Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Analysis of Selected Cryptographic Protocols with DY*

Samuel Holderbach

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Tim Würtele, M.Sc.
Commenced:	January 3, 2023
Completed:	July 3, 2023

Abstract

DY* is a framework implemented in the proof-oriented programming language F*, aiming at symbolic analysis of cryptographic protocols on the structural and on the implementation level.

In this master's thesis, we analyse three selected authentication and key exchange protocols with DY*: the *Otway-Rees protocol*, the *Yahalom protocol* and the *Denning-Sacco protocol with public keys*. Each of these protocols is designed to establish a secure channel between two users while involving a trusted third party in the authentication process. The Otway-Rees and Yahalom protocols rely on pre-shared symmetric keys with this trusted third party, while the Denning-Sacco protocol relies on digital signatures and public key encryption. In addition, the Denning-Sacco protocol proposes the use of timestamps in messages to provide users with guarantees about the timeliness of the conversation, a protocol feature that has not yet been attempted to be modeled and analyzed in DY*.

We developed accurate models for each of the three protocols in DY*, documented possible attacks and proposed improvements to prevent them, and finally proved the security of the protocol or its improved version. We found several attacks on the Otway-Rees protocol that allow an adversary to impersonate one or possibly both of the users involved in the protocol, and based on these attacks, presented improvements to prevent them. For the Yahalom protocol, we show that it satisfies security goals derived from its formal specification, and draw parallels to other approaches with similar results. We also comment on the differences between our results and those of other analyses that describe the Yahalom protocol as flawed. Moreover, we developed an extension to DY* for modeling time-based properties of protocols with timestamps and demonstrated it on the Denning-Sacco protocol. As a result, we provide the first symbolic security proof, including timestamp-dependent security properties, of the Denning-Sacco protocol in DY*.

Kurzfassung

DY* ist ein in der beweisorientierten Programmiersprache F* implementiertes Framework zur symbolischen Analyse von kryptographischen Protokollen auf der Struktur- und Implementierungsebene.

In dieser Masterarbeit analysieren wir drei ausgewählte Authentifizierungs- und Schlüsselaustauschprotokolle mit DY*: das *Otway-Rees Protokoll*, das *Yahalom Protokoll* und das *Denning-Sacco Protokoll mit öffentlichen Schlüsseln*. Jedes dieser Protokolle ist darauf ausgelegt, einen sicheren Kanal zwischen zwei Benutzern aufzubauen und dabei eine vertrauenswürdige dritte Partei in den Authentifizierungsprozess einzubeziehen. Die Protokolle Otway-Rees und Yahalom beruhen auf gemeinsam genutzten symmetrischen Schlüsseln mit dieser vertrauenswürdigen dritten Partei, während das Denning-Sacco Protokoll auf digitalen Signaturen und Verschlüsselung mit öffentlichen Schlüsseln beruht. Darüber hinaus schlägt das Denning-Sacco Protokoll die Verwendung von Zeitstempeln in Nachrichten vor, um den Benutzern Garantien in Bezug auf die Aktualität der Konversation zu geben – eine Protokolleigenschaft, die in DY* noch nicht modelliert und analysiert wurde.

Wir haben präzise Modelle für jedes der drei Protokolle in DY* entwickelt, mögliche Angriffe dokumentiert und Verbesserungen vorgeschlagen, um sie zu verhindern, und schließlich die Sicherheit des Protokolls oder seiner verbesserten Version bewiesen. Wir konnten mehrere Angriffe auf das Otway-Rees Protokoll finden, die es einem Angreifer ermöglichen, sich als einer oder möglicherweise als beide am Protokoll beteiligten Benutzer auszugeben, und haben auf der Grundlage dieser Angriffe Verbesserungen zur Verhinderung derselben vorgestellt. Für das Yahalom Protokoll zeigen wir, dass es die aus seiner formalen Spezifikation abgeleiteten Sicherheitsziele erfüllt, und ziehen Parallelen zu anderen Ansätzen mit ähnlichen Ergebnissen. Wir kommentieren auch die Unterschiede zwischen unseren Ergebnissen und denen anderer Analysen, die das Yahalom Protokoll als unsicher bezeichnen. Darüber hinaus haben wir eine Erweiterung von DY* für die Modellierung zeitbasierter Eigenschaften von Protokollen mit Zeitstempeln entwickelt und diese am Denning-Sacco Protokoll demonstriert. Als Ergebnis liefern wir den ersten symbolischen Sicherheitsbeweis, einschließlich von Zeitstempeln abhängiger Sicherheitseigenschaften, für das Denning-Sacco Protokoll in DY*.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

Since Needham and Schroeder [26] proposed the use of public and symmetric key encryption to authenticate communication entities – usually referred to as principals – in large computer networks, the field has received a great amount of attention by security researchers. Many protocols that use cryptography to achieve authentication have been proposed so far and at least as many flaws and attack vectors have been found on such protocols. In a benign environment it would be rather trivial to achieve authentication but not so in the case of a large and insecure network like the Internet. In the presence of powerful attackers that may control network traffic, read and tamper with messages or try to impersonate honest principals, authentication is a hard to achieve and very subtle property.

Because of this, security researchers stressed the need of methods to verify the correctness of authentication protocols to ensure that their use in presence of such powerful attackers still leads to strong security guarantees between honest parties in the network. Methods that have been proposed so far, roughly divide into two fields. The first field is the field of *computational analysis* of the cryptographic primitives and their combination in cryptographic protocols, aiming to show their probabilistic security. This approach, however, requires a detailed understanding of these cryptographic primitives and possible attack vectors when combining them. Thus, such proofs are usually carried out in long and complex reduction or sequence of games proofs that require a huge amount of manual proof work and are hard to maintain.

The other field attempts to analyze protocols at a higher, symbolic level by making strong security assumptions on the cryptographic primitives used in a protocol and finding attacks or proving certain security properties based on the specifications of protocols. This approach is called the *symbolic analysis* approach.

Analysing Cryptographic Protocols: Symbolic Approach

Dolev and Yao [17] pioneered in the field of symbolic protocol analysis with their model of a powerful symbolic attacker controlling the network, which has been used in many symbolic approaches that incorporate an active network attacker so far. Burrows et al. [11] proposed a logic based on believes of honest principals for the symbolic analysis of authentication protocols, known as BAN logic. The intention of the authors was to provide a formal logic to reason about principal's believes throughout the execution of protocols and express authentication properties of protocols based on these believes. However, the logic was restricted to a context of honest principals, making it not so well suited when reasoning about other properties of cryptographic protocols such as secrecy. A more complete approach for security protocol verification was introduced by Ryan et al. [31] with the CSP approach. CSP is a process algebra that can be used to model parallel processes in which messages are exchanged. The work of Ryan et al. shows how CSP can be used in a trace based approach – with a trace consisting of, e.g., messages sent, random numbers generated and events that can assist in the specification of security properties – to verify the correctness and security of cryptographic protocols. Moreover, the CSP approach allows for mechanical support of proofs via the automatic model checker FDR, making it less prone to errors in the analysis and also suited for more complex protocols. The CSP approach was also used by Lowe [23] to show, with the help of FDR, that the Needham-Schroeder public key protocol that was proposed in [26] is vulnerable to a man-in-the-middle attack.

Automated Symbolic Analysis

The ever growing number of new cryptographic protocols in the last twenty years has motivated the development of fast and fully automated symbolic provers for the verification of such protocols. Earlier tools like AVISPA [35] were already able to verify industrial-scale protocols within seconds and helped in the detection of some previously unknown attacks on protocols, e.g., from the ISO-PK family or the IKEv2 protocol with digital signatures. Newer provers like TAMARIN [25] can even account for protocol features like loops or mutable global state and capture all possible execution traces of a protocol within seconds, showing that they satisfy the security properties or giving a counter example (attack on the protocol). Automated provers can do within seconds, what would have required long and error prone manual proofs otherwise. However, there is a downside to the full automation of such proofs.

Large protocols like TLS [30], which are an essential part of the infrastructure of the Internet today, must account for a variety of protocol modes and rounds. This makes it extremely inefficient or even impossible to perform an analysis of the whole protocol at once. In a non-modular approach like the one of TAMARIN, the time and memory required for the analysis of a protocol may grow exponentially with respect to the protocol's size. In ratcheting protocols like the Signal protocol [29], where each key K_{n+1} is recursively dependent on the previous key K_n, the complexity of the analysis grows with each ratcheting step. When considering an arbitrary number of rounds in the analysis, induction is required, a feature that is only supported in combination with some manual proof work in TAMARIN. Another problem with automated provers like TAMARIN is that they do not account for low-level implementation details given in the specifications of protocols that are often a root of attacks. For example, the Otway-Rees protocol [27] – one of the protocols analyzed with DY* as part of this master's thesis – has been proved secure by a number of researchers (e.g., [4]) using symbolic or partially symbolic approaches that ignore low-level implementation details. However, the implementation of the protocol based on its original specification is at least vulnerable to an impersonation attack, resulting from a missing check of the identities in the plain text part of the second message by the authentication service. This attack and other attacks on the Otway-Rees protocol are described in Section 4.3.5. A more complete explanation of the disadvantages of fully automated symbolic provers, like TAMARIN, can be read in [6].

DY* Framework

DY* [6] is a framework for the analysis and verification of cryptographic protocols and cryptographic protocol code that aims to compensate for the problems of modern fully automated verification tools. The framework combines the approaches of using dependent type systems for the modular verification of cryptographic protocols with the approach of provers like TAMARIN that support automated trace-based verification. DY* models the global runtime semantics of protocols in terms of a mutable append-only global trace that tracks – in a sequential manner – all events that occur during the concurrent execution of arbitrary protocol sessions. This explicit notion of a global trace enables properties such as the knowledge of the attacker, secrecy lemmas about keys or random nonces, as well as assumptions about cryptographic primitives to be expressed and proved in terms of this trace and within the framework. This allows users of DY* to express and prove more fine-grained security properties, where the exact order of events on the trace matters, like perfect forward secrecy or post-compromise security, with the help of features such as long-lived mutable state and dynamic compromise. Protocol steps, unlike with many fully automated provers, can be modeled in full detail accounting for implementation details like message formats, message parsing and message validation (or the lack of validation), making it suitable for investigating real-world protocol implementations regarding their security. In fact, the authors of DY* have already used the framework for a detailed symbolic security analysis of the Signal protocol [6], which is used by messenger applications like WhatsApp. As a result of this analysis, they were able to provide the first mechanized proof of forward secrecy and post-compromise security in the Signal protocol over an unbounded number of protocol rounds.

When new approaches for the symbolic analysis of cryptographic protocols arise, it is often interesting to demonstrate them on a couple of well-known protocols that are mainly of academic interest. On the one hand, to show that they are able to model basic properties of these protocols that are still found in many of today's protocols, and on the other hand, to reason about differences and similarities in the analysis results compared to other approaches. It has already been demonstrated by the authors, that one can model Lowe's attack on the NS-PK protocol in DY* and that the security of the proposed fix can be proven. However, there are still many protocols that could be interesting to reason about in DY*.

Aim of this work

The goal of this master's thesis is to provide a symbolic analysis of three selected authentication protocols, using the DY* framework. In particular, we investigate the Otway-Rees protocol that was proposed by Otway and Rees in their attempt on Efficient and Timely Mutual Authentication [27], the Yahalom protocol, first specified by Burrows et al. in [11], and the Denning-Sacco protocol with public keys that was proposed by Denning and Sacco in their work on Timestamps in Key Distribution Protocols [16]. The analysis shows that the Otway-Rees protocol, based on its original specification, is vulnerable to multiple interception and impersonation attacks and that a fix proposed by Boyd and Mao [7] is not sufficient to prevent all found attacks. Therefore, we provide a minimal fix ourselves and prove it secure in DY* with respect to the security properties intended when the protocol was designed. We also use DY* to prove that the Yahalom and Denning-Sacco protocols are secure as long as none of the parties involved leaks secrets. Additionally, we present an approach on how to include (security) properties about timestamps in our DY* models and demonstrate our approach based on the Denning-Sacco protocol by showing that session keys, that are compromised after their use, cannot be replayed.

The outline of the rest of the thesis will be as follows: In Chapter 2 we discuss basics of the DY* framework as well as authentication protocols required for the analysis. Chapter 3 gives an overview over other (earlier) symbolic approaches used for the analysis of authentication protocols with a focus on analyses of the selected protocols. In Chapter 4, we model the selected protocols in DY*, reason about the models and prove their security or provide mitigations against discovered attacks. Finally, Chapter 5 summarizes our work and provides suggestions for improvements of DY* based on the performed analyses.

2 Basics

The purpose of this chapter is to give a brief introduction to DY* and its aspects relevant for the analyses of the selected protocols performed in Chapter 4. We will, however, not cover all aspects of the framework in great detail. Thus, we refer to the paper of Bhargavan et al. [6] for a more general introduction to DY*. Furthermore, we discuss a few basics of authenticaton protocols in this chapter.

2.1 Introduction to DY*

DY* is a framework for the verification and symbolic security analysis of cryptographic protocol code written in the programming language F* [18]. The F* programming language is a purely functional programming language with effects that aims at program verification. F* programs are verified with the help of an SMT solver and can be extracted to efficient OCaml, F# or C code and then executed. The novelty of DY* is that it bridges the gap between trace-based and type-based approaches of symbolic protocol analysis by using F*'s effects to explicitly model the global trace within the framework and by making use of the type system that F* offers. This provides the ability to express fine-grained security properties for our models in terms of the global trace, using features like long-lived mutable protocol state, fine-grained dynamic corruption, and events. Since F* is a general purpose programming language, we can also use DY* to detect flaws in implementation details of protocols, for example, in serialization and deserialization of messages.

It has already been said in Chapter 1 how DY* is a modular approach for modeling and analysing cryptographic protocols. This modular approach also applies to the implementation of the framework itself. DY* is implemented via 9 verified F* modules that are organized into two layers. The lower layer is the symbolic runtime layer that provides libraries for low level features like symbolic cryptography, symbolic random number generation, storage and mutation of long-lived state (e.g., to store long term secrets or information regarding a particular protocol session) and the simulation of a network in which messages can be sent and received by principals. Moreover, it provides an API for the attacker, which is, as the name of the framework suggests, a Dolev-Yao symbolic attacker. The lower layer is already sufficient to create functional models of protocols in DY*. However, the framework also comprises a second layer on top of the symbolic runtime layer that allows for meaningful (security) proofs based on the semantics of the lower layer. Particularly, this higher layer introduces the concept of labels to ease the proof of security properties like key secrecy.

Labels are essentially an over-approximation of the intended audience of a term, say, an encryption key. We could, for example, conclude that if the intended audience of a key does not contain a corrupted principal session[1], this particular key is secret (not known to the attacker).

In the following sections, the two layers are explained in more detail based on some selected examples in order to demonstrate how the most important protocol features like (de-)serialization, encryption and decryption, random number generation and state sessions can be modeled and what their semantics are. Further, we illustrate how we can build security proofs based on these features and their semantics.

Symbolic Runtime Layer

The essence of DY* is that principals run protocol code and exchange messages over an untrusted network, controlled by the Dolev-Yao attacker, in arbitrary many parallel sessions. Everything that happens within these protocol sessions is captured by a central component: the global execution trace. In particular, the global trace records the history of all principal state sessions, generated random numbers and sent messages. Additionally, it documents corrupted state sessions or versions of state sessions. Therefore, the trace determines the attackers knowledge at any point in time during the run of a protocol, which in turn, lets us precisely express security properties in terms of the trace.

The trace is simply implemented as array of trace entries where each entry represents a protocol action:

```
noeq type entry_t =
    | RandGen: b:bytes -> l:label -> u:usage -> entry_t
    | SetState: principal -> v:version_vec -> new_state:state_vec -> entry_t
    | Corrupt: corrupted_principal:principal -> session:nat -> version:nat -> entry_t
    | Event: sender:principal -> event -> entry_t
    | Message: sender:principal -> receiver:principal -> message:bytes -> entry_t

type trace = Seq.seq entry_t
```

The `RandGen b l u` trace entry represents the generation of random `bytes b` with intended audience `l` and `usage u`. The `bytes` type is the symbolic low-level respectively trace representation of data structures that may be used in protocols, e.g., literals, nonces, public keys, ciphertexts and messages, just to name a few. Usages tell us the scope in which a random value is or may be used; examples are `nonce_usage` or `pke_usage`. Intuitively, a random `bytes` value annotated with `nonce_usage` indicates that the value is simply used as nonce, while `pke_usage` means that the value is used as private decryption key in public key encryption. Next, `SetState p v new_state` holds a newly created or updated state of a `principal p`, where `new_state` is a vector of state sessions and `v` is a vector of versions matching the state sessions by index. State sessions can be used by principals to store, e.g., long-term private keys or current data associated with a particular protocol session. Versioning can be used to enable for even more fine-grained corruption, which can be helpful in proofs of subtle security properties

[1]Principals can have multiple independent sessions within their state storage that can be corrupted by the attacker. The corruption of one session only implies that the attacker gains knowledge of the terms in that particular session. We will discuss the concept of sessions later in more detail.

of protocols like forward secrecy or post-compromise security. Corruptions are notated by the `Corrupt corrupted_principal session version` entry, specifying which exact version of a principal's session has been corrupted. The entry `Event sender event` shows the occurrence of a specific event during a protocol run; here, `sender` is the principal that "reported" the occurrence of the event. Events consist of a meaningful name and an array of associated data and can ease and assist in the formulation of security properties that require a specific order of certain protocol actions on the trace[2]. Finally, messages are represented by an entry `Message sender receiver message`, where `sender` and `receiver` are principals and `message` is usually a compound term. The symbolic runtime layer exposes an API with functions to perform the above protocol actions that will create the respective trace entries and append them to the global trace. For example, `gen l u` generates random `bytes b` with `label l` and `usage u`, creates a trace entry `RandGen b l u`, appends it to the global trace and finally outputs `b`. Analogous, there are functions to create and update a principal's state or certain state sessions, corrupt a principal's state or state sessions, trigger events or send and receive messages.

The length of the global trace grows monotonically, so its length can be used as a symbolic timestamp in invariants and constraints on the trace, for example, to ensure that an event occurred or a message was sent before a given timestamp. The symbolic runtime layer defines an effect `Crypto` to capture these trace invariants in our protocol code. Functions we define that shall respect certain invariants on the trace are then defined in the `Crypto` effect, which comes with `requires` and `ensures` clauses to require properties of the input trace and to ensure properties of the resulting trace and the result of the computation, possibly depending on the input trace.

Another central component of DY* that is part of the symbolic runtime model is the `bytes` type. It is a type for the symbolic low-level representation of data structures used in protocols; other symbolic approaches often call this concept a "term". Terms or bytes can be atomic, e.g., a nonce that has been generated with the `gen` function, or they can be compound to build serialized messages or state sessions that can be sent across the network or stored in a principal's state. The runtime layer offers `concat` and `split` functions to concatenate bytes or split them into their atomic parts.

```
type literal =
    | ByteSeq of FStar.Seq.seq FStar.UInt8.t
    | String of string
    | Nat of nat

type bytes_ =
    | Literal: literal -> bytes_
    | Rand: n:nat -> l:label -> u:usage -> bytes_
    | Concat: bytes_ -> bytes_ -> bytes_
    | PK: secret:bytes_ -> bytes_
    | PKEnc: pk:bytes_ -> n:bytes_ -> msg:bytes_ -> bytes_
    | Extract: key:bytes_ -> salt:bytes_ -> bytes_
    | Expand: key:bytes_ -> info:bytes_ -> bytes_
    | AEnc: k:bytes_ -> iv:bytes_ -> msg:bytes_ -> ad:bytes_ -> bytes_
    | VK: secret:bytes_ -> bytes_
    | Sig: sk:bytes_ -> n:bytes_ -> msg:bytes_ -> bytes_
    | Mac: k:bytes_ -> msg:bytes_ -> bytes_
    | Hash: m:bytes_ -> bytes_
    | DH_PK: s:bytes_ -> bytes_
```

[2]This can be particularly useful when showing authentication properties, as we will see later.

```
    | DH: k:bytes_ -> s:bytes_ -> bytes_
```

Atomic bytes are either literals or random values. The concrete value of a literal can be a byte sequence, a string or a natural number. This definition of literals also allows for the extension of DY* with concrete implementations of, say, cryptography, in order to proof protocols secure under consideration of computational aspects. The symbolic runtime API has functions like string_to_bytes, bytes_to_string, nat_to_bytes, bytes_to_nat, etc., for the conversion of literals from and to its concrete values for use in higher-level data structures like message implementations that might determine the control flow in protocol steps. All other constructors of bytes create higher-order terms like public keys, ciphertexts, signatures, or even messages and state sessions. Thus, bytes do not only serve as a way to represent data within protocols but also carry the semantics of symbolic cryptographic primitives and of the model itself.

To demonstrate this, let us have a look at how public key encryption has been realized in DY*:

```
val pk:bytes -> bytes // interface
let pk s = (PK s)     // implementation

val pke_enc:pub_key:bytes -> randomness:bytes -> msg:bytes -> bytes // interface
let pke_enc p n s = (PKEnc p n s)                                   // implementation

val pke_dec:priv_key:bytes -> ciphertext:bytes -> result bytes // interface
let pke_dec s c =                                              // implementation
    match c with
    | PKEnc (PK s') n m ->
        if s = s' then Success m
        else Error "pke_dec: key mismatch"
    | _ -> Error "pke_dec: not a pke ciphertext"
```

The pk function takes a secret key of type bytes as input and outputs the corresponding public key of type bytes. Under the hood, this is done by invoking the PK constructor of bytes. Symbolic public key encryption has been implemented with a function pke_enc that accepts inputs public key, randomness and some message in plaintext – all of type bytes – and outputs a ciphertext of type bytes. Analogous to pk, the encryption function simply invokes the PKEnc constructor of bytes. Thus, we need the decryption function pke_dec that only outputs the plaintext if the secret key passed as input to the function corresponds to the public key that was used for encryption. Since the actual implementation of the encryption and decryption functions is not part of the interface of the symbolic runtime layer, F* Lemmas are used to prove semantically relevant properties of these functions, like those outlined here, to the public.

For example, the pke_dec_enc_lemma ensures the correctness property of pke_enc and pke_dec:

```
val pke_dec_enc_lemma: sk:bytes -> n:bytes -> msg:bytes ->
    Lemma (pke_dec sk (pke_enc (pk sk) n msg) == Success msg)
```

From the perspective of the interface, the semantics of real and symbolic cryptography are equivalent (ignoring the negligible probability that real cryptography can be broken): We encrypt a plaintext of type bytes and receive a ciphertext of type bytes but we cannot infer the structure of the ciphertext without the help of the decryption function and the secret key because its visible structure does not deviate from that of the plaintext.

Symbolic implemenations for other cryptographic primitives also exist in the symbolic crypto API. The ones relevant for this thesis are symmetric encryption, which is realized by *Authenticated Encryption with Associated Data*[3] (AEAD) [24] in DY*, and signatures. AEAD is realized by functions `aead_enc` and `aead_dec` for authenticated encryption and decryption with a symmetric key, and signatures are created with a function `sign` and verified with a corresponding function `verify`.

```
val aead_enc: key:bytes -> iv:bytes -> msg:bytes -> ad:bytes -> bytes
val aead_dec: key:bytes -> iv:bytes -> ciphertext:bytes -> ad:bytes -> result bytes

val aead_dec_enc_lemma: k:bytes -> iv:bytes -> m:bytes -> ad:bytes ->
    Lemma (aead_dec k iv (aead_enc k iv m ad) ad == Success m)
```

The `aead_enc` function takes as input the symmetric encryption key `key`, a symbolic random initialization vector `iv` simulating randomness in the encryption, the message to encrypt denoted by `msg`, and the public associated data `ad` to be authenticated. Returned is the ciphertext resulting from the encryption. The corresponding decryption function `aead_dec` is given the key, initialization vector and associated data from the encryption, as well as the ciphertext to decrypt, and it outputs the previously encrypted plaintext. The ciphertext is returned only if the key, initialization vector, and associated data match in the encryption and decryption. The correctness property for AEAD is captured by a lemma called `aead_dec_enc_lemma`, analogous to the `pke_dec_enc_lemma` for public key cryptography.

```
val sign: priv_key:bytes -> randomness:bytes -> msg:bytes -> bytes
val verify: pub_key:bytes -> msg:bytes -> tag:bytes -> bool

val verify_sign_lemma: sk:bytes -> n:bytes -> m:bytes ->
    Lemma (verify (vk sk) m (sign sk n m) == true)
```

The `sign` function receives as input a private signing key `priv_key`, a symbolic randomness for signing, and the message to sign dentoted by `msg`. Returned is the resulting signature tag. For verification, the `verify` function accepts the public key `pub_key` corresponding to the private key used for signing, the original message, again denoted by `msg`, and the signature tag to verify denoted by `tag`. `verify` yields `true` on a valid signature, which is the case if the public key used for verification matches the private signing key and the message verified matches the message signed; otherwise it returns `false`. The `verify_sign_lemma` ensures that the verification is sound, i.e. the `verify` function must return `true` for all valid signature tags.

It has already been pointed out that the attacker in DY* has the capabilities of a Dolev-Yao style attacker [17]. That is, it is in control of the network, can read, modify, schedule or block messages, compromise state sessions or call cryptographic functions on messages it already knows. DY* offers an API for the attacker that defines messages and terms known to the attacker via a refinement type of `bytes` called `pub_bytes (i:timestamp)`. In particular, the type is defined as `t:bytes{attacker_knows_at i t}`, meaning that the attacker knows the term `t` at trace index `i`. Functions in the attacker's API wrap the functions from the symbolic runtime layer but only accept inputs of type `pub_bytes i` instead of the more general `bytes` type to ensure that the attacker uses terms already known to it when calling these functions. For example, public key encryption is defined as `val pke_enc: #i:timestamp -> pub_key:pub_bytes i -> randomness:pub_bytes i -> msg:pub_bytes i -> pub_bytes i`. Besides this

[3]AEAD combines symmetric encryption and message authentication (MACs) into a single cryptographic primitive that provides confidentiality, as well as authenticity and integrity.

subtle difference, the functions have syntax and semantics similar to the original functions used by honest principals, which demonstrates that the attacker's API provides the attacker with the same capabilities as honest principals with respect to the processing of public bytestrings; this is also called *attacker typability*. Similar to the Dolev-Yao intruder, the DY* adversary can grow its knowledge as the global trace grows by using the (cryptographic) functions in the attacker's API, e.g., to read messages that were sent on the network, split messages into their atomic parts, or even decrypt messages if the decryption key is known to the attacker. This behavior of the adversary is captured by a predicate `attacker_can_derive: idx:timestamp -> steps:nat -> t:bytes -> Type0`, which is used in statements about the derivability of a term `t` at trace index `idx` in a certain number of `steps`. If the attacker can derive a term from terms it already knows, then it holds per definition that the derived term is also part of the attackers knowledge (at a given trace index).

As we have seen so far, the symbolic runtime model already offers everything we need to create models of cryptographic protocols. A model of a protocol can be constructed with a set of functions, where each function models an honest principal taking on some role in the protocol and executing a single protocol step. Protocol steps usually include that a principal retrieves its current state associated with the protocol run; receives a message from the network; decrypts, parses and validates the message or parts of it; builds, serializes and encrypts the response; and finally updates it's state. In principle, since F* is what we would call a Turing Complete programming language, protocol steps could even perform arbitrary computations. The attacker cannot alter the protocol steps because it cannot control how honest principals behave but it can schedule the execution of protocol steps in any order and with arbitrary parameters and perform its own actions in between. From the attacker's point of view, the functions implementing the protocol steps are simply black boxes that he can use to interact with honest principals. The executable part of a model thus consists of attack scenarios, where the attacker uses the protocol step functions and its own API, trying to find a way to break security. To verify the correctness of a model it generally makes sense to also implement a function that represents a benign environment in which the attacker remains passive and schedules the protocol steps as intended.

Labeled Security Layer

In the previous section we have learned how we can use the symbolic runtime layer of DY* to create executable models of cryptographic protocols. However, our main goal is to formulate and prove security properties for our models in DY* and mechanically verify these proofs using F*'s typechecker. This venture – although feasible in theory – would still require a huge effort with the symbolic runtime model because we would have to reason about secrecy implications of cryptographic functions in the symbolic runtime library each time we want to prove a new protocol secure. For this reason, the DY* framework consists of a second layer that was build on top of the runtime layer, namely the labeled security layer. With this layer, the concept of labels is introduced, which enables the formulation of reusable statements about the secrecy implications of cryptographic primitives and other crypto-related functions in the runtime library.

Labels carry secrecy implications of terms by annotating them with the audience that is allowed to know them. This is especially useful for terms like nonces, keys or other secrets that should remain secret with respect to their intended audience. The labeled layer specifies another trace invariant, namely `valid_trace`, which ensures that the secrecy implications of labels hold at each trace index throughout protocol execution. Particularly, a valid trace is a trace containing only publishable messages, and state sessions and events valid with respect to state session and event invariants that can be specified on the level of individual protocols. The `valid_trace` invariant also holds true in presence of the DY* adversary, which was defined on the lower symbolic runtime layer. Functions on the labeled layer are executed in terms of an effect `LCrypto` that wraps the `Crypto` effect from the symbolic layer and additionally ensures the validity of the resulting trace. The `LCrypto` effect also comes with `requires` and `ensures` predicates for specifying pre- and post-conditions of functions, but has the additional advantage that the case of an erroneous computation may be ignored in the post-condition.

There are four types of labels, namely `public`, `readers (ids:list id)`, `join` and `meet`. A term annotated with the `public` label can be safely published and transferred over the network without the use of encryption. The `readers` label is used to specify the allowed readers of a term. The fine-granularity of the readers is determined by the `id` type, which specifies the exact subset of state sessions belonging to a principal that can read the annotated term; that can either be all state sessions of a principal `p` (`P p`), a specific state session `s` of `p` (`S p s`), or only a specific version `v` of a state session (`V p s v`). Lastly, the labels `join` and `meet` denote unions and intersections of labels. In addition to the labels, the labeled layer annotates terms with a usage to ensure that keys are only to be used within the context for which they were established. For example, private decryption keys should not be used for signing as this might cause confidential information to leak if the adversary gets the owner of the key to sign a ciphertext and send it back to the adversary. The labeled layer refines all the cryptographic and stateful functions, originally defined in the symbolic runtime API, to ensure the preservation of a valid trace as well as labeling and usage constraints for these functions. Protocol code that builds on these refinements implicitly preserves or ensures the secrecy and usage implications of their inputs respectively outputs. This tremendously eases the proof of security properties involving keys and secrets, that are supposed to hold at a later time; particularly, after a successful protocol run. Since the labeled layer does not restrict the DY* adversary and even ensures that the attacker does not violate the valid trace property, all security properties that can be proven on the labeled layer specifically hold in presence of the DY* adversary.

Let us take a look at our previous example of public key encryption and how it has been refined in the labeled crypto API:

```
val pke_enc: #p:global_usage -> #i:timestamp -> #nl:label ->
    public_key:msg p i public -> nonce:pke_nonce p i nl ->
    message:msg p i (get_sk_label p.key_usages public_key){
        can_flow i (get_label p.key_usages message) nl /\
        (forall s.
         is_public_enc_key p i public_key (get_sk_label p.key_usages public_key) s ==>
         pke_pred p.usage_preds i s public_key message)} ->
    msg p i public

val pke_dec: #p:global_usage -> #i:timestamp -> #l:label ->
    private_key:lbytes p i l{is_publishable p i private_key \/
                             (exists s. is_private_dec_key p i private_key l s)} ->
    ciphertext:msg p i public ->
    result (msg p i l)
```

The type `msg p i l` is a refinement type of `bytes` that restricts possible values to `bytes`, valid with respect to some usage predicate `p` at trace index `i` and annotated with a label that can flow to `l`, i.e., is covered by the allowed readers of `l`[4]. Analogously, `lbytes p i l` denotes valid bytes with exact label `l`. `is_publishable p i b` holds true if `b` has type `msg p i public`; that is, if `b`'s label can flow to the `public` label[5]. The `pke_nonce p i l` type constrains the randomness that may be used for public key encryption to `lbytes p i l` with a fixed usage string that identifies it as randomness used for public key encryption. Given arbitrary `bytes b`, `get_label` outputs the label of `b`. Similarly, `get_sk_label` outputs the label of the corresponding private key if and only if `b` is a public key. The predicate `pke_pred` is a protocol specific constraint on the key usages and keys that may be used for encryption as well as the plaintexts that may be encrypted. The `pke_enc` function of the labeled API expects implicit arguments `#p`, `#i` and `#nl`, where `#p` is a protocol specific usage predicate, `#i` is a trace index and `#nl` denotes the label of the randomness used for encryption. The key used for encryption is safely publishable and the nonce has been specifically generated to be used as randomness in public key encryption. Further, the label of the message to be encrypted can flow to the label of the private key corresponding to the public key. This is necessary so that no information is encrypted that is more confidential than the key used for decryption. Finally, if the key material used for encryption really is a public key, i.e., has been strictly generated for the purpose of being used in public key encryption, then we have that the protocol specific usage constraint specified in `pke_pred` must hold for the used public key and the encrypted message. If all the above constraints are fulfilled, then the `pke_enc` function returns a publishable ciphertext that can be safely transmitted over the network. The `pke_dec` function also expects a global usage predicate `#p` and a timestamp `#i`. The label `#l` denotes the label of the key used for decryption. The key material used for decryption must either be publishable or must really be a decryption key and the ciphertext must be publishable. If all constraints are fulfilled and the decryption was successful, then `pke_dec` returns the original plaintext, which can flow to `l` and is hence readable by the owner of the decryption key. A lemma `pke_dec_lemma` ensures that if the correct key was used for decryption and `pke_dec` yields the plaintext, then it is either publishable (e.g. if it comes from the adversary), or the `pke_pred` must hold for the public key used in the encryption and the plaintext.

The labeled layer defines similar refinements and usage constraints for AEAD and signatures. The refined `aead_enc` function requires that the key used for encryption is either publishable (for example, a key compromised by the adversary) or a valid secret key for AEAD, and that the label of the message can flow to the label of the key, i.e. that the message is not more secret than the key. Moreover, `aead_enc` requires, analogous to public key encryption, that the protocol specific usage constraint for AEAD (`aead_pred`) holds for the key and message. The corresponding decryption function `aead_dec` in conjunction with the `aead_dec_lemma` ensures that the `aead_pred` holds if the decryption was successful and the key is not publishable. Similarly, the labeled `sign` function requires that the key is publishable or a valid signing key and that `sign_pred` holds for the message to be signed and the verification key corresponding to the secret signing key if it is indeed a signing key. `sign` guarantees that both the signed message and the signature tag flow to a common label, so when signing confidential information labeled with a `readers` label, for example, the signature tag must be additionally protected by encryption under a key with a label that is at least as restrictive.

[4]It should be noted that in general, less restrictive labels can always flow to more restrictive labels but not the other way around, except for the case when the more restrictive label contains a corrupted identity.

[5]Since the can flow relation is reflexive, the `public` label can flow to itself. However, it is also possible that a more restrictive label flows to `public` if one of its readers has been corrupted.

The verify function and verify_lemma ensure that the usage predicate holds for the successfully verified message and the used verification key, or that the sign key label is publishable, which means that the signature guarantees do not apply.

In general, the functions in the labeled API are more subtle with respect to their inputs and outputs and the types carry useful information about labels and usages of terms. While the symbolic runtime layer was mostly concerned with establishing correctness, e.g., that pke_dec returns the plaintext that was originally encrypted with pke_enc on a matching key pair, the labeled security layer is mostly concerned with the secrecy and usage context of terms. We can use the usage predicates for the cryptographic primitives to explicitly pass information about label and usage of an encrypted secret, or about logged events with attached data that matches the data in the encrypted or signed message, to the principal who will receive and decrypt or verify the message. As mentioned above, the decrypting or verifying principal can infer from the lemmas pke_dec_lemma, aead_dec_lemma, or verify_lemma that the respective usage predicate holds if the decryption or verification was successful and the key is not compromised nor does the message originate from the attacker. The properties ensured by the usage predicates can thus be carried across multiple protocol steps and can then be used to formulate and proof, among other things, secrecy and authentication properties of the respective protocols. Often, these properties are defined based on the final protocol state of principals or relations of trace events, which are ensured to be valid by the implicit state and event invariants also specified on the individual protocol level. The labeled layer already yields some generic security lemmas that were proven sound once and for all based on the implementation of the labeled API. A central lemma useful for secrecy proofs is the secrecy_lemma, which states that the secrecy of a bytes value b depends on the honesty of the principals who can read b according to its label:

```
val secrecy_lemma: #pr:preds -> b:bytes ->
    LCrypto unit pr
    (requires (fun t0 -> True))
    (ensures (fun t0 _ t1 -> t0 == t1 /\
             (forall ids. (is_labeled pr.global_usage (trace_len t0) b (readers ids) /\
                     attacker_knows_at (trace_len t0) b) ==>
                     (exists id. List.Tot.mem id ids /\ corrupt_at (trace_len t0) id)))
```

Precisely, the secrecy lemma states that if b is labeled with readers ids and the attacker knows b at trace_len t0, then there must be a corrupt state session identifier within ids. Hence, we can use the lemma to derive statements about a term's secrecy if we know the label of that term.

2.2 Authentication Protocols

Authentication protocols or authentication and key exchange protocols (short AKE protocols) are protocols that aim at establishing a secure channel for communication between two or more parties as well as mutually authenticating the parties to each other, i.e., to provide each party with the correct identities of the other parties. Strictly speaking, authentication protocols do not necessarily perform key establishment between parties but for the sake of simplicity we assume so and use the term interchangeably with the term AKE protocol. In this Master Thesis, we only take a look at the case where two parties want to communicate with each other. We use the term *user* to denote any principal that participates in a protocol with the goal of establishing a communication channel with another party – apparently, also a user. Trusted third parties tasked with session key generation or public key certification, we call *authentication server* or simply *server*. The term *principal* is used for any party participating in a protocol, including all users and possible servers. There are plenty of different protocol architectures that aim to achieve authentication and key establishment. In the context of this work, they can be classified in two categories based on the question, what keys have to be established in advance to be able to run the protocol [8]. The rationale behind this question is that there is no way to securely exchange keys if there is no previously established secure channel already available for the key exchange. We call keys that are already available at the beginning of a protocol run *long-term keys* and keys that are being established as a result of a protocol run *session keys* or *communication keys*. An AKE protocol requires one of the following to provide each party with guarantees about the channel that will be established:

- symmetric long-term keys, or
- asymmetric public keys

In protocols that require symmetric long-term keys readily available, these keys are established either between

- the users directly, such that the presence of an authentication server is not required, or
- each user and an authentication server, where the server is in charge of the key exchange and often also its generation.

The latter has the advantage that users do not need to maintain long-term keys with all the other users they want to talk to, but only with a single principal who is trusted to never leak them. The two protocols investigated in this master's thesis that build on symmetric long-term secrets, i.e., the Otway-Rees [27] and Yahalom protocols [11], both rely on a trusted authentication server to generate and securely exchange sessions keys using the established long-term keys.

Similarly, in protocols that rely on public key encryption and the availability of public keys, we have that either

- users need to know the public keys of all other users they want to talk to, or
- an authentication server takes over the role of a certification authority to distribute public keys to users that wish to communicate.

Public keys have the advantage that no symmetric long-term secrets need to be established in advance. The second variant further simplifies key management because a user only needs to know their own public key and that of the authentication server. On the other hand, public key encryption adds computational overhead to the protocol [8]. The third protocol that we analyze, namely the Denning-Sacco protocol [16], falls into the second category of AKE protocols that rely on public key encryption.

3 Related Work

This chapter comprises a summary of available literature on authentication protocols and different techniques and approaches for their symbolic analysis with a special focus on the three authentication protocols that are discussed as part of this thesis. Because the analyzed protocols all date back more than three decades ago, we will mainly present literature that was released in the nineties and early two thousands. For an overview of more recent approaches with fully automated proof techniques, we refer to the DY* paper by Bhargavan et al. [6].

3.1 Authentication Protocols

Detailed introductions to the nature of authentication and key establishment protocols are given in "A Survey of Authentication Protocol Literature" by Clark and Jacob [14] (1997) and in the book "Protocols for Authentication and Key Establishment" by Boyd et al. [8] (2003). Both give a detailed overview over cryptographic tools, architectural styles, design patterns, security goals and attack types that come with such protocols and hence complement the short introduction to authentication protocols that we gave in Section 2.2. Clark and Jacob also summarize formal methods used to analyze such protocols. Further, both literatures provide large lists with descriptions of well known authentication protocols and possible attacks, including the protocols studied in this thesis. Clark and Jacob document attacks on all three protocols, while Boyd et al. only present attacks on two of them.

3.2 Symbolic Analysis Approach

The symbolic approach for the analysis of cryptographic protocols was introduced by Dolev and Yao [17] (1983) with their work on the notion of a symbolic attacker that is still used in modern symbolic provers like DY* as of today. Another milestone with respect to symbolic analysis methods is the work of Burrows et al. [11] (1989) on the BAN logic for authentication protocols. The BAN logic was the first approach to capture the goals and assumptions of authentication protocols in terms of a formal language for principal beliefs. A BAN analysis always starts with a set of initial beliefs representing a protocol's prerequisites and a set of belief goals representing the authentication goals of a protocol. Further beliefs can be derived from the initial beliefs using a fixed set of inference rules, as messages are received. This process eventually results in a set of final beliefs representing the actual assumptions that principals can make when running the protocol successfully. Burrows et al. demonstrated their logic on multiple well known authentication protocols, including the Otway-Rees and Yahalom protocols. Justified criticism against the BAN logic was mainly concerned with optimizations, which the BAN authors proposed based on their analysis results, that turned out to be less secure (e.g., [7] and [33]) and with its lack of sensibility against the permutation of protocol steps [32]. Based on this criticism, multiple extensions to the logic were proposed and later unified in the so-called SVO logic by Syverson and Van Oorschot [34] (1994). The SVO logic has been used by researchers to analyze and improve the Otway-Rees [12] and Yahalom [13] protocols, among others.

Mid and late nineties Ryan et al. [31] developed a completely new approach to the symbolic analysis of cryptographic protocols based on a process algebra named *Communicating Sequential Process* (CSP). CSP itself is a mathematical notation to describe and analyze systems of processes communicating with each other via messages. Ryan et al. present a trace-based approach that uses CSP to model the roles of principals and the messages in a protocol. Additionally, they include the notion of an active adversary that can be equipped with different capabilities. Secrecy and authentication properties are defined with the help of trace events that claim messages secret or commit to protocol runs. The verification of protocol models and security properties is automated via the model checker FDR and a compiler named Casper that translates models and properties to FDR code.

The CSP approach was followed by a whole series of new approaches with automated tool support. For example, Brackin developed an Automatic Authentication Protocol Analyzer (AAPA) [9] (1998) based on an extension of the GNY belief logic[1] [19], the BGNY logic. The AAPA tool was able to analyze 52 of the 53 protocols presented in “A Survey of Authentication Protocol Literature” (see Section 3.1) and detected 9 flaws (including a flaw on the Yahalom protocol) of which 3 flaws were previously unidentified. However, it also missed 19 flaws that Clark and Jacob had found in their protocol library, including flaws on the Otway-Rees protocol and the Denning-Sacco protocol with symmetric cryptography. Brackin was able to give an explanation for 18 of these misses and sketched how the AAPA could be further improved to account for the undetected flaws. Later fully automated provers like AVISS [3] (2002) or its successor AVISPA [35] (2006) came with an expressive formal language for the specification of security protocols and properties as well as support for multiple backends implementing different automatic analysis techniques, like protocol falsification or abstraction-based verification, to account for a large variety of protocols. Furthermore, the network over which messages are exchanged has been modeled in terms of a Dolev-Yao style intruder to simulate an inherently insecure environment for the execution of protocols. The AVISS tool could find 31 flaws in the protocol library of Clark and Jacob; this also includes a flaw in the Otway-Rees protocol and a previously unreported flaw in the Denning-Sacco protocol. The authors of AVISPA tested their tool on their own library consisting of 33 industrial-scale security protocols, which have been used to derive a total of 215 security problems. Their results include unreported flaws on protocols from the ISO-PK family as well as on the IKEv2 protocol with digital signatures. Tools like AVISPA were followed by the latest generation of fully automated provers like TAMARIN [25] (2013), which even support complex protocol features such as loops or mutable global state and capture all possible traces of a protocol in the time of a wink.

Other modern approaches try to combine the advantages of symbolic and computational methods for the analysis of cryptographic protocols. Symbolic methods treat the ability to produce much simpler and often automated proofs against weaker security guarantees because of restrictions on the capability of an adversary. On the other hand, computational methods result in powerful security guarantees in presence of a probabilistic polynomial-time attacker but proofs are usually

[1]A successor of the BAN logic that accounts for the permutation of protocol steps, motivated by [32]

complex and more prone to errors. The results of the research on unifying the two approaches and their advantages were captured in a survey by Cortier et al. [15]. The pioneers in this field were Backes et al. [5] (2003) with their proposal of an abstract cryptographic library based on Dolev-Yao style primitives, for which they proved that a real cryptographic implementation of their library is as least as secure as the abstract library itself. Their abstract library accounts for all kinds of protocol features like public-key encryption, digital signatures, nonces, list operations and application data. Furthermore, active attacks are modeled by embedding the library in a stateful Dolev-Yao system, where honest principals and the attacker can both perform any cryptographic operation that is possible based on previous events. The proof provided by Backes et al. for the security of the real cryptographic library is based on simulatability, i.e., that the abstract library simulates the real library, which is a common proof technique used in computational proofs. To proof the adequacy of their work, one of the authors (Backes) [4] used the ideal cryptographic library to obtain a cryptographically sound security proof of the Otway-Rees protocol.

3.3 Selected Protocols and Conducted Analyses

In the previous section, we gave an overview over some of the most important milestones in the symbolic approach to cryptographic protocol analysis. Many of the discussed methods have been used, among other things, for the analysis of the protocols Otway-Rees, Yahalom, and Denning-Sacco, which we investigate in this work. In the rest of this chapter, we present the three protocols and discuss and reason about the results of their analyses under consideration of the assumptions made by the used approaches and also by the authors who conducted the analyses.

3.3.1 Otway-Rees

The Otway-Rees protocol is an authentication protocol that performs a session key exchange between two users via symmetric long-term keys and a trusted authentication server. The protocol was proposed by Otway and Rees in their paper "Efficient and Timely Mutual Authentication" [27] and goes as follows:

1. $A \rightarrow B: \quad N_{cid}, A, B, \{N_a, N_{cid}, A, B\}^s_{k_{as}}$
2. $B \rightarrow AS: \quad N_{cid}, A, B, \{N_a, N_{cid}, A, B\}^s_{k_{as}}, \{N_b, N_{cid}, A, B\}^s_{k_{bs}}$
3. $AS \rightarrow B: \quad N_{cid}, \{N_a, k_{ab}\}^s_{k_{as}}, \{N_b, k_{ab}\}^s_{k_{bs}}$
4. $B \rightarrow A: \quad N_{cid}, \{N_a, k_{ab}\}^s_{k_{as}}$

The users A and B initially share symmetric long-term keys k_{as} and k_{bs} with an authentication server AS. The protocol is initiated by A who first chooses a random conversation identifier N_{cid} and a challenge N_a. A then sends N_{cid} along with its own identifier, the identifier of the responder B and a part encrypted under k_{as} that contains the initiator's challenge as well as copies of the identifiers to B. Upon receiving of the first message, the responder B chooses its own random challenge N_b, appends a part encrypted under k_{bs} with a structure similar to A's part to the message and forwards it to the server. The server uses the identifiers of A and B in the plaintext part of the second message to retrieve A's and B's long term-keys, decrypts the encrypted parts and checks whether the components N_{cid}, A and B match in these parts. If they match, it chooses a fresh key k_{ab} for communication between A and B and encrypts the key respectively with each of the principals' challenges. The server then sends the encrypted keys and challenges together with the conversation identifier N_{cid} back to the responder who decrypts his part, verifies his challenge N_b and obtains k_{ab}. Finally, the responder forwards the rest of the message to the initiator such that it can also verify its challenge and obtain the key.

Analyses and Results

The Otway-Rees protocol is one of the protocols that have been investigated in the BAN paper [11]. In the paper, a short description of the protocol is given that essentially matches the original description by Otway and Rees as well as ours. The BAN authors also account for the fact that the identifiers N_{cid}, A and B in the plaintext part of the second message are never compared with their copies in the encrypted parts. This is however disregarded in the BAN analysis, as the plaintext parts of messages are discarded in the protocol idealization process that precedes the analysis. With their BAN analysis, Burrows et al. come to the conclusion that the protocol is well designed and provides strong guarantees regarding the exchanged key and timeliness of the messages. The initial assumptions made by the authors are that A, B and the AS trust their shared long-term keys to be suitable for secret communication between them and that A and B trust in the AS to choose a key that is suitable for secret communication between A and B. From that, they conclude – using the BAN logic – that A as well as B both belief that k_{ab} is a good key for communication between them, i.e., is only known to them or parties trusted by them. Moreover, they state that B authenticates to A via the second message from B to the AS, which contains an encrypted part with A's challenge in it. However, they admit that the authentication is one sided in that B cannot say the same about A because A never sends a message containing something that B believes to be fresh.

Boyd and Mao [7] later pointed out the limitations of the BAN logic on the example of the Otway-Rees protocol. They show that the Otway-Rees protocol is not secure under the assumptions made in the BAN paper and that the optimizations proposed by Burrows et al. are dangerous in that they do not result in a secure version of the protocol either. The attack pointed out by Boyd and Mao on the original protocol is the following:

1. $A \rightarrow I(B):\quad N_{cid}, A, B, \{N_a, N_{cid}, A, B\}^s_{k_{as}}$
2. $I(B) \rightarrow AS:\quad N_{cid}, A, E, \{N_a, N_{cid}, A, B\}^s_{k_{as}}, \{N_e, N_{cid}, A, B\}^s_{k_{es}}$
3. $AS \rightarrow I(B):\quad N_{cid}, \{N_a, k_{ae}\}^s_{k_{as}}, \{N_e, k_{ae}\}^s_{k_{es}}$
4. $I(B) \rightarrow A:\quad N_{cid}, \{N_a, k_{ae}\}^s_{k_{as}}$

They explain that this attack is possible because the AS does not compare the conversation identifier N_{cid} and the user identifiers A and B in the plaintext part of the second message with their copies in the encrypted parts. Boyd and Mao see the main problem in the idealization process applied to protocols in order to derive initial assumptions for the BAN analysis. Their argument is that the implementation-level behavior of the AS does not reflect in the abstractions made by the idealization process and that an implementation-level fix to prevent the attack would not have affected the BAN analysis in any way. Based on their research, they introduce the concept of *robust* protocols as generic fix for the problem encountered in the Otway-Rees protocol and similar problems. Intuitively, robust protocols are designed such that messages contain all the information necessary to authenticate them. Boyd and Mao argue that this is not the case for the original Otway-Rees protocol since the third message, issued by the AS, does not make explicit to the users who the respective peer of the key k_{ab} is. Based on this observation, they propose the following fix for the third message:

3. $AS \rightarrow B: \quad N_{cid}, \{A, N_a, k_{ab}\}^s_{k_{as}}, \{B, N_b, k_{ab}\}^s_{k_{bs}}$

We show that the proposed fix is actually not sufficient to prevent the previously discussed impersonation attack and present our own improved version of the Otway-Rees protocol in Section 4.3.

A more recent effort to perform a symbolic analysis of the Otway-Rees protocol using a belief logic has been made by Chen [12] who proposed a new improved version of the Otway-Rees protocol and used the SVO logic [34] to show that their version fulfills the desired authentication properties. They refer to the impersonation attack detected by Boyd and Mao [7] as well as their proposed fix to include users' identities in the third message and argue that it is not sufficient to establish authentication between A and B. Chen's improved version of the Otway-Rees protocol goes as follows:

1. $A \rightarrow B: \quad N_{cid}, A, \{N_a, N_{cid}, B\}^s_{k_{as}}$
2. $B \rightarrow AS: \quad N_{cid}, A, B, \{N_a, N_{cid}, B\}^s_{k_{as}}, \{N_b, N_{cid}, A\}^s_{k_{bs}}$
3. $AS \rightarrow B: \quad N_{cid}, \{N_a, B, k_{ab}\}^s_{k_{as}}, \{N_b, A, k_{ab}\}^s_{k_{bs}}$
4. $B \rightarrow A: \quad N_{cid}, \{N_a, B, k_{ab}\}^s_{k_{as}}, \{N_b, B\}^s_{k_{ab}}$
5. $A \rightarrow B: \quad N_{cid}, \{N_b - 1, A\}^s_{k_{ab}}$

The initial assumptions that Chen defines for their improved protocol are the same as those made by Burrows et al. [11] in their BAN analysis plus the assumption that A and B trust the AS to generate a communication key that is fresh. The security goals are however slightly stronger than the goals originally intended by Otway and Rees [27]. In particular, A and B should both be convinced that the communication key k_{ab} is secret to A and B or parties they trust (including the AS), and that the key is also fresh. Moreover, users should be convinced that the other party has the same beliefs about k_{ab} as they do. Chen then goes on to show that their improved protocol is in fact secure by providing the respective SVO derivations for A and B that lead to the specified security goals. However, the improved protocol significantly deviates from the original protocol in that all four messages have been modified and two parts encrypted under k_{ab} have been added in the fourth and respectively in an additional fifth message to reassure each user that the other user is in possession of the same key and has the same beliefs about it. In Section 4.3, we show that less modifications already suffice to achieve the originally intended security goals.

Yu et al. [37] very recently suggested multiple possible improvements to the Otway-Rees protocol based on the results of their analysis with *Enhanced Authentication Tests* (EAT). Guttman and Thayer [20] first proposed *Authentication Tests* (AT) as a symbolic method for the analysis of cryptographic protocols regarding their authentication properties. The method was later improved by Yang and Luo [36] to also reason about confidentiality aspects of protocols, which led to EAT. The proposed improvements have the advantage over those of Chen in [12] that they achieve the same strong security goals with only minimal alternations to the original messages of the protocol and only one additional part encrypted with k_{ab} in the fourth message. Nonetheless, we will propose our own improved version of the protocol in Section 4.3 that does not even require the additional encrypted component to achieve key secrecy of the communication key k_{ab} and mutual authentication between the principals A and B.

In contrast to the previously discussed analyses stands that of Backes [4], who used the abstract cryptographic library [5] presented in Section 3.2 to receive "A Cryptographically Sound Dolev-Yao Style Security Proof of the Otway-Rees Protocol". The approach of Backes includes a symbolic security proof of the Otway-Rees protocol in the abstract library and exploits the fact that the abstract library can be safely realized using real cryptography – proven in [5] – to further derive its security in a computational setting. The proof assumes that the protocol is implemented such that commonly known type-flaw and implementation dependent attacks like those pointed out in [14] or [8] are not possible. Backes defines the security of the Otway-Rees protocol in the ideal (abstract) library in terms of a secrecy and consistency property. The secrecy property states that the adversary cannot learn a key shared by two honest users after successfully running the protocol; the consistency property refers to the consistency of users' views regarding the peer of their respective key session. Secrecy of the communication key k_{ab} is proven based on the fact that the key is only exchanged over channels secured by the long-term keys k_{as} and k_{bs} shared between A and the AS respectively B and the AS; the proof for consistency is directly derived from the secrecy proof. Backes then carries the properties over to the cryptographic setting by using the proof work in [5] and showing that the so-called *commitment problem* does not occur in the Otway-Rees protocol, i.e., that the long-term keys used to encrypt the communication key are never leaked. This is derived from the assumption that the long-term keys of honest principals are secret. Thus, Backes is able to show that the secrecy and consistency properties also hold for an implementation of the library that uses real cryptography.

3.3.2 Yahalom

The Yahalom protocol is an authentication protocol that, similarly to the Otway-Rees protocol, performs a key exchange using symmetric long-term keys shared with an authentication server. The protocol originates from personal communication and was formalized in [11]:

1. $A \rightarrow B: \quad A, N_a$
2. $B \rightarrow AS: \quad B, \{A, N_a, N_b\}^s_{k_{bs}}$
3. $AS \rightarrow A: \quad \{B, k_{ab}, N_a, N_b\}^s_{k_{as}}, \{A, k_{ab}\}^s_{k_{bs}}$
4. $A \rightarrow B: \quad \{A, k_{ab}\}^s_{k_{bs}}, \{N_b\}^s_{k_{ab}}$

The long-term keys that A and B share with the AS are denoted – similar as in Otway-Rees – k_{as} and k_{bs}. The first message is emitted by the initiator A who chooses a random nonce N_a and sends its identity together with the generated nonce to B. On receipt of the first message, the responder B chooses its own nonce N_b and encrypts a message containing A's identity and the two nonces under k_{bs}. B then sends the second message containing its own identity and the encrypted part to the AS. Using B's identity, the AS is able to retrieve k_{bs} and decrypt the ciphertext in the second message. Among other things, it obtains A's identity, which it needs to retrieve the other long-term key corresponding to A. The AS goes on to choose a fresh session key k_{ab} for the communication of A and B and then sends the third message containing two parts encrypted under the users' long-term keys to A. The first part is encrypted under k_{as} and intended for A; it contains B's identity, the session key k_{ab}, and the nonces N_a and N_b. Likewise, the second part is encrypted under k_{bs} and intended for B; it only contains A's identity and k_{ab}. When the initiator receives the third message, it decrypts the part encrypted under k_{as} and verifies the responder's identity as well as its own challenge N_a. If the verification succeeds, it accepts k_{ab} as session key and uses it to encrypt N_b for verification by the responder. A then forwards the second encrypted part from the third message together with B's encrypted nonce to B. In the last step, B receives the fourth message, decrypts the part from the AS using k_{bs}, verifies A's identity, and, if successful, uses k_{ab} to decrypt A's part and verify its challenge N_b. Finally, if the challenge is correct, it also accepts k_{ab} as session key.

Analyses and Results

As mentioned, the Yahalom protocol was first formalized in the BAN paper [11] and was therefore naturally subject to a BAN analysis. The initial assumptions made for Yahalom are quite similar to those in the analysis of Otway-Rees – the principals believe in the appropriateness of their long-term keys and the users believe in the freshness of their nonces and trust the authentication server to choose an appropriate session key. However, in case of Yahalom, several additional assumptions were needed due to the unusual protocol design. For example, B requires a statement about the freshness of k_{ab} to infer that the last message has been sent timely and A must trust the AS to pass on B's challenge in the third message. From the initial assumptions and the third message Burrows et al. easily derive that A trusts in the suitability of k_{ab} as communication key. To derive the same belief for B, a small adaption to the logic was necessary allowing B to use the session key k_{ab} for the decryption of N_b before it can assume that the key is good by verifying N_b. This adaption and the fact that B trusts AS and A to generate and respectively use a fresh session key k_{ab} suffice to infer that B also beliefs in the suitability of the session key. Further, B is convinced A has the same beliefs about k_{ab} as itself and therefore B would not notice if A decides to replay an old key in the fourth message. According to the authors this does however not impose a major flaw in the protocol because principals are generally assumed to be honest in the context of a BAN analysis. Based on the analysis results, the BAN authors have proposed a modified version of Yahalom that should lead to the same outcome with less encryption and fewer assumptions. The modified version was later proven insecure by Syverson [33] who documented an attack in which the attacker is able to make A accept a public nonce chosen by B as session key.

The results of the BAN analysis were later taken up by Chen and Shi [13] in an analysis of the Yahalom protocol with the SVO logic [34]. Chen and Shi interpret the fact highlighted by Burrows et al. [11] that a malicious A could replay an old session key in the fourth message such that the protocol cannot achieve the authentication goals it intends to achieve. To proof their point, they first formalize the security goals of Yahalom in terms of the SVO logic and then show that the goals are not achieved by the original protocol. The initial assumptions overlap with those of the BAN analysis in assumptions about A's and B's trust in the long-term keys they share with AS, the secrecy and suitability of the session key generated by AS, and the freshness of their respective nonces, as well as B's trust in a statement about the freshness of the session key from AS. In addition to the BAN assumptions, Chen and Shi assume that A trusts AS equally regarding the freshness of k_{ab}. Further assumptions that were required in the BAN analysis to derive B's belief in the freshness and thus the secrecy of the session key are not included in the SVO analysis because A may act maliciously. The goals defined by Chen and Shi are stronger than the ones raised by the BAN authors. Naturally, the users A and B shall both be convinced that the session key is secret and a good communication key. However, they also shall be satisfied in their belief that the session key is fresh and that the respective other party is equally convinced about its suitability and freshness. The analysis shows that A is not convinced that B shares its beliefs about the key and B itself is not convinced that the key is fresh. This leads Chen and Shi to propose the following improved protocol:

1. $A \rightarrow B: \quad A, N_a$
2. $B \rightarrow AS: \quad B, \{A, N_a, N_b\}^s_{k_{bs}}$
3. $AS \rightarrow A: \quad \{B, k_{ab}, N_a, N_b\}^s_{k_{as}}, \{A, N_b, k_{ab}\}^s_{k_{bs}}$
4. $A \rightarrow B: \quad \{A, N_b, k_{ab}\}^s_{k_{bs}}, \{A, N_b\}^s_{k_{ab}}$
5. $B \rightarrow A: \quad \{B, N_a - 1\}^s_{k_{ab}}$

They show that their improvements suffice to derive the remaining security goals and hence complete the SVO derivations of A and B. The improved Yahalom protocol's structure is somewhat similar to that of the improved Otway-Rees protocol proposed by Chen [12] and discussed in Section 3.3.1; it therefore also achieves the same strong security goals. Although the improved Yahalom protocol only slightly deviates from the original protocol in the third and fourth message, it still requires the additional fifth message to achieve its security goals.

Besides symbolic analyses based on belief logics, the Yahalom protocol further was subject to an analysis with the CSP approach [31]. Ryan et al. used the Yahalom protocol as running example to demonstrate how to formalize and proof security properties using the CSP approach. We have already explained in Section 3.2 how CSP supports the notation of trace events, which mark certain points passed during a protocol run and allow for the deduction of assumptions about the state of the particular principal triggering them. Ryan et al. denote the secrecy property in CSP as the requirement that if a principal claims that a message is secret and sender as well as receiver of the message are honest, then this message should never leak to the adversary. This must particularly apply to the session key contained in the third message from the AS to A and claimed to be only known to A and B. Mutual authentication is formalized in two separate properties for the initiator and respectively, the responder. Authentication of the initiator to the responder requires that an honest initiator A triggers a running signal before B commits to the run and both should agree on their identities and nonces as well as the exchanged key. Responder authentication is a little weaker in that Ryan et al. only require an honest B to emit a running signal before A completes the run and that both parties should agree on their identities as well as nonces, but not necessarily on the session key. This simply comes from the fact that A gets no reassurance whether B really receives or accepts the key k_{ab} within the scope of the protocol (because B never encrypts a message for A using k_{ab}). Finally, Ryan et al. use CSP to show that the specified security properties indeed hold for the Yahalom protocol, i.e., the session key k_{ab} remains secret as long as principals do not act in a malicious way, and initiator and responder achieve timely mutual authentication.

Another effort to analyze the Yahalom protocol was made by Paulson [28], who used an inductive method based on possible protocol traces over time. Paulson's approach also includes the model of an active adversary that controls the communication and can occasionally compromise session keys. Their work aims to confirm the results of the BAN analysis [11] regarding both, the original Yahalom protocol, as well as the modifications proposed by the BAN authors. The secrecy property for the Yahalom protocol is formalized and proven in terms of multiple theorems and lemmas. The session key compromise theorem states that the loss of one session key does not affect the secrecy of other session keys. The session key secrecy theorem ensures that if the server issues a key to A and B, only A and B will obtain the key. Additionally, the third message from which the users obtain the session key and deduce timeliness guarantees must really originate from the AS. This already suffices to prove that the session key, which A accepts, is unknown to the adversary. To conclude the same for B, Paulson shows that, under certain condiditions, N_b remains secret and therefore B is able to deduce that the session key is fresh. Mutual authentication is directly derived from the secrecy of the long-term keys and the session key. Paulson argues that if A receives its certificate from the AS in the third message and B is honest, then B must have sent the second message in which it encrypted N_a using k_{bs}. B on the other hand gets its guarantees from the fourth message containing N_b encrypted under the session key k_{ab}, which B believes to be only known to A (and the AS, which it trusts not to act maliciously), and therefore, the encryption must have originated from A. Our analysis in Section 4.4 confirms the results of Paulson's analysis and formalizes the security properties of the Yahalom protocol in a corresponding DY* model.

In contrast to security proofs such as those of Ryan et al. [31] or Paulson [28] are attacks such as the replay attack described in [11] and mentioned in [13]. Another attack on the Yahalom protocol was documented by Clark and Jacobs in "A Survey of Authentication Protocol Literature" [14]:

1. $I(A) \rightarrow B: \quad A, N_a$
2. $B \rightarrow I(AS): \quad B, \{A, N_a, N_b\}^s_{k_{bs}}$
3. omitted in the attack, but certainly relevant for the adversary to get a handle to the key
4. $I(A) \rightarrow B: \quad \{A, N_a, N_b\}^s_{k_{bs}}, \{N_b\}^s_{N_a,N_b}$

Here, the adversary plays the role of A and the AS to establish a session key with B that has not been issued by the AS. However, it is clear to see that in order for the adversary to benefit from this attack, it must somehow learn N_b, which it can only do if it gets a handle to A's long-term key k_{as}. As the BAN authors already clarified in their analysis, the Yahalom protocol is not designed to protect against such attacks based on compromised long-term secrets like the attack outlined here or the replay attack mentioned in the BAN paper. Therefore, these attacks are not in conflict with the DY* security proof of the Yahalom protocol we give in Section 4.4.

3.3.3 Denning-Sacco

The Denning-Sacco protocol with public keys is an authentication and key exchange protocol based on public key encryption and signatures. Public keys are known to an authentication server or certification authority that helps distributing them. The protocol was outlined by Denning and Sacco in "Timestamps in Key Distribution Protocols" [16] and has the following structure:

1. $A \rightarrow AS: \quad A, B$
2. $AS \rightarrow A: \quad C_A, C_B$
3. $A \rightarrow B: \quad C_A, C_B, \{\{k_c, T\}^a_{S_A}\}^a_{P_B}$

C_A and C_B are certificates issued by the AS for public key distribution, where $C_A = \{A, P_A, T\}^a_{S_{AS}}$ and $C_B = \{B, P_B, T\}^a_{S_{AS}}$. $P\_$ denotes public keys for usage in public key encryption or signature schemes, $S\_$ denotes the corresponding private keys for decryption of ciphertexts or signature verification, k_c is the communication key established as a result of the protocol, and T denotes a timestamp. Denning and Sacco proposed the use of timestamps to ensure timeliness of messages and certificates and to eliminate the need of handshake messages between users. They suggested that principals should synchronize their local clocks by setting them according to some standard source. A timestamp T is valid as long as $|Clock - T| < \Delta t1 + \Delta t2$ holds true, where $\Delta t1$ is a tolerance value to account for discrepancies between users' local clocks (more precisely, between possibly different standard sources) and $\Delta t2$ is the time needed to send a message over the network, i.e., the network delay time. Therefore, the timestamps can protect against replays of old messages, especially of the third message containing the communication key. More precisely, if a communication key between A and B somehow leaks, the attacker cannot use it to establish communication with B under the name of A at a later time, particularly any time T' such that $T' >= T + \Delta t1 + \Delta t2$. Moreover, the timeliness guarantees we get from the timestamps make a handshake between the users obsolete.

The protocol begins with A telling the AS its own identity and the identity of the other user it wants to talk to, here denoted as B, in the first message. The AS then retrieves the users public keys based on the identities provided in the first message and issues the certificates C_A for A and C_B for B, with T set to the AS's current local clock. The certificates are included in the second message to A who first validates its own certificate by checking whether identity and public key are correct, and whether T is valid and the certificate therefore current. If C_A is valid and has been issued timely, A goes on by verifying the identity of B and the timestamp T in C_B, before it retrieves B's public key P_B. A then chooses a fresh communication key k_c and signs it with its private key S_A, together with the timestamp T from the certificates C_A and C_B. Finally, it encrypts the certificate for the communication key under P_B such that only B can obtain k_c, and forwards it together with the server certificates C_A and C_B to B. B receives the last message and, similarly to A, validates its own certificate and retrieves A's public key from C_A. Afterwards, B decrypts A's certificate containing k_c with S_B and then verifies it with P_A. To ensure that k_c is not a replay, B moreover checks T contained in the certificate issued by A, before it finally accepts k_c.

Ambiguities regarding the Message Structure

The paper of Denning and Sacco is not completely clear when it comes to the origin of the timestamp in the communication key certificate issued by A and sent to B in message (3). Recognized literature commonly interprets the protocol such that the timestamp in the certificate for the communication key originates from A instead of the AS (e.g., [1], [2], [8], [22]). This interpretation would leave the third message with the following structure:

3. $C_A, C_B, \{\{k_c, T_A\}^a_{S_A}\}^a_{P_B}$

T_A denotes a new timestamp with the value of A's local clock at the time the third message is created. This interpretation allows for an attack in which B can masquerade as A and reuse A's certificate to establish communication with a third user in the name of A. We will discuss this attack in more detail in the next section.

It was a little bit surprising that none of the existing literature about the Denning-Sacco protocol with public keys interpreted the protocol such that A reuses the timestamp from the server certificates. From our point of view, this is the more logical interpretation since all timestamps are denoted with the same identifier T. It also makes the protocol more efficient, since B only needs to check that the timestamps are the same and then ensure its validity for one of those timestamps. One could argue that Denning and Sacco do not indicate in the scope of their paper that checks on the relation of timestamps in different certificates are performed. We claim that it is reasonable to assume that A compares the timestamps in the server certificates before it reuses one of them in the certificate for the communication key. Consequently, if A performs such a check, so does B. We therefore argue that our interpretation of the protocol and the behavior of principals is reasonable in itself. Nevertheless, we also wanted to make sure that our research on the Denning-Sacco protocol is sound with respect to the protocols originally intended structure. From personal communication with Giovanni Maria Sacco, one of the protocol's authors, we got the confirmation that our interpretation of the protocol is valid and the timestamp in A's certificate is actually the same timestamp that the server used in the public key certificates.

Documented Attacks

Abadi and Needham [1] were the first to find an attack on the Denning-Sacco protocol with public keys. Their attack bases on the interpretation of the protocol where A uses the value of its current local clock for the timestamp in the communication key certificate. It was later mentioned again by Anderson and Needham in [2] and also recognized in general authentication protocol literature such as [8]. Abadi and Needham regard the public key certificates provided by the AS as black boxes so that no checks are performed regarding the relation of possible timestamps they contain. The attack they describe can thus be carried out by a malicious B in two parallel sessions as follows:

Session 1:

1. (3) $A \rightarrow I(B):\quad C_A, C_B, \{\{k_c, T_A\}^a_{S_A}\}^a_{P_B}$

Session 2:

2. (1) $I(B) \rightarrow AS:\quad A, C$
3. (2) $AS \rightarrow I(B):\quad C_A, C_C$
4. (3) $I(B) \rightarrow C:\quad C_A, C_C, \{\{k_c, T_A\}^a_{S_A}\}^a_{P_C}$

In Session 1, A establishes communication with B. B continues in a new session by requesting new certificates for the public keys of A and a third user C from the AS. The AS responds to B by issuing the requested certificates. As stated, the attack abstracts from the actual contents of the certificates, so we may just assume that they suffice to distribute public keys and to provide reassurance of their correctness. Hence, B obtains C's public key P_C from C_C, re-encrypts the certificate for k_c, which was originally issued by A in the third message of Session 1, with P_C, and finally forwards the public key certificate and the re-encrypted communication key certificate to C. When C receives this message from B, it successfully verifies C_A and C_C, obtains A's public key P_A, decrypts the last part using P_C, and finally uses P_A to verify the signature of A containing k_c and T_A, which consequently must originate from A. If T_A is valid, C accepts k_c assuming it has established a secure channel with A, when in fact it is talking to the attacker.

Although we clarified in Section 3.3.3 that the attack does not fully reflect the properties of the original protocol as specified by Denning and Sacco in [16], it is still interesting because it highlights the subtlety required in the specification of security properties regarding assumptions about the protocol's structure and also its implementation. Moreover, it once again shows the importance of a core principle for the design of security protocols – robustness [7] –, that messages should include all the information to unambiguously authenticate them. This particularly holds true for the inclusion of key peer identities where they are not clear, for example, the inclusion of B's identity in A's signature in the third message of the Denning-Sacco protocol containing the communication key k_c, which is generated by A for secret communication with B.

Security Proofs

There are no formal security proofs of the Denning-Sacco protocol in existing literature that we know of. Generally, there seems to be a lack of effort regarding the symbolic analysis of the Denning-Sacco protocol. This lack may come from the fact that the protocol includes timestamps, which are hard to model, and for which it is hard to express security properties in a symbolic setting because this presupposes the existence of a global notion of time that cannot be controlled by the attacker, but does not restrict it either. One example for a symbolic prover that has been used to analyze the Denning-Sacco protocol is the AVISS tool [3]. In their work, the authors state that they could find one man-in-the-middle attack on the protocol, which was previously not documented. However, they neither explain how the attack can be carried out nor what assumptions have been made and what the exact outcome of the attack is. We counteract the shortage of sufficiently documented formal analyses of the Denning-Sacco protocol with a detailed symbolic security proof for the protocol with DY* in Section 4.5 of this thesis.

4 Analysis of the Selected Protocols

In the course of this thesis, we performed an elaborate security analysis of three cryptographic protocols concerned with authentication and key exchange with the DY* symbolic prover. The investigated protocols are the Otway-Rees protocol [27], the Yahalom protocol [11], and the Denning-Sacco protocol with public keys [16]. This chapter documents the procedure and results of our analysis in full detail and further puts our results in a context with results of other approaches – summarized in Section 3.3 – where they can be interpreted.

4.1 General Procedure

Each analysis follows roughly the same procedure. We start by creating a precise outline of the protocol in question based on its formal description taken from the paper in which the protocol was first presented or formalized. Our main goal in this phase of the analysis is to answer questions about the following protocol properties or goals that are bound to those properties:

- principal roles, e.g., users, authentication servers, initiator, responder, etc.;
- initial knowledge of principals, e.g., other principals' identities, or previously established long-term or public keys;
- messages, i.e., origin, destination and structure of messages;
- generated nonces, i.e., origin, purpose and intended audience of generated nonces;
- exchanged keys or secrets, i.e., keys or secrets that are established as a result of a protocol run and their intended audience; and
- authentication goals.

The question of message origin, destination, and structure has already been answered in Section 3.3.

Given the formal outline of the protocol, we continue by developing a symbolic model of the protocol in the labeled security layer of DY*. To ensure the correctness of our model, we verify it with F* and check whether its execution results in a reasonable trace. We also substantiate the coherence of our model with the actual protocol as depicted in Section 3.3 and with the properties elaborated in our outline by explaining the key parts of the model in greater detail. In case of ambiguities in the protocol description, we furthermore justify the validity of our interpretation and the resulting parts of the model. Having established correctness and validity, we derive security properties that can be expressed in DY* from the protocol goals defined in our formal outline of the protocol, complement them with our own suggestions, and finally formulate them in terms of our DY* model. Since the security properties are bound to the model, they are verified together with the model using F*. If the specified security properties cannot be proven, we explore resulting attack vectors and augment our DY* model based on found attacks. Additionally, we propose minimal changes sufficient to counter the deficiencies of the protocol and then apply these fixes to the DY* model with the ultimate goal to prove the remaining security properties in the enhanced model, and hence complete the security proof of the protocol.

In the last part of the analysis, we compare our results to the results of other proof efforts or analyses, i.e., those outlined in Section 3.3. That is, we look at other security proofs and the concrete security properties proven by them as well as documented attacks, and put them into a context with the security properties and attacks we were able to model and verify in DY*. We then interpret differences and similarities in the results of our approach and other approaches under consideration of the general proof setting, the attacker model (i.e., active or passive) and attacker capabilities, as well as assumptions and abstractions made in this and in other approaches.

4.2 DY* Models and Source Code Repository

The complete source code of the DY* models of the authentication protocols analyzed as part of this master's thesis has been published in a GitHub repository [21]. In total, the repository contains five DY* models, where each model represents one of the three investigated protocols or variations of these protocols, possibly based on different assumptions. Each DY* model divides into a set of modules that model different parts of the protocol, e.g. messages, protocol states, the individual protocol steps performed by honest principals, or possible attacks carried out by the DY* adversary. In particular, the repository comprises the following five DY* models:

- *otway-rees*: contains a model of the original Otway-Rees protocol
- *otway-rees-fixed*: contains a model of an improved version of the Otway-Rees protocol that prevents known attacks on the protocol and renders it secure with respect to the goals of the original version (see Section 4.3.1)
- *yahalom*: contains a model of the Yahalom protocol, including a security proof based on the properties and goals stated in Section 4.4.1
- *denning-sacco*: contains a model of the Denning-Sacco protocol and a DY* extension to capture protocol properties dependent on timestamps, resulting in an incomplete security proof regarding the properties and goals stated in Section 4.5.1
- *denning-sacco-central-auth*: contains the Denning-Sacco model, but defines the authentication server as a global singleton to account for some shortcomings of DY* that render a general security proof of the protocol in DY* as yet impossible

To illustrate our work and the results of our analyses, we will use selected code examples from the DY* models listed above.

4.3 Otway-Rees

This section comprises the analysis of the Otway-Rees protocol. We show that the protocol as originally proposed and described by Otway and Rees [27] does not achieve intended secrecy and authentication goals. To this end, we demonstrate multiple attacks on the protocol that break its security. We then provide fixes that prevent all these attacks and prove that the improved protocol is able to achieve all security goals. The message structure of the protocol has been outlined in Section 3.3.1.

4.3.1 Properties and Goals

Prior to the analysis, we capture the protocol's properties and goals that are essential for the development of a coherent model in DY*.

Principal Roles

A run of the Otway-Rees protocol involves three principals: two users A and B that want to communicate to each other and a trusted authentication server AS that helps in establishing a secure communication channel for them. One user – A – takes the role of the initiator and the other – B – that of the responder. The initiator communicates only to the responder, while the responder involves the authentication server for the key generation and key exchange before responding to the initiator.

Initial Knowledge

The initiator initially knows the identities of the responder and of the trusted authentication server. The responder is not necessarily aware of the initiator's existence, but also has a handle to the identity of the server. The server itself must know both principals in order to establish a channel between them. Each of the users shares a symmetric long-term key with the server, which forms a secure channel over which a key can be exchanged. The long-term key of initiator and server is denoted by k_{as}; that of responder and server by k_{bs}.

Generated Nonces

During a protocol run, the initiator generates two nonces N_{cid} and N_a. N_{cid} is a public conversation identifier to identify the protocol run, while N_a is a random challenge to assure the initiator of the timeliness of the last message from the responder. Similarly to the initiator, the responder generates a nonce N_b to get assurance of the timeliness of the third message from the server. Since both N_a and N_b appear in the protocol messages only in encrypted form under k_{as} and k_{bs}, only the server should have access to them besides the respective user who generated them.

Exchanged Keys or Secrets

The server is in charge of key generation and distribution. Upon receiving the second message from the responder, it generates a session key k_{ab} and distributes it to the initiator and responder via the already installed channels secured by the long-term keys. The session key is supposed to remain secret to the two users and the authentication server. In particular, this means that the session key shall not leak to the attacker unless it is able to corrupt a principal with a handle to the key. Moreover, the session key accepted by the users shall in fact be the key that was previously generated in the protocol run by the server.

Authentication Goals

Authentication goals of the protocol involve mutual and timely authentication between two users A and B. Precisely, if A and B accept a session key k_{ab}, then there must be a corresponding protocol run in which A has taken the role of the initiator and B that of the responder. After the successful protocol run, both A and B should be convinced that the other's identity is correct and that the other was actually involved in the run. The authentication goals build on the same premise as the secrecy goals, that the principals involved in the protocol run are honest.

4.3.2 Modeling the Protocol in DY*

Now that we have a clear idea of the properties and goals of the Otway-Rees protocol, we have all the information we need to develop our DY* model. We begin by modeling the protocol's message structure and its stateful parts consisting of the protocol specific state sessions. Next, we define characterizing events occurring during a regular run of the protocol and implement possible protocol specific usage and trace predicates. Once we have modeled the protocol base, we define and implement the individual protocol steps of honest principals and finally add execution capabilities to our model.

Messages

```
/// Format of encrypted message parts
noeq type encval =
    | EncMsg1: n_a:bytes -> cid:bytes -> a:string -> b:string -> encval
    | EncMsg2: n_b:bytes -> cid:bytes -> a:string -> b:string -> encval
    | EncMsg3_I: n_a:bytes -> k_ab:bytes -> encval
    | EncMsg3_R: n_b:bytes -> k_ab:bytes -> encval

(* Serialized and encrypted encvals with tags *)
let ser_encval i l = (tag:string & msg i l)
let enc_encval i = (tag:string & msg i public)

noeq type message (i:nat) =
    | Msg1: cid:bytes -> a:string -> b:string -> ev_a:enc_encval i -> message i
    | Msg2: cid:bytes -> a:string -> b:string -> ev_a:enc_encval i -> ev_b:enc_encval i ->
message i
```

```
        | Msg3: cid:bytes -> ev_a:enc_encval i -> ev_b:enc_encval i -> message i
        | Msg4: cid:bytes -> ev_a:enc_encval i -> message i
```

We modeled the message structure of the Otway-Rees protocol in a module `OYRS.Messages` via the two types `encval` and `message (i:nat)` depicted above. `encval` captures parts of messages that are protected by encryption. Each of the four constructors of `encval` corresponds to a specific chunk of data that appears in the messages in encrypted form. For example, `EncMsg1 n_a cid a b` corresponds to A's encryption $\{N_a, N_{cid}, A, B\}^s_{k_{as}}$ in the first message. Because the messages of the Otway-Rees protocol also contain data that is not protected by encryption, we need a second type `message i` for the actual messages. The constructors of this type let us join unencrypted and encrypted message parts into a single compound value representing messages that can be serialized and then safely sent over the network. To allow for safe transmission of messages over the network, terms used in unencrypted parts and the terms representing the encrypted parts of messages must be publishable. The publishability of ciphertexts, i.e., the encrypted parts, is guaranteed by the encryption function in the labeled layer. Hence, encryption is done before message construction, i.e., the data that is to be encrypted, which is carried in values of type `encval`, is first serialized itself, then encrypted, and afterwards passed to the respective constructor of `message i`. For simplicity, we refer to values of type `encval`, i.e., values containing data to be protected by encryption, from now on as *encryption values*. Two additional types `ser_encval i l` and `enc_encval i` are introduced as wrappers for the results of the serialization and encryption processes of encryption values. They respectively annotate the value `msg i l` resulting from serialization and the ciphertext `msg i public` resulting from encryption with a tag identifying the constructor of `encval` that is to be used by the parse function to reconstruct the original value from the serialized value or ciphertext. The type `msg i l` is a refinement of `bytes` and denotes values valid at trace index `i` with respect to the protocol specific usage predicate and labeled with some label that can flow to `l`. To construct the first message, we would use the `Msg1 cid a b ev_a` constructor, where the first three parameters correspond to the unencrypted part (that is, N_{cid}, A, B) containing the conversation identifier and the identities of the two users, while the last parameter `ev_a` is the encrypted form of `EncMsg1 n_a cid a b` under k_{as}.

```
    val serialize_encval: i:nat -> ev:encval -> l:label{valid_encval i ev l} -> sev:(
ser_encval i l)
        {
            let (|tag,_|) = sev in
            match (tag, ev) with
            | ("ev1", EncMsg1 _ _ _ _)
            | ("ev2", EncMsg2 _ _ _ _)
            | ("ev3_i", EncMsg3_I _ _)
            | ("ev3_r", EncMsg3_R _ _) -> True
            | _ -> False
        }
    val parse_encval: #i:nat -> #l:label -> sev:(ser_encval i l) -> r:(result encval)
        {
            match r with
            | Success ev -> valid_encval i ev l
            | Error _ -> True
        }

    val serialize_msg: i:nat -> m:(message i){valid_message i m} -> msg i public
    val parse_msg: #i:nat -> sm:(msg i public) -> r:(result (message i))
        {
            match r with
            | Success m -> valid_message i m
            | Error _ -> True
        }
```

The serialization and parsing of messages is implemented with functions `serialize_msg` and `parse_msg`. Encryption values are serialized and parsed separately – respectively, before encryption or after decryption – using the functions `serialize_encval` and `parse_encval`. `valid_msg i m` and `valid_encval i ev l` ensure the validity of the data in messages `m` and encryption values `ev` at trace index `i`. `valid_encval` additionally requires that the data is labeled with labels readable by the readers of `l`. That is, because the data in encryption values must be readable by anyone that has access to the key used to encrypt them. The serialization functions only accept values valid with respect to the `valid_msg` and `valid_encval` functions and, likewise, the parse functions only output values that are valid. The tag of the result of type `ser_encval i l` of `serialize_encval` must match the constructor used to construct the serialized encryption value `ev`. For messages, the tag is part of the serialized value itself and is thus handled implicitly by the serialization API. We will explain later, why we handle tags of encryption values explicitly in the model of the Otway-Rees protocol. Unlike serialization, parsing may fail if serialized values are tagged incorrectly, which is why the output of `parse_msg` and `parse_encval` is wrapped in a `result`.

State

```
    noeq type session_st =
        (* Auth server session for secret keys shared with principals *)
        | AuthServerSession: p:principal -> k_ps:bytes -> us:string -> session_st
        (* Initial knowledge of principals *)
        | InitiatorInit: srv:principal -> k_as:bytes -> b:principal -> session_st
        | ResponderInit: srv:principal -> k_bs:bytes -> session_st
        (* Protocol states *)
        | InitiatorSentMsg1: srv:principal -> k_as:bytes -> b:principal -> cid:bytes -> n_a:
bytes -> session_st
        | ResponderSentMsg2: srv:principal -> k_bs:bytes -> a:principal -> cid:bytes -> n_b:
bytes -> session_st
        | AuthServerSentMsg3: a:principal -> b:principal -> cid:bytes -> n_a:bytes -> n_b:
bytes -> k_ab:bytes -> session_st
        | ResponderSentMsg4: srv:principal -> a:principal -> k_ab:bytes -> session_st
        | InitiatorRecvedMsg4: srv:principal -> b:principal -> k_ab:bytes -> session_st
```

The stateful parts of the Otway-Rees protocol are captured by the `OYRS.Sessions` module. The central component of this module is the type `session_st` with constructors for each set of data that needs to be stored by some principal throughout execution of the protocol. The first part of the constructor name refers to the principal role for which the constructor is intended and the second part describes the context or type of the stored data. For example, at the end of the first protocol step, after sending the first message to the responder, the initiator creates and stores a state session `InitiatorSentMsg1 srv k_as b cid n_a`. In this state session, the initiator stores the identity of the authentication server `srv` and the long-term key `k_as` it shares with `srv`, the identity of the responder `b`, and the conversation identifier `cid` as well as the nonce `n_a`, which were generated by the initiator during the first protocol step. That way, the initiator can access the stored data at a later time, i.e., in the last protocol step, where it must verify whether the server has encrypted the correct challenge n_a and whether the responder has answered with the correct conversation identifier cid. We also model state sessions for storing the long-term keys that initiator, responder, and authentication server must have available at the beginning of protocol execution in the protocol specific state session type defined in the OYRS.Sessions module, since DY* does not yet have a generic API to install symmetric long-term keys. The authentication server can maintain key sessions with multiple users. In a server session, the server stores the identity of the user p with whom he shares a key, the shared long-term key k_ps itself, and the usage context us of the key. To construct a new server session, the server uses the constructor AuthServerSession p k_ps us. The initiator and responder store the long-term keys on their end, respectively, using the InitiatorInit srv k_as b and ResponderInit srv k_bs constructors. In addition to the server's identity srv and the long-term key k_as, the initiator also stores the identity of the responder b, which he must know to address the first message to it. Hence, the server sessions and the initial states of initiator and responder model the initial knowledge of the principals before protocol execution.

```
    let valid_session (i:nat) (p:principal) (si vi:nat) (st:session_st) =
        match st with
        | AuthServerSession peer k_peer_srv us ->
          is_aead_key i k_peer_srv (readers [P peer; P p]) us
        | InitiatorInit srv k_as b ->
          is_aead_key i k_as (readers [P p; P srv]) "sk_i_srv"
        | ResponderInit srv k_bs ->
          is_aead_key i k_bs (readers [P p; P srv]) "sk_r_srv"
        | InitiatorSentMsg1 srv k_as b cid n_a ->
          is_aead_key i k_as (readers [P p; P srv]) "sk_i_srv" /\
          is_labeled i cid public /\
          is_labeled i n_a (readers [P p; P srv])
        | ResponderSentMsg2 srv k_bs a cid n_b ->
          is_aead_key i k_bs (readers [P p; P srv]) "sk_r_srv" /\
          is_msg i cid public /\
          is_labeled i n_b (readers [P p; P srv])
        | AuthServerSentMsg3 a b cid n_a n_b k_ab ->
          is_msg i cid public /\
          is_msg i n_a (readers [P p]) /\
          is_msg i n_b (readers [P p]) /\
          is_labeled i k_ab (readers [P p; P a; P b])
        | ResponderSentMsg4 srv a k_ab -> is_msg i k_ab (readers [P p])
        | InitiatorRecvedMsg4 srv b k_ab -> is_msg i k_ab (readers [P p])

    val serialize_session_st: i:nat -> p:principal -> si:nat -> vi:nat -> st:session_st{
valid_session i p si vi st} -> msg i (readers [V p si vi])
    val parse_session_st: sst:bytes -> result session_st
```

Serialization and parsing is handled similarly as with messages by functions serialize_session_st and parse_session_st. The valid_session predicate accepts additional parameters to make statements about the labels of the terms stored in the respective state sessions, and in this context, to account for the fine-grained labeling in DY* including principals p, specific state sessions si, or even versions vi. For example, a server session AuthServerSession peer k_peer_srv us is valid if the stored key k_peer_srv is an AEAD key with label readers [P peer; P p] and usage string us, where peer is the stored user with whom the server shares the key. That is, if the key was specifically generated for AEAD and only peer and p have access to it. The server is p in this case; the principal in whose state the given state session is stored. For stored terms that are not publishable by nature like strings or natural numbers, we must at least require that they are valid `bytes` at trace index `i` (implicitly ensured by the aliases for `is_labeled`, `is_msg`, and `is_aead_key` defined in the model) and that their labels flow to `readers [V p si vi]`. Thus, we ensure that a principal `p` accessing version `vi` of a state session `si` storing some arbitrary term, can also read the particular term. The result of the serialization is of type `msg i (readers [V p si vi])`, which ensures that the principal can read the serialized state session itself. For `parse_session` we do not explicitly require that the resulting state session is valid, since this is implicitly ensured by the `valid_trace` predicate of the labeling layer. Parsing serialized state sessions may fail for the same reason as parsing serialized messages. Therefore, the result of `parse_session_st` is also wrapped in a `result`.

Events

```
let event_initiate (cid:bytes) (a b:principal) (n_a:bytes) : event =
    ("initiate",[cid;(string_to_bytes a);(string_to_bytes b);n_a])
let event_request_key (cid:bytes) (a b:principal) (n_b:bytes) : event =
    ("req_key",[cid;(string_to_bytes a);(string_to_bytes b);n_b])
let event_send_key (cid:bytes) (a b:principal) (n_a n_b k_ab:bytes) : event =
    ("send_key",[cid;(string_to_bytes a);(string_to_bytes b);n_a;n_b;k_ab])
let event_forward_key (cid:bytes) (a b:principal) (k_ab:bytes) : event =
    ("fwd_key",[cid;(string_to_bytes a);(string_to_bytes b);k_ab])
let event_recv_key (cid:bytes) (a b:principal) (k_ab:bytes) : event =
    ("recv_key",[cid;(string_to_bytes a);(string_to_bytes b);k_ab])
```

Five events are defined for the Otway-Rees protocol, which align with the five steps of protocol execution that we define later. The *initiate* event (`event_initiate cid a b n_a`) is triggered by the initiator `a` to signal that it has initiated a protocol run identified by `cid` with the responder `b` and that it has generated a challenge `n_a` in association with the run. Similarly, the other events mark other important milestones throughout protocol execution. The responder triggers the *request key* event when it requests the session key from the server. When the server distributes the key to the principals, it triggers the *send key* event. The responder receives the key first and triggers the *forward key* event when it forwards the server message containing the key to the initiator. Finally, the initiator receives the key and triggers the *receive key* event. The event definitions are part of the `OYRS.Messages` module.

Protocol Specific Usage and Trace Predicates

We do not have to implement a protocol specific predicate for AEAD that further specifies the properties of keys used for encryption and of the data to encrypt. Similarly, we also do not require an event predicate, which specifies properties of the data attached to events. These predicates are useful if we want to pass on statements about the label and usage of a key or nonce, or statements about the order of events on the trace. In the context of this analysis, we show that multiple attacks violate the secrecy and authentication goals of the Otway-Rees protocol, and that it is therefore impossible to provide sufficient information through the encryption and event predicates to derive useful secrecy properties or properties about the relation of events that lead to statements about authentication.

Protocol Steps

Finally, we model the protocol code in a module `OYRS.Protocol` in terms of five steps realized as five independent functions that can be scheduled in arbitrary order. Functions for installing long-term keys and setting up the states of principals according to their initial knowledge are also part of the protocol API. For the initiator and responder, we have functions `initiator_init` and `responder_init`, which set up their state with their respective initial knowledge including the generation of long-term keys for secure communication with the authentication server. The server can then install these keys and associate them with the respective key peer in server sessions via `install_sk_at_auth_server`. Before one can schedule protocol steps, it is always required to first set up the initial knowledge of the principals involved in the protocol run, or runs in case of multiple parallel sessions.

First Step (Initiator) The first protocol step is realized by a function `initiator_send_msg_1` and comprises the start of a new protocol run by the initiator, who emits a message to the responder in which it provides data needed by the server to issue a verifiable session key certificate to the initiator. For demonstration, the implementation of the first protocol step is given below:

```
    val initiator_send_msg_1:
        a:principal ->
        a_ii:nat ->
        LCrypto (sess_idx:nat * message_idx:timestamp) (pki oyrs_preds)
        (requires fun t0 -> True)
        (ensures fun t0 (si, msg_idx) t1 -> msg_idx < trace_len t1 /\ (trace_len t0 <
trace_len t1))

    let initiator_send_msg_1 a a_ii =
        // get initiator session
        let now = global_timestamp () in
        let (|_,ser_st|) = get_session #oyrs_preds #now a a_ii in

        match parse_session_st ser_st with
        | Success (InitiatorInit srv k_as b) -> (
            // generate conversation id and initiator nonce
            let (|_,cid|) = rand_gen #oyrs_preds public (nonce_usage "conv_id") in
            let (|_,n_a|) = rand_gen #oyrs_preds (readers [P a; P srv]) (nonce_usage "nonce_i"
) in

            // trigger event 'initiate'
            let event = event_initiate cid a b n_a in
            trigger_event #oyrs_preds a event;

            // create and send first message
            let ev1 = EncMsg1 n_a cid a b in
            let now = global_timestamp () in
            let (|tag_ev1,ser_ev1|) = serialize_encval now ev1 (get_label oyrs_key_usages k_as
) in
            let c_ev1 = aead_enc #oyrs_global_usage #now #(get_label oyrs_key_usages k_as)
k_as (string_to_bytes #oyrs_global_usage #now "iv") ser_ev1 (string_to_bytes #
oyrs_global_usage #now "ev_i") in

            let msg1:message now = Msg1 cid a b (|tag_ev1,c_ev1|) in
            let ser_msg1 = serialize_msg now msg1 in

            let send_m1_idx = send #oyrs_preds a b ser_msg1 in

            // store initiator session
            let new_sess_idx = new_session_number #oyrs_preds a in
            let st_i_m1 = InitiatorSentMsg1 srv k_as b cid n_a in
            let now = global_timestamp () in
            let ser_st = serialize_session_st now a new_sess_idx 0 st_i_m1 in
            new_session #oyrs_preds #now a new_sess_idx 0 ser_st;

            (new_sess_idx, send_m1_idx)
        )
        | _ -> error "i_send_m1: wrong session"
```

The function receives two input parameters `a` and `a_ii`. The first parameter `a` is of type `principal` and denotes the identity of the initiator – the principal who executes the protocol step. The second parameter `a_ii` is the index of the state session containing the initiator's initial knowledge. This index is required by the initiator in order to retrieve its initial knowledge from its state at the beginning of the protocol step. Since state sessions are stored in their serialized form, the initiator next parses the retrieved state session. If the parsing is successful and the parsed state session has the structure `InitiatorInit srv k_as b`, where `srv` is the server identity, `k_as` is the long-term key of initiator and server, and `b` is the identity of the initiator, then the initiator continues with the first step. Otherwise, it aborts and returns an error stating that the session index points to a wrong state session. In case the state session has the expected structure, the initiator goes on to generate the public conversation identifier `cid` and its challenge `n_a` only intended for the initiator `a` itself and the server `srv`. With that, it has all the information to trigger the *initiate* event, signaling that it has initiated a protocol run and is about to send the first message to the responder. Afterwards, the initiator builds the first message for the responder. Therefore, it first builds the encryption value `EncMsg1 n_a cid a b` corresponding to the concatenation N_a, N_{cid}, A, B. The encryption value is then serialized via `serialize_encval` and encrypted via `aead_enc` under `k_as`, resulting in a ciphertext that corresponds to the encryption $\{N_a, N_{cid}, A, B\}^s_{k_{as}}$, which makes up the last part of the first message. We abstract from the use of initialization vectors and associated data in AEAD by using a static initialization vector `"iv"` and by associating data with the message that does not provide additional context to it, essentially resulting in pure symmetric encryption. Given the encrypted part of the message, the initiator can construct the first message as a whole using the `Msg1` constructor of the `message i` type and send it to the responder; the send function returns the index of the message on the trace. Finally, the initiator stores a state session `InitiatorSentMsg1` with information that it needs to verify the message it receives in the last protocol step. The function returns the session index of the newly created state session (needed by the initiator to retrieve the state session later), and the message index (needed by the responder to receive the message in the second protocol step).

Second Step (Responder) The second protocol step is implemented in a function `responder_send_msg_2`. In this step, the responder receives the first message that was previously emitted by the initiator, augments it with more data such that the server can issue a similar session key certificate to the responder, and finally forwards the request to the server. The implementation of the second step is given below:

```
    val responder_send_msg_2:
        b:principal ->
        msg1_idx: nat ->
        b_ii:nat ->
        LCrypto (sess_idx:nat * message_idx:nat) (pki oyrs_preds)
        (requires fun t0 ->  msg1_idx < trace_len t0)
        (ensures fun t0 (si, msg_idx) t1 -> msg_idx < trace_len t1 /\ (trace_len t0 <
trace_len t1))

    let responder_send_msg_2 b msg1_idx b_ii =
        // get responder session
        let now = global_timestamp () in
        let (|_,ser_st|) = get_session #oyrs_preds #now b b_ii in

        match parse_session_st ser_st with
        | Success (ResponderInit srv k_bs) -> (
            // receive and parse first message
            let (|_,a,ser_msg1|) = receive_i #oyrs_preds msg1_idx b in

            match parse_msg ser_msg1 with
            | Success (Msg1 cid a' b' (|tag_ev_a,c_ev_a|)) -> (
                if b <> b' then error "r_send_m2: responder in message does not match with
actual responder"
                else if a <> a' then error "r_send_m2: initiator in message does not match
with actual initiator"
                else
                    // generate responder nonce
                    let (|_,n_b|) = rand_gen #oyrs_preds (readers [P b; P srv]) (nonce_usage "
nonce_r") in

                    // trigger event 'request key'
                    let event = event_request_key cid a b n_b in
                    trigger_event #oyrs_preds b event;

                    // create and send second message
                    let ev2 = EncMsg2 n_b cid a b in
                    let now = global_timestamp () in
                    let (|tag_ev2,ser_ev2|) = serialize_encval now ev2 (get_label
oyrs_key_usages k_bs) in
                    let c_ev2 = aead_enc #oyrs_global_usage #now #(get_label oyrs_key_usages
k_bs) k_bs (string_to_bytes #oyrs_global_usage #now "iv") ser_ev2 (string_to_bytes #
oyrs_global_usage #now "ev_r") in

                    let c_ev_a:msg oyrs_global_usage now public = c_ev_a in
                    let msg2:message now = Msg2 cid a b (|tag_ev_a,c_ev_a|) (|tag_ev2,c_ev2|)
in

                    let ser_msg2 = serialize_msg now msg2 in

                    let send_m2_idx = send #oyrs_preds b srv ser_msg2 in

                    // store responder session
                    let new_sess_idx = new_session_number #oyrs_preds b in
                    let st_r_m2 = ResponderSentMsg2 srv k_bs a cid n_b in
                    let now = global_timestamp () in
                    let ser_st = serialize_session_st now b new_sess_idx 0 st_r_m2 in
                    new_session #oyrs_preds #now b new_sess_idx 0 ser_st;

                    (new_sess_idx, send_m2_idx)
            )
            | _ -> error "r_send_m2: wrong message"
        )
        | _ -> error "r_send_m2: wrong session"
```

Parameters are the responder identity `b`, the trace index of the first message `msg1_idx`, and the session index of the state session modeling the responder's initial knowledge `b_ii`. The responder starts – similarly to the initiator – by retrieving its initial knowledge consisting of the server identity `srv` and the long-term key with the server `k_bs`. Therefore, it expects the parsed state to have the structure `ResponderInit srv k_bs` and aborts if not. Afterwards, the responder receives the message emitted by the initiator in the first step, which it gets from the trace at `msg1_idx`. The receive function returns, besides the serialized message, also the identity of the sender – the initiator `a`. The serialized message is then parsed by the responder. If the parsing was successful and the message has the structure `Msg1 cid a' b' (|tag_ev_a,c_ev_a|)`, the responder verifies the contents of the message; otherwise, it aborts. During content verification, the responder particularly checks whether the initiator identity returned by the receive function and the responder identity passed as input to the protocol step match with the principals `a'` and `b'` in the first message. After the message has been verified, the responder generates its challenge `n_b`, which is supposed to remain secret to itself and the server. The rest of the second step is basically equivalent to the first step. First, the responder creates an encryption value similar to that of the initiator, but includes its own challenge instead. Then, the ciphertext resulting from the encryption is appended to the first message, which is serialized and sent to the server. Finally, the responder stores, among other things, conversation identifier, principal identities, and its challenge in a new state session for later use. Similarly as in the first step, the index of the created state session and the message trace index are returned.

All remaining protocol steps follow roughly the same pattern as the first two steps. First, any previously stored state sessions of the executing principal containing data needed for the particular protocol step are retrieved and parsed. Next, a possible message addressed to the principal, e.g., from a previous protocol step, is fetched from the trace, parsed, and possibly verified. Then, the principal might do some computations individual to the protocol step, like nonce or key generation. After that, a possible trace event is triggered and an output message is created and sent to the principal for whom it is destined. Finally, the principal's retrieved state session is updated or a new state session is created to store the newly generated nonces or keys or data from the received message. As this is essentially a universal pattern for implementing protocol steps, we will also use it later in the implementation of the Yahalom and Denning-Sacco protocol models.

Third Step (Server) The third step `server_send_msg_3` implements the authentication server's role in the protocol, which is the generation and distribution of a new session key as well as providing the users with assurance of its freshness and suitability for secret communication between them.

```
val server_send_msg_3:
    srv:principal ->
    msg2_idx: nat ->
    LCrypto (state_session_idx:nat * message_idx:nat) (pki oyrs_preds)
```

```
        (requires fun t0 ->  msg2_idx < trace_len t0)
        (ensures fun t0 (si, msg_idx) t1 -> msg_idx < trace_len t1 /\ (trace_len t0 <
trace_len t1))
```

The server first receives the second message, which was sent by the responder, from the trace. It parses the message and expects it to have the form `Msg2 cid a b' (|tag_ev_a,c_ev_a|) (|tag_ev_b, c_ev_b|)`, where `c_ev_a` and `c_ev_b` are respectively the encrypted message parts of the initiator from the first step and the responder from the second step. The server then verifies whether `b'` matches the responder's identity returned by the receive function. Afterwards, it uses the identities provided in the message to find the long-term keys `k_as` and `k_bs` shared with the initiator, and respectively the responder, in its state. With the long-term keys, the server is able to decrypt both the initiator's and the responder's encrypted parts containing their respective challenges `n_a` and `n_b` and the concatenation N_{cid}, A, B. As suggested by Otway and Rees in their description of the protocol, the server checks whether the conversation identifier and the user identities in the two encrypted parts match. The principal identities in the unencrypted part are however used exclusively for long-term key retrieval. After the server has performed all verifications, it continues by generating the session key `k_ab`, which it labels with `readers [P srv; P a; P b]` indicating that the key is supposed to remain secret to the server `srv` and the two users `a` and `b`. The server then triggers an event *send key* to signal that it has generated and now distributes a key in response to the responder's request, i.e., the second message. Further, it creates the third message `Msg3 cid (|tag_ev3_i,c_ev3_i|) (|tag_ev3_r,c_ev3_r|)` containing two encrypted parts `EncMsg3_I n_a k_ab` – encrypted under `k_as` – and `EncMsg3_R n_b k_ab` – encrypted under `k_bs` –, respectively, for the initiator and the responder. The third message is sent back to the responder, who forwards the initiator's part to the initiator in the fourth step. Although not mentioned by Otway and Rees, we also let the server store all terms it receives from the second message as well as the generated key in a new state session. The stored data allows us to specify and prove certain security properties from the server's point of view. In practice, the server may as well be stateless, except for the long-term key sessions that it must keep for its entire lifetime. Like the first two steps, the function outputs state session index and message index.

Fourth Step (Responder) Next, the responder performs the fourth protocol step `responder_send_msg_4`. In this step, the responder receives the session key for direct and secure communication with the initiator from the third message sent by the server, and forwards the session key certificate intended for the initiator.

```
    val responder_send_msg_4:
        b:principal ->
        msg3_idx:nat ->
        b_ii:nat ->
        b_si:nat ->
        LCrypto (message_idx:nat) (pki oyrs_preds)
        (requires fun t0 ->  msg3_idx < trace_len t0)
        (ensures fun t0 msg_idx t1 -> msg_idx < trace_len t1 /\ (trace_len t0 < trace_len t1))
```

The responder therefore retrieves and parses two state sessions at the beginning of the protocol step – the state session it stored at the end of the second step `ResponderSentMsg2 srv k_bs a cid n_b` and the state session representing its initial knowledge `ResponderInit srv' k_bs'`. The server principal `srv` and the long-term key `k_bs` are included in `ResponderSentMsg2` because the model was first implemented such that the responder simply updates its initial knowledge with the terms learned and generated in the second step. However, this implementation did not account for the possibility of multiple parallel protocol sessions and was thus changed such that the responder creates a separate state session in the second step. Since we already had a running model at that time, server identity and long-term key were not removed from the constructor of `ResponderSentMsg2`. Because F* cannot know that `srv = srv'` and `k_bs = k_bs'`, an explicit check by the responder is required. Subsequent to state session retrieval, the responder uses the receive function to get the third message from the trace. Before parsing, it verifies whether the server identity included in the two state sessions matches the sender principal returned by the receive function. The responder expects the parse function to return `Msg3 cid' (|tag_ev_a,c_ev_a|) (|tag_ev_b,c_ev_b|)`. It checks whether the conversation identifier in the message is the same as the one stored in its state session related to the current protocol run with the initiator `a`. If the conversation identifiers match, the responder decrypts the second encrypted part `c_ev_b`, which was encrypted under `k_bs`. The decrypted and parsed message part should have the form `EncMsg3_R n_b' k_ab`. The responder verifies its challenge `n_b` by checking that `n_b = n_b'` and triggers the *forward key* event to indicate that it accepts the session key `k_ab` and is about to forward the first encrypted part `c_ev_a` – containing session key and initiator challenge encrypted under the initiators long-term key – to the initiator. It then creates the last message `Msg4 cid (|tag_ev_a,c_ev_a|)` and sends it to the initiator. Finally, the responder updates the state session associated with the protocol run to `ResponderSentMsg4 srv a k_ab`. Since no new state session is created, the function only outputs the message trace index.

Fifth Step (Initiator) The last step is `initiator_recv_msg_4` and implements the initiator's role in the protocol during and after receiving the last message from the responder. The initiator receives the session key intended for communication with the responder – in a similar way as the responder – from a message originally emitted by the server, but forwarded to the intiator by the responder.

```
val initiator_recv_msg_4:
    a:principal ->
    msg4_idx:nat ->
    a_ii:nat ->
    a_si:nat ->
    LCrypto unit (pki oyrs_preds)
    (requires fun t0 -> msg4_idx < trace_len t0)
    (ensures fun t0 _ t1 -> True)
```

The outline of this step is quite similar to the previous step performed by the responder, except that the initiator does not send a message at the end. Again, two state sessions are retrieved: one corresponding to the protocol run from the initiator's point of view `InitiatorSentMsg1 srv k_as b cid n_a` and one corresponding to the initiator's initial knowledge `InitiatorInit srv' k_as' b'`. Terms appearing in both state sessions are checked regarding their equality. Then, the last message is received from the trace, the message sender is verified to be the stored responder `b`, and the message is parsed. The initiator expects a message of form `Msg4 cid' (|tag_ev_a,c_ev_a|)` and verifies whether `cid = cid'`. Next, it decrypts `c_ev_a` encrypted under `k_as` and parses the serialized encryption value. If the encryption really comes from the server, the parsed encryption value has the structure `EncMsg3_I n_a' k_ab`. The initiator verifies its challenge by checking that `n_a = n_a'`, triggers the *receive key* event to signal that the protocol has been completed and the key was accepted by the initiator, and finally updates its state session associated with the protocol run to `InitiatorRecvedMsg4 srv b k_ab`. The last step of the protocol does not create any new state sessions or send any messages; thus, the function outputs `unit`.

The result of a protocol run reflects in the state sessions of the principals in form of the established session key and the identities of the key peers as well as in the trace itself in form of the trace events associated with the protocol run. As we pointed out in Section 2.1, the principals' state sessions and the trace events can be used to specify and prove security properties involving terms stored in the state sessions or associated with the trace events.

Execution

Protocol code modeled in DY* cannot only be verified in F* regarding its soundness, but can also be compiled and executed to verify its correctness. To enable execution of our Otway-Rees model, we specify an entry point, from which we can call scheduler functions for protocol steps. In DY*, the network is under control of the attacker, which means that the attacker may embed protocol steps in its own code implemented in terms of the attacker API and schedule them in any order and arbitrarily often. The entry point for the Otway-Rees protocol is the module `OYRS.Debug`. At this point, we simply want to verify whether our model is correct with respect to its original specification. Therefore, we implement a scheduler function modeling a benign environment where the attacker schedules the protocol functions in the intended order and does nothing besides monitoring the network traffic, i.e., the messages sent by honest principals. The declaration and implementation of the function `benign_attacker` is given below:

```
val benign_attacker:
    unit ->
    LCrypto unit (pki oyrs_preds)
    (requires fun _ -> True)
    (ensures fun _ _ _ -> True)

let benign_attacker () =
    let a:principal = "initiator" in
    let b:principal = "responder" in
    let srv:principal = "server" in

    let ((|t_as,us_as,k_as|), a_ii) = initiator_init a srv b in
    let ((|t_bs,us_bs,k_bs|), b_ii) = responder_init b srv in
    install_sk_at_auth_server #t_as #us_as srv a k_as;
    install_sk_at_auth_server #t_bs #us_bs srv b k_bs;

    let (a_si, msg1_idx) = initiator_send_msg_1 a a_ii in
    let (b_si, msg2_idx) = responder_send_msg_2 b msg1_idx b_ii in
    let (srv_si, msg3_idx) = server_send_msg_3 srv msg2_idx in
    let msg4_idx = responder_send_msg_4 b msg3_idx b_ii b_si in
    initiator_recv_msg_4 a msg4_idx a_ii a_si
```

The function accepts an input of type `unit` and outputs `unit`. The output is wrapped in the `LCrypto` effect to account for an implicit global trace capturing protocol execution. Every expression evaluated in the function must therefore maintain the validity of the trace. The first three expressions declare the three principal roles involved in the protocol run, i.e., the initiator `a`, the responder `b`, and the authentication server srv. The next four expressions set up the states of the principals according to their initial knowledge including generation and installation of long-term keys. This results in two session indices pointing to the initial state sessions of initiator (a_ii) and responder (b_ii). After that, the actual protocol is executed by scheduling the protocol steps in the intended order and with intended inputs. The first three steps, respectively, output a session index and a message trace index; the fourth step only outputs a message trace index; the last step outputs unit. The output session indices of initiator (a_si) and responder (b_si) are later required in the fourth and fifth step to update the respective state sessions to their final form. The output message trace indices are respectively passed as input to the subsequent step for the principal receiving the message to fetch it from the trace. Protocol execution ends when the last step initiator_recv_msg_4 finishes.

The scheduler function benign_attacker is wrapped in a function benign that first calls benign_attacker and then prints the trace afterwards. The printed trace can help in debugging the model and verifying its correctness. Finally, we have the main function that is the actual entry point of our executable model:

```
    let main =
        IO.print_string "======================\n";
        IO.print_string "Otway-Rees            \n";
        IO.print_string "======================\n";
        let t0 = Seq.empty in
        IO.print_string "Starting Benign Attacker:\n";
        assume(valid_trace (pki oyrs_preds) t0);
        let r,t1 = (reify (benign ()) t0) in
        (match r with
        | Error s -> IO.print_string ("ERROR: "^s^"\n")
        | Success _ -> IO.print_string "PROTOCOL RUN: Successful execution of Otway-Rees
protocol.\n");
        IO.print_string "Finished Benign Attacker:\n";
```

The main function initially creates an empty trace t0. Functions that are wrapped in the LCrypto effect, like the benign function, implement the trace as implicit concept that is present during verification and does not actually reflect in the execution. In order to materialize the trace, we need the reify keyword, which takes the benign function and the initial trace t0 as input and executes benign with t0 as global trace. reify returns the result of the protocol execution – which contains either a value of type unit if execution was successful, or otherwise an error implicitly captured by the LCrypto effect – and the output trace t1.

4.3.3 Correctness and Coherence of the Model

The correctness of our Otway-Rees model can be verified in two steps. We start by type checking the individual modules of our model independently in F* to verify that the type restrictions made in the model are sound. The type check is also performed as part of the compilation process. If type check and compilation are successful, we can then run the protocol in a benign environment where the attacker remains passive and protocol steps are scheduled in the correct order, and can inspect the resulting global trace to see if it correctly depicts a regular protocol run. In Section 4.3.2, we discussed the implementation of a module OYRS.Debug with functions benign_attacker and benign for the simulation of such a benign environment and the main function being the entry point for execution. We have executed the model ourselves and printed the trace, which is depicted in

Appendix A.1. The trace shows a successful execution of our Otway-Rees model and captures all important protocol actions performed within the five protocol steps. Moreover, we see that the execution of our model results in the distribution of the session key generated by the authentication server to initiator and responder, which both store the key in their state after the last step. Therefore, we can verify that our model is correct and results in the expected outcome given a sufficiently benign environment. Nonetheless, we still need to argue that our model is coherent with the original protocol described by Otway and Rees in [27]. We support the coherence of our model with the original Otway-Rees protocol based on the following points:

- The initial knowledge of the principals is correctly represented in the model. That is, initiator and responder respectively share a long-term key with the authentication server, which implies that they know the server's identity and that the server also knows their identities. The initiator must furthermore know the responder such that it can address the first message to it.
- The origin of generated nonces and keys in the model corresponds to the origin specified in the protocol description. The initiator generates the conversation identifier N_{cid} and its challenge N_a, the responder only generates its own challenge N_b, and the server generates the session key k_{ab}. The origin of the long-term keys k_{as} and k_{bs} is not specified. We assume that the respective long-term keys could be generated either by the associated user or by the server. In our model, the users generate them and the server installs them afterwards. However, the protocol is not less secure if it were the other way around.
- The labels of the generated nonces and keys correctly represent their intended audience. The long-term keys are for secret communication between the users and the server (`readers [P a; P srv]`, `readers [P b; P srv]`), the conversation identifier is – by definition – `public`, the challenges only appear encrypted under the long-term keys in messages and are therefore as secret as the keys themselves, and the session key is distributed by the server and destined for secret communication between the users (`readers [P srv; P a; P b]`).
- The types `message i` and `encval` from the module `OYRS.Messages` correctly represent the original message structure, and the encryption values are encrypted under the correct long-term keys in the protocol steps. `EncMsg1` and `EncMsg3_I` are encrypted under the initiator's long-term key k_{as} (denoted `k_as` in the first, third, and fifth step), while `EncMsg2` and `EncMsg3_R` are encrypted under the responder's long-term key k_{bs} (denoted `k_bs` in the second, third, and fourth step).
- The type `session_st` from the module `OYRS.Sessions` correctly captures the stateful parts and outcome of the protocol, which is that the session key is stored in state sessions of the initiator and the responder at the end of protocol execution. The exchange of a key was one of the goals of the Otway-Rees protocol elaborated in Section 4.3.1 and thus defined for the model. Based on this, we already concluded that the execution trace in Appendix A.1 verifies the model's correctness. Since Otway and Rees clearly state that initiator and responder both obtain the session key, respectively, in the steps 4 and 5, we can also conclude that the model is valid. The fact that the server stores the session key might impose a security risk in practice. However, we have already argued that this is purely symbolic in order to be able to prove certain security properties, such as the secrecy of the session key, from the servers point of view.
- Checks performed on message contents in the protocol steps implemented in the module `OYRS.Protocol` are reasonable with respