result this way. In the case of honest minorities, we cannot achieve fairness and honest parties only get their outputs if not too many corrupted parties misbehave.

To sum this up: We can guarantee the classical MPC security properties in the honest majority case. On the other hand, secrecy is achieved if less than t parties are corrupted. Correctness (in the sense of honest parties not receiving wrong outputs) can always be achieved. The other security properties cannot be guaranteed for honest minorities, but they are still achievable if not too many corrupted parties misbehave. See also Table 6.1 for an overview.

Note that we did not investigate independence of inputs. In fact, the protocol (and functionality) as presented in Section 4.2 do not accomplish this security property: A corrupted input party could – for example – send $2 \otimes \langle x \rangle$ where $\langle x \rangle$ is the cipher for the input of an honest party. Intuitively, this can be fixed by letting the input parties prove knowledge for the cipher they publish. For the application to e-voting systems, this is not an issue as the input is given (in form of a ciphertext) not by a single party but by combining all votes (see Chapter 8 for more details).

¹We assume that either the number of honest parties or the number of corrupted parties is at least t .

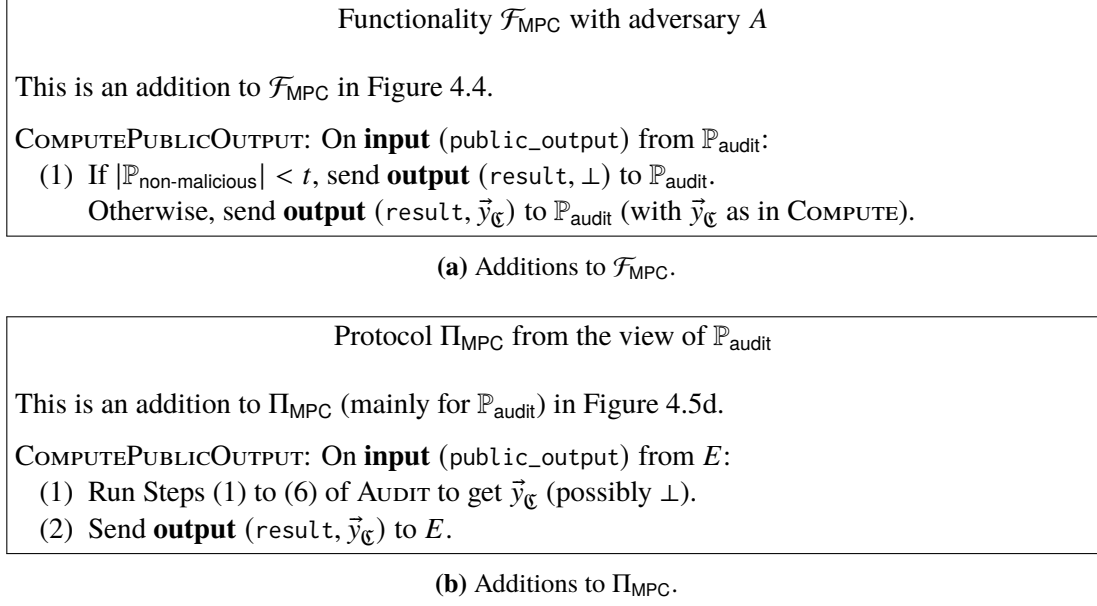


Figure 6.1: Additions to functionality and protocol for the accountability goal.

While the achieved security properties might not sound very promising in case of honest minorities, if a certain class of adversaries is assumed, all security properties can be still achieved. Risk-avoiding adversaries (as already mentioned in Section 5.7) will not try to manipulate a protocol run if they are likely to be caught. Exactly this is the case with our protocol, as it is not only verifiable but also provides accountability. We will prove accountability in the following.

6.5 Accountability

To show accountability, we have to define a goal for our protocol. Intuitively, this should be: “The output of the protocol is the evaluation of the arithmetic circuit.” Unfortunately, we have multiple “outputs” in our protocol and we want our goal to be applicable to the actual protocol Π_{MPC} and the ideal functionality \mathcal{F}_{MPC} . For this, we add a “public output”. With some additions (see Figure 6.1) we can define the output of **COMPUTEPUBLICOUTPUT** as the public output of the protocol or the functionality. Note that this can be run after **COMPUTE** and is only a minor addition and should not influence the results of Section 4.2 w. r. t. universal composability.

For an arbitrary arithmetic circuit \mathcal{C} with inputs $x_1, \dots, x_{\#\text{inputs}}$ and $\vec{y}_{\mathcal{C}} = \mathcal{C}(x_1, \dots, x_{\#\text{inputs}})$, we can define the following accountability constraint $C_{\text{MPC}} = (\gamma_{\text{MPC}}, \phi_{\text{MPC}, 1}, \dots, \phi_{\text{MPC}, n})$ with

$$\begin{aligned} \gamma_{\text{MPC}} &= \{R_{\Pi} \mid I_{\Pi}(1^n) \mapsto \{\mathbb{P}_{\text{audit}} : \vec{y} \mid \vec{y} = \vec{y}_{\mathcal{C}}\} \text{ in } R_{\Pi}\} \\ \phi_{\text{MPC}, i} &= \text{DIS}(\mathbb{P}_i), \end{aligned}$$

e. g.,

$$C_{\text{MPC}} = \gamma_{\text{MPC}} \Rightarrow \text{DIS}(\mathbb{P}_1) \mid \dots \mid \text{DIS}(\mathbb{P}_n)$$

where Π can be either $\Pi_{\text{MPC}}^{\text{wrap}}$ or $\Pi_{\text{MPC}}^{\text{ideal}}$, the adversary A of I_{Π} gives \mathfrak{C} and $x_1, \dots, x_{\#\text{inputs}}$ to the respective parties, and the output \vec{y} of $\mathbb{P}_{\text{audit}}$ is the result of `COMPUTEPUBLICOUTPUT`. The verdict of $\mathbb{P}_{\text{audit}}$ can be extracted from the final **output** of `AUDIT`. This is either $(\text{abort}, \mathbb{P}_{\text{malicious}})$, $(\text{accept}, \vec{y}, \mathbb{P}_{\text{malicious}})$, or $(\text{reject}, \vec{y}, \mathbb{P}_{\text{malicious}})$. We define the verdict ϕ to be

$$\phi = \bigwedge_{\mathbb{P}_i \in \mathbb{P}_{\text{malicious}}} \text{DIS}(\mathbb{P}_i).$$

Note that this does not depend on the input \vec{y} of `AUDIT`. The judging procedure would be running `AUDIT` and calculate ϕ based on $\mathbb{P}_{\text{malicious}}$. With the accountability property $\Phi_{\text{MPC}} = \{C_{\text{MPC}}\}$ we can prove accountability.

Theorem 6.1 (Accountability of \mathcal{F}_{MPC}).

In Π as described above, the judge $J = \mathbb{P}_{\text{audit}}$ ensures $(\Phi_{\text{MPC}}, 0)$ -accountability.

Proof. We only prove accountability for the functionality. How we can get accountability for the protocol (using UC-security) is outlined at the end of the chapter.

Fairness. For fairness, we can see that $\mathbb{P}_{\text{malicious}} \subseteq \mathbb{P}_{\text{corrupt}}$ (with $\mathbb{P}_{\text{corrupt}} \cap \mathbb{P}_{\text{honest}} = \emptyset$), e. g., honest parties are never (wrongly) accused of cheating.

Completeness. For completeness (with a bound of 0), we have to show that

$$(6.1) \Pr[I_{\Pi}(1^n) \mapsto \{\mathbb{P}_{\text{audit}} : \phi \mid \phi \not\equiv C_{\text{MPC}}\}]$$

is negligible. Recall that C_{MPC} requires that \vec{y} of `COMPUTEPUBLICOUTPUT` equals $\vec{y}_{\mathfrak{C}}$ of `COMPUTE`. This is only not the case if $|\mathbb{P}_{\text{non-malicious}}| < t$: In this case, $\vec{y} = \perp \neq \vec{y}_{\mathfrak{C}}$. From this we get

$$\begin{aligned} |\mathbb{P}_{\text{malicious}}| &= |\mathbb{P} \setminus \mathbb{P}_{\text{non-malicious}}| \\ &= n - |\mathbb{P}_{\text{non-malicious}}| \\ |\mathbb{P}_{\text{malicious}}| &> n - t \geq 0, \end{aligned}$$

e. g., that $\mathbb{P}_{\text{malicious}}$ is not empty. This means, if the goal of C_{MPC} is not met, the verdict always implies some formula of the constraint. In other words, $\mathbb{P}_{\text{audit}}$ always ensures C_{MPC} and the probability of Equation (6.1) is 0 for the ideal functionality \mathcal{F}_{MPC} . \square

Note that we could have used different goals and formulas for C_{MPC} . The goal we used (the output is the correctly evaluated circuit) and the definition of accountability make no implications about the verdicts of $\mathbb{P}_{\text{audit}}$ if the goal is met (except that no honest party is accused of cheating). This means, as long as the public result is correct, $\mathbb{P}_{\text{audit}}$ would not have to blame any malicious party. So even the modifications presented in Section 5.7 (only checking individual shares if the reconstruction has no valid decommitment) provide accountability.

6.6 Verifiability

Next, we prove verifiability. For this, we use the same goal γ_{MPC} as for accountability in Section 6.5. The verifiability assumption we use is

$$\varphi_{\text{MPC}} = \text{HON}(J) \wedge \bigvee_{\substack{\mathbb{P}' \subseteq \mathbb{P} \\ |\mathbb{P}'| \geq t}} \bigwedge_{\mathbb{P}_i \in \mathbb{P}'} \text{HON}(\mathbb{P}_i).$$

The judge J for accountability is defined similar to the one for accountability. There, we ran `AUDIT` of $\mathbb{P}_{\text{audit}}$ to get the set $\mathbb{P}_{\text{malicious}}$. Now, we do the same, but output `accept` if $|\mathbb{P}_{\text{non-malicious}}| = |\mathbb{P} \setminus \mathbb{P}_{\text{malicious}}| \geq t$. Otherwise, we return `reject`.

Theorem 6.2 (Verifiability of \mathcal{F}_{MPC}).

In Π as described in Theorem 6.1, the goal γ_{MPC} is guaranteed by φ_{MPC} and J (as described above) ensures 0-verifiability.

Proof. Recall that we consider only \mathcal{F}_{MPC} for the proof (as in Theorem 6.1).

Property 1. φ_{MPC} is constructed in a way that the formula is true iff. t or more parties are honest. This is exactly the condition for J to return `accept`, e. g., $\Pr[I_{\Pi}(1^n) \mapsto \{J : \phi \mid \phi = \text{accept}\}] = 1$ for all instances I_{Π} .

Property 2. Next, we have to show that $\Pr[I_{\Pi}(1^n) \mapsto \{J : \phi \mid \phi = \text{accept}\}, R_{\Pi} \notin \gamma_{\text{MPC}}]$ is negligible. Recall that we only output `accept` if $|\mathbb{P}_{\text{non-malicious}}| \geq t$. This also implies that the public output of the functionality is correct (e. g., `COMPUTEPUBLICOUTPUT` returns $\vec{y}_{\mathbb{C}}$). This means that the goal γ_{MPC} is always fulfilled if J outputs `accept`. In other words, $\Pr[I_{\Pi}(1^n) \mapsto \{J : \phi \mid \phi = \text{accept}\}, R_{\Pi} \notin \gamma_{\text{MPC}}] = 0$. \square

Remark. If we would have used a different accountability property in Theorem 6.1, we could use the fact that certain types of accountability imply some verifiability [81, 82].

Note that until now, we did not consider protocol runs that do not terminate, e. g., where the adversary is blocking (see Section 5.1). If we now change φ_{MPC} to

$$\varphi'_{\text{MPC}} = \text{HON}(J) \wedge \bigvee_{\substack{\mathbb{B}' \subseteq \mathbb{B} \\ |\mathbb{B}'| \geq \frac{2|\mathbb{B}|}{3}}} \bigwedge_{\mathbb{B}_i \in \mathbb{B}'} \text{HON}(\mathbb{B}_i) \wedge \bigvee_{\substack{\mathbb{P}' \subseteq \mathbb{P} \\ |\mathbb{P}'| \geq t}} \bigwedge_{\mathbb{P}_i \in \mathbb{P}'} \text{HON}(\mathbb{P}_i)$$

where we now consider that $\mathcal{F}_{\text{bulletin}}$ is implemented by the parties \mathbb{B} (\mathbb{B} is not necessarily disjoint from \mathbb{P}). Together with the results from Section 5.1, we get that – under this new assumption φ'_{MPC} – the protocol is verifiable and terminates if the assumption holds true. This is because corrupted parties can no longer block the broadcasting via the bulletin board and at least t parties are honest. This means, all honest parties always get the required messages from the honest parties and have enough information to finish the computation even if – for example – corrupted parties refuse to send their shares for openings.

Using UC-Emulation with the KTV-Framework

Here, we will show that we can use the results for functionalities in the KTV-framework (i. e., accountability and verifiability as shown in Sections 6.5 and 6.6 for \mathcal{F}_{MPC}) for the respective protocols. We use the fact that the functionalities are UC-realized by the protocols. The results of this section also give us the ability to replace the sub-functionalities (i. e., \mathcal{F}_{key}) with the respective protocol (presented in Section 4.3).

First, note that we have only probabilities of the form

$$\Pr[I_{\Pi}(1^n) \mapsto \{J : y \mid P(y)\}]$$

in the properties that we have to show for accountability (Definition 2.24) and verifiability (Definition 2.25). More specifically, we have to show that for a protocol (not a functionality) and any adversary A (a protocol instance I_{Π} includes an adversary; in our case we use Π^{wrap} instead of Π as the wrapping is necessary to let the honest protocol be corrupted)

$$\Pr[(A \parallel \Pi^{\text{wrap}})(1^n) \mapsto \{J : y \mid P(y)\}]$$

is negligible, overwhelming, or δ -bounded. We now simply split the adversary in the part that interacts with the protocol via **input** or **output** tapes and the part that uses **side-channel** tapes to interact with machines (this can be done by constructing an adversary E for every adversary A and forwarding all **side-channel** messages of A now as **input** or **output** to an adversary D that sends them to the protocol machines). Obviously, this does not change the behavior of the interactions. Especially, A (now included as a part of E) has the same view on the interaction and all messages sent and received by protocol machines are the same. Now, we have

$$(6.2) \quad \Pr[(E \parallel D \parallel \Pi^{\text{wrap}})(1^n) \mapsto \{J : y \mid P(y)\}].$$

If we prove accountability or verifiability in the KTV-framework for a functionality (or its corresponding ideal protocol), we have

$$\forall I_{\Pi^{\text{ideal}}} : \quad \Pr[I_{\Pi^{\text{ideal}}}(1^n) \mapsto \{J : y \mid P(y)\}] \leq f(\eta)$$

where f is negligible, overwhelming, or δ -bounded. With “ \leq ” we mean “ \leq ” for negligible or δ -bounded f and “ \geq ” for an overwhelming f . This can be transformed to

$$\forall E \forall A : \quad \Pr[(E \parallel A \parallel \Pi^{\text{ideal}})(1^n) \mapsto \{J : y \mid P(y)\}] \leq f(\eta)$$

which implies

$$(6.3) \quad \forall E : \quad \Pr[(E \parallel S \parallel \Pi^{\text{ideal}})(1^n) \mapsto \{J : y \mid P(y)\}] \leq f(\eta).$$

As Π^{wrap} UC-emulates Π^{ideal} , we know that the real world and the ideal world are indistinguishable, e. g.,

$$\exists S \forall E : \quad \left| \Pr[(E \parallel D \parallel \Pi^{\text{wrap}})(1^n) = \text{real}] - \Pr[(E \parallel S \parallel \Pi^{\text{ideal}})(1^n) = \text{real}] \right| \leq g(\eta)$$

for a negligible function g . Now, we split E in two parts: F interacts with D or S and the protocol. G gives the final **output** (e. g., ideal or real). Obviously, an environment E' that only looks at the outputs (of a single machine) is worse at distinguishing Π^{wrap} and Π^{ideal} . Formally we get this because the probability is negligible for all environments, including $E' = (F, G)$ where G simply checks $P(y)$:

$$\begin{aligned} \exists S \forall E : & \quad \left| \Pr[(E \parallel D \parallel \Pi^{\text{wrap}})(1^\eta) = \text{real}] - \Pr[(E \parallel S \parallel \Pi^{\text{ideal}})(1^\eta) = \text{real}] \right| \leq g(\eta) \\ \exists S \forall F : & \quad \left| \Pr[(F \parallel D \parallel \Pi^{\text{wrap}})(1^\eta) \mapsto \{J : y \mid G(y) = \text{real}\}] \right. \\ & \quad \left. - \Pr[(F \parallel S \parallel \Pi^{\text{ideal}})(1^\eta) \mapsto \{J : y \mid G(y) = \text{real}\}] \right| \leq g(\eta) \\ \exists S \forall F : & \quad \left| \Pr[(F \parallel D \parallel \Pi^{\text{wrap}})(1^\eta) \mapsto \{J : y \mid P(y)\}] \right. \\ & \quad \left. - \Pr[(F \parallel S \parallel \Pi^{\text{ideal}})(1^\eta) \mapsto \{J : y \mid P(y)\}] \right| \leq g(\eta). \end{aligned}$$

This can be transformed to:

$$(6.4a) \quad \exists S \forall F : \quad \Pr[(F \parallel D \parallel \Pi^{\text{wrap}})(1^\eta) \mapsto \{J : y \mid P(y)\}] \\ \leq \Pr[(F \parallel S \parallel \Pi^{\text{ideal}})(1^\eta) \mapsto \{J : y \mid P(y)\}] + g(\eta)$$

$$(6.4b) \quad \exists S \forall F : \quad \Pr[(F \parallel D \parallel \Pi^{\text{wrap}})(1^\eta) \mapsto \{J : y \mid P(y)\}] \\ \geq \Pr[(F \parallel S \parallel \Pi^{\text{ideal}})(1^\eta) \mapsto \{J : y \mid P(y)\}] - g(\eta).$$

Now, we have everything to continue with Equation (6.2):

$$\begin{aligned} \forall E : & \quad \Pr[(E \parallel D \parallel \Pi^{\text{wrap}})(1^\eta) \mapsto \{J : y \mid P(y)\}] \\ & \stackrel{(6.4)}{\leq} \Pr[(E \parallel S \parallel \Pi^{\text{ideal}})(1^\eta) \mapsto \{J : y \mid P(y)\}] \pm g(\eta) \\ & \stackrel{(6.3)}{\leq} f(\eta) \pm g(\eta). \end{aligned}$$

This is negligible, overwhelming, or δ -bounded in the same way that f is (e. g., the same as for the functionality).²

To sum this up, we showed that the probabilities used in the KTV-framework for accountability and verifiability differ by only a negligible amount for a functionality and the protocol that UC-realizes it. We can use this to show accountability or verifiability for an ideal functionality (e. g., Π^{ideal}) and have the same properties also apply to the protocol (e. g., Π^{wrap}) that realizes the functionality. In our case, we can prove Corollaries 6.3 and 6.4.

Corollary 6.3 (Accountability of Π_{MPC}).

$J = \mathbb{P}_{\text{audit}}$ ensures $(\Phi_{\text{MPC}}, 0)$ -accountability for Π_{MPC} .

Proof. This follows from Theorem 6.1 and the observation about using universal composability with the KTV-framework. \square

²Again, “ \leq ” and “+” (of “ \pm ”) is used for negligible or δ -bounded functions, and “ \geq ” together with “−” for overwhelming ones.

Corollary 6.4 (Verifiability of Π_{MPC}).

The goal γ_{MPC} is guaranteed by φ_{MPC} and $J = \mathbb{P}_{\text{audit}}$ ensures 0-verifiability in Π_{MPC} .

Proof (Sketch). This follows from Theorem 6.2 and the observation about using universal composition with the KTV-framework. Technically, we also have to prove that probabilities like the one in Property 2 of the verifiability definition (Definition 2.25) differ by only a negligible amount in the real and ideal world. \square

Additionally, the same argument can be used together with composition in the UC-framework to argue that we can simply replace the sub-functionalities with their respective protocols and still have accountability or verifiability. This means, we get accountability and verifiability for $\Pi_{\text{MPC}}^{\mathcal{F}_{\text{rand-cipher}} \rightarrow \Pi_{\text{rand-cipher}}, \mathcal{F}_{\text{com-cipher}} \rightarrow \Pi_{\text{com-cipher}}, \mathcal{F}_{\text{key}} \rightarrow \Pi_{\text{key}}, \mathcal{F}_{\text{bulletin}} \rightarrow \Pi_{\text{bulletin}}}$ as well.

7 Comparison to Other MPC Protocols

In this chapter we compare our protocol to other verifiable and accountable version of SPDZ (as mentioned in Section 3.1). The main focus here is on the communication volume and computational complexity (an overview is given in Table 7.2) in the online phase of the protocols, but other properties are discussed as well. Table 7.1 should give an intuition why the computational complexity differs for the protocols: Protocols with MACs have good best-case runtime, while the individual commitments for shares are used for individual accountability (which is related to the higher worst-case runtime needed to identify cheaters).

As mentioned already, the protocols in Section 3.1.4 do not give us verifiability or accountability in the fully malicious setting. Additionally, the communication cost (i. e., per multiplication) is higher by a factor of n , as each party sends a commitment to each other party.

Compared to the other protocols (described in the following), the communication cost of our protocol for an opening or a multiplication is the same. One share $\langle x \rangle_i$ and one commitment randomness $\langle \tilde{x} \rangle_i$ (or \tilde{x}_i) is broadcasted / published on the bulletin board. Note that the MAC checks also include communication (one commitment and share per party) that is absent in our protocol (except the modifications for full-threshold secret sharing as proposed in Section 5.8 are used).

Protocol	Private Information*	Public Information
SPDZ	$\langle x \rangle_i$ $\langle \alpha \cdot x \rangle_i$	n/a
Auditable SPDZ	$\langle x \rangle_i$ $\langle \alpha \cdot x \rangle_i$ $\langle \tilde{x} \rangle_i$ $\langle \alpha \cdot \tilde{x} \rangle_i$	$\llbracket x \rrbracket$
Accountable SPDZ	$\langle x \rangle_i$ $\langle \alpha \cdot x \rangle_i$ \tilde{x}_i	$\llbracket \langle x \rangle_1 \rrbracket, \dots, \llbracket \langle x \rangle_n \rrbracket$
Ours ¹	$\langle x \rangle_i$ \tilde{x}_i	$\llbracket \langle x \rangle_1 \rrbracket, \dots, \llbracket \langle x \rangle_n \rrbracket$
Ours ²	$\langle x \rangle_i$ \tilde{x}_i	$\llbracket \langle x \rangle_1 \rrbracket, \dots, \llbracket \langle x \rangle_n \rrbracket, \llbracket x' \rrbracket$
Ours ³	$\langle x \rangle_i$ $\langle \alpha \cdot x \rangle_i$ $\langle \tilde{x} \rangle_i$ $\langle \alpha \cdot \tilde{x} \rangle_i$	$\llbracket \langle x \rangle_1 \rrbracket, \dots, \llbracket \langle x \rangle_n \rrbracket$

* for a shared value $x \in \mathbb{X}$

* only known to party $\mathbb{P}_i \in \mathbb{P}$ (before openings)

¹ as presented in Section 4.2

² with changes from Section 5.7

³ with changes from Section 5.8

Table 7.1: Public and private information of the protocols.*

While we do support full-threshold secret sharing (see Section 5.8), our proposed protocol uses Shamir secret sharing. This is because we think accountability is a strong deterrent for parties to act maliciously. A property we sacrifice this way is secrecy.¹ If more parties are corrupted (and collaborate), shared secrets can be reconstructed and even enough private key shares are available to decrypt any input. So, we have to make sure to set the threshold t not too low. On the other hand, we get a form of fairness, as we do not have to assume that t parties are honest, but only that the sum of honest parties and parties that want to avoid being detected (while cheating) is at least t . If we would only have verifiability (as in auditable SPDZ), parties could force the protocol to abort on purpose without fearing to be detected. Note that, in some cases, we cannot simply restart the MPC protocol as suggested by Cunningham et al. [44] while removing the cheating parties. The compute parties generate the encryption keys. This means, a protocol restart implies that the input parties send their inputs again (encrypted with the new keys). Especially in e-voting, where the input parties are voters (or the inputs are dependent on the votes as described in Chapter 8), this would imply a rerun of the election.

There are several small optimizations that we did not mention yet as they do not improve the asymptotic complexity. One such change is the following: Instead of computing the circuit on full views $[x]_i \in \mathbb{S} \times \mathbb{R} \times \mathbb{C}^n$, a compute party \mathbb{P}_i could just compute it on a partial view where t instead of n commitments are used.² Now, if all t decommitments are valid at an opening, a value can be reconstructed. If some decommitment is wrong, \mathbb{P}_i can begin to recompute the circuit on other commitments until enough of the decommitments at the OPEN gate are valid or \mathbb{P}_i finds out that they have to abort. This increases the best-case runtime up to a factor of two if $t = \lceil n/2 \rceil$. While some cheaters might not get detected this way, \mathbb{P}_i or anyone else can fully audit the computation (like $\mathbb{P}_{\text{audit}}$) to identify the remaining cheaters.

7.1 Auditable SPDZ

As already mentioned, the communication complexity is almost the same in their protocol and in ours. In the full-threshold secret sharing setting, the computational complexity is the same as well (if we assume compute parties do not want to identify malicious parties).³ In our originally proposed protocol, we would have much more computational work as we compute on views $[x]_i \in \mathbb{S} \times \mathbb{R} \times \mathbb{C}^n$, e. g., additional work on n circuits that are evaluated over commitments. In Section 5.7, we described a way to reduce this to evaluating one circuit on commitments if no cheating occurs (otherwise, we go back to n circuits to find the cheater and find which shares should be ignored for the reconstruction). This means, we have a similar complexity as if each compute party also runs the auditing procedure of auditable SPDZ, but we also get accountability in case we need it (for a higher cost).

¹The other systems do not provide secrecy for $|\mathbb{P}_{\text{malicious}}| \geq t$. They simply set $t = n$. We believe, t can be set lower when having a protocol with individual accountability.

²In fact, one commitment less (e. g., $t - 1$ or $n - 1$ commitments) is enough in both cases as \mathbb{P}_i does not have to check their own commitment.

³if they do, they can just run the audit procedure

Protocol	Communication*	Computation** (No Manipulation) [†]	Computation** (Manipulation) [‡]
SPDZ	$\#\text{openings} \cdot \mathbb{S} + \mathbb{C}$	$ \mathbb{C} \cdot \mathbb{S}$	$ \mathbb{C} \cdot \mathbb{S}$
Auditable SPDZ	$\#\text{openings} \cdot (\mathbb{S} + \mathbb{R}) + \mathbb{C}$	$ \mathbb{C} \cdot (\mathbb{S} + \mathbb{R})$	$ \mathbb{C} \cdot (\mathbb{S} + \mathbb{R})$
Accountable SPDZ	$\#\text{openings} \cdot (\mathbb{S} + \mathbb{R}) + \mathbb{C}$	$ \mathbb{C} \cdot (\mathbb{S} + \mathbb{R})$	$ \mathbb{C} \cdot (\mathbb{S} + \mathbb{R} + n \cdot \mathbb{C})$
Ours ¹	$\#\text{openings} \cdot (\mathbb{S} + \mathbb{R})$	$ \mathbb{C} \cdot (\mathbb{S} + \mathbb{R} + n \cdot \mathbb{C})$	$ \mathbb{C} \cdot (\mathbb{S} + \mathbb{R} + n \cdot \mathbb{C})$
Ours ²	$\#\text{openings} \cdot (\mathbb{S} + \mathbb{R})$	$ \mathbb{C} \cdot (\mathbb{S} + \mathbb{R} + \mathbb{C})$	$ \mathbb{C} \cdot (\mathbb{S} + \mathbb{R} + n \cdot \mathbb{C})$
Ours ³	$\#\text{openings} \cdot (\mathbb{S} + \mathbb{R}) + \mathbb{C}$	$ \mathbb{C} \cdot (\mathbb{S} + \mathbb{R})$	$ \mathbb{C} \cdot (\mathbb{S} + \mathbb{R} + n \cdot \mathbb{C})$

* for a circuit $\mathbb{C}: \mathbb{X}^{\#\text{inputs}} \rightarrow \mathbb{X}^{\#\text{outputs}}$ with $\#\text{multiplications}$ multiplications, $\#\text{openings} = 2 \cdot \#\text{multiplications} + \#\text{outputs}$, and $|\mathbb{C}| \geq \#\text{multiplications}$ (see Section 5.9); given in big O notation

* \mathbb{S} etc. mean the size of a single element of \mathbb{S}

** \mathbb{S} etc. mean the cost of doing an operation on elements of \mathbb{S}

[†] no compute parties misbehaves, e. g., the best-case runtime

[‡] some compute parties misbehave, e. g., the worst-case runtime

¹ as presented in Section 4.2

² with changes from Section 5.7

³ with changes from Section 5.8

Table 7.2: Complexity* of the protocols in the online phase (per compute party).

7.2 Accountable SPDZ

Our protocol and the accountable version of SPDZ are quite similar. If we use full-threshold secret sharing (e. g., use our optimization with MACs as in Section 5.8), we would do (almost) the same computational work as them. We also check the MACs of the randomness to avoid cases where a protocol run is accepted by compute parties (as the MAC check is valid), but auditing parties reject the run (if some randomness is wrong and some decommitment is wrong as a consequence). Similarly, our suggested improvement in Section 5.7 to only compute the circuit on n commitments if cheating is detected is similar to how they use the MACs and fall back to commitments of all shares if there were malicious actions. The main difference is that they only check the MACs at the end and we would check one commitment at each opening. As already mentioned, we check the commitments eagerly to have the option to reconstruct values even if some shares are wrong. This is not possible with full-threshold secret sharing but with Shamir secret sharing. The main difference of the protocols is using different secret sharing schemes. We choose to use Shamir secret sharing to achieve some kind of fairness. Especially in the honest majority setting, we achieve fairness. Otherwise, we do that if we can assume that accountability keeps corrupted parties from cheating. As mentioned before, we also extend their work by showing that not only the Pedersen commitment scheme can be used.

8 Application to E-Voting Systems

The usefulness of our MPC protocol for e-voting systems should be quite apparent. As the compute parties only need encrypted inputs from the input parties to evaluate any arithmetic circuit on them and encrypted votes is exactly what most e-voting systems have in common. Additionally, we provided several ways to speed up the computation of the result while still maintaining accountability (while identifying malicious parties might be still much slower). We start by looking at Ordinos, where any MPC protocol that meets certain criteria can be used as a drop-in replacement for their MPC protocol. We do this in Section 8.2. Then we look at other e-voting systems in Section 8.3. But first, we examine how our protocol can be coupled with existing e-voting systems by pointing out which parts of the system can be reused to implement our protocol.

8.1 Reusing Existing Components

Obviously, it is beneficial to have as few components in a system as possible. For example, a system with fewer components can be reasoned about more easily because the complexity of the system (and the code) is smaller than in a system with components that all have to interact with one another. As already mentioned, the e-voting systems use a bulletin board to store encrypted votes on. Our MPC protocol could reuse this bulletin board and store all public information on it as well.

The e-voting systems we mentioned use the popular ElGamal (Helios and ElectionGuard) or Paillier (Ordinos) encryption schemes. Our protocol needs a different encryption scheme as we have to be able to multiply ciphers, but we only do so in the preprocessing to get multiplication triples. As mentioned in Section 5.4, we can use another encryption scheme for generating triples than for running the rest of the protocol. This would allow us to keep the online phase of the protocol similar to the existing system, e. g., by using the same (tested components for the) encryption scheme that is already in use. Additionally, we could completely get rid of SHE using ciphertext multiplication protocols to generate the triples [40]. Then, no changes to the encryption components are necessary (but the preprocessing might be less efficient without SHE).

Finally, a commitment scheme has to be chosen for the system. By Theorem 2.7 we know that we could reuse the existing encryption schemes as commitment schemes. This way, one less component (a commitment component) is needed to use our protocol with an e-voting system. Additionally, some parts of the protocol might be simplified if the same scheme is used as encryption scheme and commitment scheme at the same time. For example, the proof of committed ciphers (Section 2.6.10) could be replaced by a single proof of plaintext-knowledge.¹

¹Recall that we need that $\langle x \rangle$ and $\llbracket x \rrbracket$ are computed for the same plaintext x , and that the commitment-randomness \tilde{x} is known. With an encryption scheme as commitment scheme, this can become the following: A party has to prove that $\llbracket x \rrbracket = \langle x \rangle = \text{ENC}_k(x, \tilde{x})$, while proving knowledge for x and \tilde{x} . This is just a proof of plaintext-knowledge.

8.2 Ordinos

As mentioned by Küsters et al. [79], any (general) MPC protocol with the following properties can be used with Ordinos to evaluate the function to compute the election result:

1. It must fit the client-server model [46].
2. It must be publicly verifiable.
3. It must be accountable (or provide identifiable abort).
4. The properties must hold if all compute parties are malicious.

Our MPC protocol satisfies all these requirements.

The first property is fulfilled as the input parties (Figure 4.5a) are not required as long as a cipher of the input is available to the compute parties (Figure 4.5b). Even if we consider voters as actively participating input parties, their only job would be to publish an encrypted vote. After that, they could still audit the protocol but are not required to do anything for the protocol.

As mentioned in Sections 6.5 and 6.6, our protocol is publicly verifiable and accountable. Cheating compute parties (e. g., trustees of the e-voting system) are individually identifiable. The protocol provides the $(\Phi_{\text{MPC}}, 0)$ -accountability needed for Ordinos, e. g., the goal is that the output of the multi-party computation is correct and the probability of failing to blame cheaters is negligible. Also, the verifiability assumption is compatible with the one of Ordinos: The judge and the bulletin board have to be honest – in addition to at least t trustees. Accountability and verifiability still hold when all compute parties are corrupted.

If the bulletin board is implemented with separate parties (as mentioned in Section 6.6) and the bulletin board is assumed to be honest (as in Ordinos), we can even guarantee output delivery if the majority of trustees is honest as well (otherwise we could also not guarantee correct results).

This means, we can use our MPC protocol instead of the one they use while retaining an end-to-end verifiable e-voting system with accountability. The protocol they used focuses on comparisons compute functions like finding the top m candidates. Our protocol is more flexible in what computation are possible, but the general-purpose nature of our protocol in addition to a focus on arithmetic circuits (instead of Boolean ones) might turn out to be detrimental for e-voting systems. One might have to compare the runtime of a system using our protocol to find out if it is currently suitable to be used in practice. For certain functions that have to be evaluated to get an election result, our protocol might still be one of the few possibilities to conduct a tally-hiding election.

8.3 Other E-Voting Systems

Our protocol is also compatible with the systems mentioned in Sections 3.2.1 and 3.2.3. In these systems, the votes are publicly available in encrypted form on a bulletin board. Additionally, it can be checked by everyone if a vote is malformed and voters can check if their vote is included on the bulletin board.

Now, one could let the trustees sum up the valid votes homomorphically (as in Ordinos or ElectionGuard) or even take all valid votes as input of our protocol (similar to Helios). Our protocol then handles the case where malicious trustees want to change the election result. As our protocol is publicly verifiable and accountable, the correctness of the election result can be checked, and malicious trustees can be identified (and punished or excluded from future elections).

A combination with Helios could be done in two ways: 1. The votes are shuffled and revealed, or 2. the election result is computed and then revealed. The second option yields a similar system to Ordinos and we can simply use our MPC protocol as described there. The first option is not tally-hiding anymore, but one could use an MPC protocol to shuffle the votes as well [12]. With $\#voters$ votes, auditable SPDZ could run in $O(\#voters \cdot \log(\#voters) \cdot \eta_1^2)$ time compared to $O(n \cdot \#voters \cdot \eta_2^3)$ with mix-nets [5, 12]. Here, η_1 is the (statistical) security parameter that might be one or two orders of magnitude smaller than η_2 – the (computational) security parameter used for the public key cryptography of the mix-net [12]. If we would use our system instead of auditable SPDZ, we would get the factor n back. If no cheating occurs (or the probability for this is low as cheating is discouraged by accountability), we can use some optimizations (see Chapter 7 and more specifically Table 7.2). With these, we can be as efficient as auditable SPDZ (with changes from Section 5.8) or similarly efficient – e. g., not dependent on n (with changes from Section 5.7).

Alternatively, one could compute a more complex result function directly on the encrypted votes. This would be tally-hiding, but the MPC protocol has now $\#voters \cdot \#candidates$ inputs instead of $\#candidates$, e. g., one input for each vote of each voter instead of one input for each candidate / choice that corresponds to the aggregated vote of all voters for this candidate / choice. One advantage of this is that more complex result functions are possible, but the complexity will also influence the computational work to compute it and to audit it.

9 Conclusion and Outlook

In this master’s thesis, we introduced an accountable multi-party computation protocol that can achieve fairness in an honest majority setting. To our knowledge we are the first to do so. Based on previous work [79], we showed how this MPC protocol can be used to enable tally-hiding e-voting. The function that is evaluated to get the election result can be anything (that an MPC protocol can compute) and only as much as necessary about the actual votes is revealed.

The protocol is secure in the UC-framework. Accountability and verifiability were proven in the KTV-framework. Anyone can audit our protocol – even if all parties involved in the computation were acting maliciously. While not as efficient as comparable MPC protocols [12, 44] in the dishonest majority setting, our protocol can be sped up in the cases that no party cheats (we assume this is more likely to happen as our protocol achieves individual accountability).

To be used in e-voting systems, we only need a bulletin board and votes have to be given in an encrypted form. Additionally, the trustees have to hold shares of the private key of the encryption scheme. Even encryption schemes that are only additive homomorphic can be supported if a protocol to multiply ciphertexts is employed in the preprocessing phase. An implementation can choose from a wide range of commitment schemes that enable accountability and verifiability.

Outlook

While we can perform any operation on field elements and check the results with commitments, the security of our protocol depends on the inability of adversaries to break the binding property of the commitment scheme. This, in turn, depends on the size of the underlying spaces. To be more versatile and support Boolean circuits (which in turn can be used to do computations on power-of-two integers), we would generate a large overhead by using $\mathbb{X} = \mathbb{Z}_2$ and a large commitment space \mathbb{C} (and large \mathbb{R}). With small \mathbb{C} (and \mathbb{R}), the decommitments are easy to forge, which is obviously not an option. For SPDZ there have been proposals how to handle small spaces and still have secure MACs [13, 34, 49], so it would be of interest to work out similar techniques that work with commitments.

Another area of interest is achieving accountability without commitments. While it is possible [13, 70] (see Section 3.1.4), these approaches have some disadvantages such as being not secure if all compute parties are corrupt or sending different MACs to each party.

Lastly, one could investigate how general MPC protocols like ours compare to more specialized ones (such as the one used in Ordinos [79]). Here, one could also benchmark our protocol to see how the complexity differences of various protocols impact real world performance. A comparison of our protocol to ones that use full-threshold secret sharing would be of interest as well to see the “price” of achieving fairness.

Bibliography

- [1] M. Abe, S. Fehr. “Adaptively Secure Feldman VSS and Applications to Universally-Composable Threshold Cryptography”. In: *Annual International Cryptology Conference*. Springer, 2004, pp. 317–334 (cit. on p. 98).
- [2] B. Adida. *Helios Election System*. GitHub. <https://github.com/benadida/helios-server>. 2020 (cit. on p. 69).
- [3] B. Adida. *Helios Voting*. <https://heliosvoting.org/>. 2020 (cit. on p. 69).
- [4] B. Adida. “Helios: Web-based Open-Audit Voting”. In: *USENIX Security Symposium*. Vol. 17. 2008, pp. 335–348 (cit. on pp. 62, 68, 69).
- [5] M. Ajtai, J. Komlós, E. Szemerédi. “An $O(n \log n)$ Sorting Network”. In: *Symposium on Theory of Computing*. ACM, 1983, pp. 1–9 (cit. on p. 121).
- [6] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, V. Vaikuntanathan. *Homomorphic Encryption Security Standard*. Tech. rep. HomomorphicEncryption.org, 2018 (cit. on p. 56).
- [7] J. Algesheimer, J. Camenisch, V. Shoup. “Efficient Computation Modulo a Shared Secret with Application to the Generation of Shared Safe-Prime Products”. In: *Annual International Cryptology Conference*. Springer, 2002, pp. 417–432 (cit. on p. 70).
- [8] E. Alkim, L. Ducas, T. Pöppelmann, P. Schwabe. “Post-Quantum Key Exchange—A New Hope”. In: *USENIX Security Symposium*. Vol. 25. 2016, pp. 327–343 (cit. on p. 98).
- [9] B. Applebaum, Y. Ishai, E. Kushilevitz. “How to Garble Arithmetic Circuits”. In: *Journal on Computing* 43.2 (2014), pp. 905–929 (cit. on p. 56).
- [10] D. W. Archer, D. Bogdanov, Y. Lindell, L. Kamm, K. Nielsen, J. I. Pagter, N. P. Smart, R. N. Wright. “From Keys to Databases—Real-World Applications of Secure Multi-Party Computation”. In: *The Computer Journal* 61.12 (2018), pp. 1749–1771 (cit. on p. 55).
- [11] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, M. Strand. *A Guide to Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2015/1192. <https://eprint.iacr.org/2015/1192>. 2015 (cit. on p. 56).
- [12] C. Baum, I. Damgård, C. Orlandi. “Publicly Auditable Secure Multi-Party Computation”. In: *Security and Cryptography for Networks*. Springer, 2014, pp. 175–196 (cit. on pp. 24, 55, 67, 73, 77, 78, 83, 86, 94, 99, 103, 104, 121, 123, 133).
- [13] C. Baum, E. Orsini, P. Scholl. “Efficient Secure Multiparty Computation with Identifiable Abort”. In: *Theory of Cryptography Conference*. Springer, 2016, pp. 461–490 (cit. on pp. 68, 123, 133).
- [14] D. Beaver. “Efficient Multiparty Protocols Using Circuit Randomization”. In: *Advances in Cryptology*. Springer, 1991, pp. 420–432 (cit. on pp. 28, 83).

- [15] M. Bellare, V. T. Hoang, P. Rogaway. “Foundations of Garbled Circuits”. In: *Conference on Computer and Communications Security*. ACM, 2012, pp. 784–796 (cit. on p. 56).
- [16] A. Ben-Efraim, M. Nielsen, E. Omri. “Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing”. In: *Applied Cryptography and Network Security*. Springer, 2019, pp. 530–549 (cit. on p. 66).
- [17] M. Ben-Or, S. Goldwasser, A. Wigderson. “Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation”. In: *Symposium on Theory of Computing*. ACM, 1988, pp. 1–10 (cit. on p. 57).
- [18] J. Benaloh. *ElectionGuard Preliminary Specification v0.85*. GitHub. <https://github.com/microsoft/electionguard/wiki/Informal/ElectionGuardSpecificationV0.85.pdf>. 2020 (cit. on pp. 62, 70).
- [19] J. Benaloh. “Simple Verifiable Elections”. In: *EVT 6* (2006), pp. 5–5 (cit. on p. 62).
- [20] J. Benaloh. “Verifiable Secret-Ballot Elections”. PhD thesis. Yale University, 1987 (cit. on p. 62).
- [21] J. Benaloh, T. Moran, L. Naish, K. Ramchen, V. Teague. “Shuffle-Sum: Coercion-Resistant Verifiable Tallying for STV Voting”. In: *Transactions on Information Forensics and Security* 4.4 (2009), pp. 685–698 (cit. on p. 62).
- [22] R. Bendlin, I. Damgård, C. Orlandi, S. Zakarias. “Semi-Homomorphic Encryption and Multiparty Computation”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2011, pp. 169–188 (cit. on pp. 17, 68).
- [23] D. Bogdanov, P. Laud, J. Randmets. “Domain-Polymorphic Programming of Privacy-Preserving Applications”. In: *Workshop on Programming Languages and Analysis for Security*. ACM, 2014, pp. 53–65 (cit. on p. 105).
- [24] J. W. Bos, C. Costello, M. Naehrig, D. Stebila. “Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem”. In: *Symposium on Security and Privacy*. IEEE, 2015, pp. 553–570 (cit. on p. 98).
- [25] G. Bracha. “An Asynchronous $\lfloor (n - 1)/3 \rfloor$ -Resilient Consensus Protocol”. In: *Symposium on Principles of Distributed Computing*. ACM, 1984, pp. 154–162 (cit. on p. 97).
- [26] G. Bracha, S. Toueg. “Asynchronous Consensus and Broadcast Protocols”. In: *Journal of the ACM* 32.4 (1985), pp. 824–840 (cit. on p. 97).
- [27] Z. Brakerski, C. Gentry, V. Vaikuntanathan. “(Leveled) Fully Homomorphic Encryption Without Bootstrapping”. In: *Transactions on Computation Theory* 6.3 (2014), pp. 1–36 (cit. on pp. 17, 35).
- [28] Z. Brakerski, V. Vaikuntanathan. “Efficient Fully Homomorphic Encryption from (Standard) LWE”. In: *Journal on Computing* 43.2 (2014), pp. 831–871 (cit. on p. 35).
- [29] Z. Brakerski, V. Vaikuntanathan. “Fully Homomorphic Encryption From Ring-LWE and Security for Key Dependent Messages”. In: *Annual Cryptology Conference*. Springer, 2011, pp. 505–524 (cit. on p. 35).
- [30] R. Canetti. *Universally Composable Security: A New Paradigm for Cryptographic Protocols*. Cryptology ePrint Archive, Report 2000/067. <https://eprint.iacr.org/2000/067>. 2000 (cit. on pp. 40, 41, 43, 44, 52, 134).

- [31] R. Canetti, T. Rabin. “Fast Asynchronous Byzantine Agreement with Optimal Resilience”. In: *Symposium on Theory of Computing*. ACM, 1993, pp. 42–51 (cit. on p. 97).
- [32] D. Chaum, C. Crépeau, I. Damgård. “Multiparty Unconditionally Secure Protocols”. In: *Symposium on Theory of Computing*. ACM, 1988, pp. 11–19 (cit. on p. 57).
- [33] D. Chaum, T. P. Pedersen. “Wallet Databases with Observers”. In: *Advances in Cryptology*. Springer, 1992, pp. 89–105 (cit. on p. 54).
- [34] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, A. Nof. “Fast Large-Scale Honest-Majority MPC for Malicious Adversaries”. In: *Advances in Cryptology*. Springer, 2018, pp. 34–64 (cit. on pp. 66, 123).
- [35] I. Chillotti, N. Gama, M. Georgieva, M. Izabachène. “A Homomorphic LWE based E-Voting Scheme”. In: *Post-Quantum Cryptography*. Springer, 2016, pp. 245–265 (cit. on p. 71).
- [36] V. Cortier, D. Galindo, R. Küsters, J. Mueller, T. Truderung. “SoK: Verifiability Notions for E-Voting Protocols”. In: *Security and Privacy*. IEEE, 2016, pp. 779–798 (cit. on pp. 64, 70).
- [37] V. Cortier, J. Lallemand. “Voting: You can’t have privacy without individual verifiability”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 53–66 (cit. on p. 64).
- [38] R. Cramer, I. Damgård, D. Escudero, P. Scholl, C. Xing. “SPD \mathbb{Z}_2^k : Efficient MPC mod 2^k for Dishonest Majority”. In: *Advances in Cryptology*. Springer, 2018, pp. 769–798 (cit. on p. 66).
- [39] R. Cramer, I. Damgård, U. Maurer. “General Secure Multi-Party Computation from Any Linear Secret-Sharing Scheme”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2000, pp. 316–334 (cit. on p. 28).
- [40] R. Cramer, I. Damgård, J. B. Nielsen. “Multiparty Computation from Threshold Homomorphic Encryption”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2001, pp. 280–300 (cit. on pp. 100, 119).
- [41] R. Cramer, I. Damgård, B. Schoenmakers. “Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols”. In: *Annual International Cryptology Conference*. Springer, 1994, pp. 174–187 (cit. on p. 70).
- [42] R. Cramer, R. Gennaro, B. Schoenmakers. “A Secure and Optimally Efficient Multi-Authority Election Scheme”. In: *Advances in Cryptology*. Springer, 1997, pp. 103–118 (cit. on p. 34).
- [43] C. Culnane, S. Schneider. “A Peered Bulletin Board for Robust Use in Verifiable Voting Systems”. In: *Computer Security Foundations Symposium*. IEEE, 2014, pp. 169–183 (cit. on p. 98).
- [44] R. Cunningham, B. Fuller, S. Yakoubov. “Catching MPC Cheaters: Identification and Ppenability”. In: *International Conference on Information Theoretic Security*. Springer, 2017, pp. 110–134 (cit. on pp. 24, 67, 78, 79, 94, 103, 116, 123, 133).
- [45] I. Damgård. *Lectures on Data Security: Modern Cryptology in Theory and Practice*. Vol. 1561. Springer, 1999 (cit. on p. 21).
- [46] I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, T. Toft. “Confidential Benchmarking Based on Multiparty Computation”. In: *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 169–187 (cit. on p. 120).

- [47] I. Damgård, M. Jurik. “A Generalisation, a Simplification and some Applications of Paillier’s Probabilistic Public-Key System”. In: *Practice and Theory in Public Key Cryptography*. Springer, 2001, pp. 119–136 (cit. on pp. 33, 54).
- [48] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, N. P. Smart. “Practical Covertly Secure MPC for Dishonest Majority – Or: Breaking the SPDZ Limits”. In: *European Symposium on Research in Computer Security*. Springer, 2013, pp. 1–18 (cit. on pp. 35, 66, 98).
- [49] I. Damgård, V. Pastro, N. Smart, S. Zakarias. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: *Advances in Cryptology*. Springer, 2012, pp. 643–662 (cit. on pp. 17, 21, 33, 35, 36, 65, 68, 78, 83, 86, 99, 123, 133).
- [50] W. Diffie, M. Hellman. “New Directions in Cryptography”. In: *Transactions on Information Theory* 22.6 (1976), pp. 644–654 (cit. on p. 34).
- [51] T. ElGamal. “A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms”. In: *Transactions on Information Theory* 31.4 (1985), pp. 469–472 (cit. on pp. 33, 34).
- [52] A. Fiat, A. Shamir. “How to Prove Yourself: Practical Solutions to Identification and Signature Problems”. In: *Advances in Cryptology*. Springer, 1986, pp. 186–194 (cit. on p. 51).
- [53] M. J. Flynn. “Very High-Speed Computing Systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909 (cit. on p. 35).
- [54] J. A. Garay, J. Katz, R. Kumaresan, H.-S. Zhou. “Adaptively Secure Broadcast, Revisited”. In: *Symposium on Principles of Distributed Computing*. ACM, 2011, pp. 179–186 (cit. on p. 97).
- [55] J. A. Garay, P. MacKenzie, K. Yang. “Strengthening Zero-Knowledge Protocols Using Signatures”. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2003, pp. 177–194 (cit. on p. 52).
- [56] R. Gennaro, S. Jarecki, H. Krawczyk, T. Rabin. “Secure Applications of Pedersen’s Distributed Key Generation Protocol”. In: *Cryptographers’ Track at the RSA Conference*. Springer, 2003, pp. 373–390 (cit. on p. 98).
- [57] R. Gennaro, S. Jarecki, H. Krawczyk, T. Rabin. “Secure Distributed Key Generation for Discrete-Log Based Cryptosystems”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 295–310 (cit. on p. 98).
- [58] C. Gentry. “Fully Homomorphic Encryption Using Ideal Lattices”. In: *Symposium on Theory of Computing*. ACM, 2009, pp. 169–178 (cit. on p. 56).
- [59] C. Gentry, S. Halevi, N. P. Smart. “Fully Homomorphic Encryption with Polylog Overhead”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2012, pp. 465–482 (cit. on p. 35).
- [60] C. Gentry, A. Sahai, B. Waters. “Homomorphic Encryption From Learning With Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based”. In: *Annual Cryptology Conference*. Springer, 2013, pp. 75–92 (cit. on pp. 17, 35).
- [61] K. Gjøsteen, M. Strand. “A Roadmap to Fully Homomorphic Elections: Stronger Security, Better Verifiability”. In: *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 404–418 (cit. on p. 71).

-
- [62] O. Goldreich. *Foundations of Cryptography (Fragments of a Book)*. 1995 (cit. on pp. 21, 26).
- [63] O. Goldreich, H. Krawczyk. “On the Composition of Zero-knowledge Proof Systems”. In: *Journal on Computing* 25.1 (1996), pp. 169–192 (cit. on p. 52).
- [64] O. Goldreich, S. Micali, A. Wigderson. “How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority”. In: *Symposium on Theory of Computing*. ACM, 1987, pp. 218–229 (cit. on p. 57).
- [65] O. Goldreich, S. Micali, A. Wigderson. “Proofs that Yield Nothing but Their Validity or All Languages in NP have Zero-Knowledge Proof Systems”. In: *Journal of the ACM* 38.3 (1991), pp. 690–728 (cit. on p. 47).
- [66] S. Goldwasser, Y. Lindell. “Secure Multi-Party Computation without Agreement”. In: *Journal of Cryptology* 18.3 (2005), pp. 247–287 (cit. on pp. 57, 97).
- [67] S. Goldwasser, S. Micali, R. L. Rivest. “A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks”. In: *Journal on Computing* 17.2 (1988), pp. 281–308 (cit. on p. 70).
- [68] D. M. Gordon. “A Survey of Fast Exponentiation Methods”. In: *J. Algorithms* 27.1 (1998), pp. 129–146 (cit. on p. 25).
- [69] Homomorphic Encryption Standardization. *Introduction*. <https://homomorphicencryption.org/introduction/>. 2020 (cit. on p. 56).
- [70] Y. Ishai, R. Ostrovsky, V. Zikas. “Secure Multi-Party Computation with Identifiable Abort”. In: *Annual Cryptology Conference*. Springer, 2014, pp. 369–386 (cit. on pp. 68, 123).
- [71] J. Katz, Y. Lindell. *Introduction to Modern Cryptography*. CRC Press, 2014 (cit. on pp. 21, 26).
- [72] M. Keller, V. Pastro, D. Rotaru. “Overdrive: Making SPDZ Great Again”. In: *Advances in Cryptology*. Springer, 2018, pp. 158–189 (cit. on pp. 66, 100).
- [73] A. Kiayias, A. Kuldmää, H. Lipmaa, J. Siim, T. Zacharias. “On the Security Properties of e-Voting Bulletin Boards”. In: *International Conference on Security and Cryptography for Networks*. Springer, 2018, pp. 505–523 (cit. on p. 98).
- [74] A. Kiayias, H.-S. Zhou, V. Zikas. “Fair and Robust Multi-Party Computation Using a Global Transaction Ledger”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2016, pp. 705–734 (cit. on p. 97).
- [75] J. Kilian. “Founding Cryptography on Oblivious Transfer”. In: *Symposium on Theory of Computing*. ACM, 1988, pp. 20–31 (cit. on p. 55).
- [76] N. Koblitz. “Elliptic Curve Cryptosystems”. In: *Mathematics of Computation* 48.177 (1987), pp. 203–209 (cit. on p. 34).
- [77] S. Kremer, M. Ryan, B. Smyth. “Election Verifiability in Electronic Voting Protocols”. In: *European Symposium on Research in Computer Security*. Springer, 2010, pp. 389–404 (cit. on p. 63).
- [78] R. Küsters, T. Truderung, A. Vogt. “Clash Attacks on the Verifiability of E-Voting Systems”. In: *Symposium on Security and Privacy*. IEEE, 2012, pp. 395–409 (cit. on pp. 63, 64).

- [79] R. Küsters, J. Liedtke, J. Müller, D. Rausch, A. Vogt. *Ordinos: A Verifiable Tally-Hiding E-Voting System*. Cryptology ePrint Archive, Report 2020/405. <https://eprint.iacr.org/2020/405>. 2020 (cit. on pp. 19, 61–64, 68–70, 98, 101, 120, 123, 133).
- [80] R. Küsters, J. Müller, E. Scapin, T. Truderung. “sElect: A Lightweight Verifiable Remote Voting System”. In: *Computer Security Foundations Symposium*. IEEE, 2016, pp. 341–354 (cit. on pp. 71, 101).
- [81] R. Küsters, T. Truderung, A. Vogt. “Accountability: Definition and Relationship to Verifiability”. In: *Conference on Computer and Communications Security*. ACM, 2010, pp. 526–535 (cit. on pp. 17, 58, 70, 111, 134).
- [82] R. Küsters, T. Truderung, A. Vogt. *Accountability: Definition and Relationship to Verifiability*. Cryptology ePrint Archive, Report 2010/236. <https://eprint.iacr.org/2010/236>. 2010 (cit. on p. 111).
- [83] Y. Lindell, A. Lysyanskaya, T. Rabin. “On the Composition of Authenticated Byzantine Agreement”. In: *Journal of the ACM* 53.6 (2006), pp. 881–917 (cit. on p. 97).
- [84] M. Lindeman, P. B. Stark. “A Gentle Introduction to Risk-Limiting Audits”. In: *Security and Privacy* 10.5 (2012), pp. 42–49 (cit. on p. 70).
- [85] H. Lipmaa, T. Toft. “Secure Equality and Greater-Than Tests with Sublinear Online Complexity”. In: *International Colloquium on Automata, Languages, and Programming*. Springer, 2013, pp. 645–656 (cit. on p. 70).
- [86] A. López-Alt, E. Tromer, V. Vaikuntanathan. “On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption”. In: *Symposium on Theory of Computing*. ACM, 2012, pp. 1219–1234 (cit. on p. 56).
- [87] C. Luck, H. Marques. “Zum ‘Schweizer’ Cybervoting: das Vertrauensproblem bleibt ungelöst”. In: *Datenschleuder* 100 (2019), pp. 1–6 (cit. on p. 71).
- [88] V. Lyubashevsky, C. Peikert, O. Regev. “On Ideal Lattices and Learning with Errors Over Rings”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 1–23 (cit. on pp. 35, 36).
- [89] U. Maurer. “Secure Multi-Party Computation Made Simple”. In: *Discrete Applied Mathematics* 154.2 (2006), pp. 370–381 (cit. on p. 28).
- [90] Microsoft. *ElectionGuard*. GitHub. <https://github.com/microsoft/electionguard>. 2020 (cit. on pp. 62, 68, 70).
- [91] J. C. Mitchell, R. Sharma, D. Stefan, J. Zimmerman. “Information-Flow Control for Programming on Encrypted Data”. In: *Computer Security Foundations Symposium*. IEEE, 2012, pp. 45–60 (cit. on p. 105).
- [92] J. Müller. “Design and Cryptographic Security Analysis of E-Voting Protocols”. PhD thesis. Universität Stuttgart, 2019 (cit. on p. 71).
- [93] C. A. Neff. “A Verifiable Secret Shuffle and Its Application to E-Voting”. In: *Computer and Communications Security*. ACM, 2001, pp. 116–125 (cit. on p. 62).
- [94] J. D. Nielsen, M. I. Schwartzbach. “A Domain-Specific Programming Language for Secure Multiparty Computation”. In: *Workshop on Programming Languages and Analysis for Security*. ACM, 2007, pp. 21–30 (cit. on p. 105).

-
- [95] T. Nishide, K. Sakurai. “Distributed Paillier Cryptosystem without Trusted Dealer”. In: *International Workshop on Information Security Applications*. Springer, 2010, pp. 44–60 (cit. on p. 98).
- [96] P. Paillier. “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 223–238 (cit. on pp. 34, 70).
- [97] M. Pease, R. Shostak, L. Lamport. “Reaching Agreement in the Presence of Faults”. In: *Journal of the ACM* 27.2 (1980), pp. 228–234 (cit. on p. 97).
- [98] T. P. Pedersen. “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing”. In: *Advances in Cryptology*. Springer, 1991, pp. 129–140 (cit. on pp. 39, 40).
- [99] C. Peikert. “A Decade of Lattice Cryptography”. In: *Foundations and Trends in Theoretical Computer Science* 10.4 (2016), pp. 283–424 (cit. on p. 35).
- [100] C. Peikert. “Lattice Cryptography for the Internet”. In: *International Workshop on Post-Quantum Cryptography*. Springer, 2014, pp. 197–219 (cit. on p. 98).
- [101] R. del Pino, V. Lyubashevsky, G. Seiler. “Short Discrete Log Proofs for FHE and Ring-LWE Ciphertexts”. In: *Public-Key Cryptography*. Springer, 2019, pp. 344–373 (cit. on pp. 53, 54, 99).
- [102] D. Pointcheval, J. Stern. “Security Proofs for Signature Schemes”. In: *Advances in Cryptology*. Springer, 1996, pp. 387–398 (cit. on p. 51).
- [103] B. Preneel. “Analysis and Design of Cryptographic Hash Functions”. PhD thesis. Katholieke Universiteit te Leuven, 1993 (cit. on p. 26).
- [104] M. O. Rabin. *How to Exchange Secrets with Oblivious Transfer. Technical Report TR-81*. Tech. rep. Aiken Computation Lab, Harvard University, 1981 (cit. on p. 55).
- [105] A. Rastogi, M. A. Hammer, M. Hicks. “Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations”. In: *Symposium on Security and Privacy*. IEEE, 2014, pp. 655–670 (cit. on p. 105).
- [106] O. Regev. “On Lattices, Learning with Errors, Random Linear Codes, and Cryptography”. In: *Journal of the ACM* 56.6 (2009), pp. 1–40 (cit. on p. 35).
- [107] P. Rogaway, T. Shrimpton. “Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance”. In: *International Workshop on Fast Software Encryption*. Springer, 2004, pp. 371–388 (cit. on p. 26).
- [108] K. Sako. “Verifiable Encryption”. In: *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 1356–1357 (cit. on p. 30).
- [109] B. Schoenmakers, M. Veeningen. “Universally Verifiable Multiparty Computation From Threshold Homomorphic Cryptosystems”. In: *International Conference on Applied Cryptography and Network Security*. Springer, 2015, pp. 3–22 (cit. on p. 70).
- [110] A. Shamir. “How to Share a Secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613 (cit. on p. 27).
- [111] N. P. Smart, F. Vercauteren. “Fully Homomorphic SIMD Operations”. In: *Designs, Codes and Cryptography* 71.1 (2014), pp. 57–81 (cit. on p. 35).

- [112] Swiss Post. *Online Voting in Switzerland*. <https://www.evoting.ch/en>. 2020 (cit. on p. 71).
- [113] Y. Tsiounis, M. Yung. “On the Security of ElGamal Based Encryption”. In: *International Workshop on Public Key Cryptography*. Springer, 1998, pp. 117–134 (cit. on p. 34).
- [114] B. Wang, J. Sun, Y. He, D. Pang, N. Lu. “Large-Scale Election Based on Blockchain”. In: *Procedia Computer Science* 129 (2018), pp. 234–237 (cit. on p. 71).
- [115] D. Wikström. “A Commitment-Consistent Proof of a Shuffle”. In: *Australasian Conference on Information Security and Privacy*. Springer, 2009, pp. 407–421 (cit. on p. 62).
- [116] D. Wikström. “Universally Composable DKG with Linear Number of Exponentiations”. In: *International Conference on Security in Communication Networks*. Springer, 2004, pp. 263–277 (cit. on p. 98).
- [117] A. C.-C. Yao. “How to Generate and Exchange Secrets”. In: *Symposium on Foundations of Computer Science*. IEEE, 1986, pp. 162–167 (cit. on p. 56).

All links were last followed on June 19, 2020.

A Accountable Secure Multi-Party-Computation für Tally-Hiding E-Voting

Zusammenfassung

Diese Masterarbeit beschäftigt sich mit Secure Multi-Party-Computation (MPC) und deren Nutzen für E-Voting. Wir betrachten MPC-Protokolle, die Accountability bieten. In solchen Protokollen ist es möglich, Manipulationsversuche zu erkennen. Dabei können Parteien, die dies versuchen, eindeutig identifiziert (und dafür belangt) werden. Damit bieten solche Protokolle stärkere Garantien als solche, die nur Verifiability bieten. Die Erwartung ist, dass dies abschreckend wirkt und in Systemen, die MPC-Protokolle mit Accountability benutzen, weniger Manipulationen stattfinden.

Außerdem bieten sich MPC-Protokolle (vor allem mit Accountability) für die Benutzung in E-Voting-Systemen, die tally-hiding sein sollen, an. Tally-hiding bezeichnet Systeme, bei denen die ausgezählten Stimmen nicht (oder nur teilweise) bekannt werden. In vielen anderen E-Voting-Systemen werden alle (anonymisierten) Stimmen oder die ausgezählten Stimmen veröffentlicht, wonach jeder das Wahlergebnis (z. B. die Zusammenstellung eines Parlaments) selbst berechnen kann. Mit einem tally-hiding System könnte stattdessen das Wahlergebnis direkt berechnet werden. Damit bleiben die Stimmen (zu einem größtmöglichen Teil) geheim. Ordinos [79] bietet bereits ein E-Voting-System mit End-To-End-Verifiability an, wobei jedes MPC-Protokoll mit bestimmten Voraussetzungen (z. B. Accountability) zur Berechnung des Wahlergebnisses benutzt werden kann. In dieser Arbeit stellen wir ein Protokoll vor, das diese Voraussetzungen erfüllt und beliebige Funktionen berechnen kann.

Unser Protokoll baut auf einem verbreiteten MPC-Protokoll auf: dem SPDZ-Protokoll [49]. Wie vorheriger Erweiterungen, die Auditability zu SPDZ hinzugefügt haben [12], erweitern wir dies um Accountability. Einige Versionen von SPDZ mit Accountability existieren zwar bereits [13, 44], jedoch bieten diese entweder nicht alle Voraussetzungen für Ordinos oder sie sind nicht direkt auf E-Voting-Systeme ausgelegt. Eines dieser Protokolle [44] weist deutliche Ähnlichkeiten zu unserem Protokoll auf, jedoch bietet es keine Fairness (wenn die Mehrheit der beteiligten Parteien nicht versucht die Berechnung zu manipulieren). Die Option, eine fehlgeschlagene MPC-Berechnung ohne die Parteien, die versucht haben, das Protokoll zu manipulieren, zu wiederholen, scheint für E-Voting-Systeme äußerst ungeeignet, da die Wahl dann wiederholt werden müsste. Deshalb bieten wir Fairness, wenn weniger als die Hälfte der Beteiligten versuchen zu manipulieren. Wir hoffen, dass die Abschreckungsfunktion von Accountability groß genug ist, um dies zu gewährleisten.

Wie SPDZ setzen wir auf Secret-Sharing. Hierbei erhält jede beteiligte Partei (Compute-Party) einen Share für jeden Eingabewert des MPC-Protokolls. Die Compute-Parties können dann die gewünschte Funktion auf ihrem Share berechnen.¹ Am Ende kann das Ergebnis aus den Shares wiederhergestellt werden. Hierbei werden mindestens t von n Shares benötigt.² Weniger Shares enthalten keinerlei Informationen über den geshareten Wert. Damit alle Parties ihren Share preisgeben und keinen falschen Share, der das Ergebnis verfälschen würde, benutzen wir Commitments. Anders als für Auditability benötigen wir nicht nur ein Commitment für jeden Eingabewert, um festzustellen, ob das Ergebnis korrekt war – wir benutzen ein Commitment für jeden Share (jedes Eingabewerts). Damit können wir die Berechnung für jede Party überprüfen und erhalten Accountability.

Anders als die bisherigen Protokolle unterstützt unser MPC-Protokoll eine Vielzahl an Commitment-Schemes. In der Masterarbeit findet sich ein Beweis für die Security unseres Protokolls im UC-Framework [30]. Außerdem werden Accountability und Verifiability im KTV-Framework [81] analysiert (und bewiesen). Damit weist unser Protokoll alle Voraussetzungen auf, um in Kombination mit Ordinos (oder anderen E-Voting-Systemen) sichere und überprüfbare Wahlen durchzuführen.

¹Ein Trick und Interaktionen zwischen den Parties sind erforderlich, um Multiplikationen von Werten zu unterstützen.

² n ist die Zahl der Compute-Parties. $t = \lceil n/2 \rceil$ entspricht dem Fall, dass mindestens die Hälfte aller Shares benötigt wird.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature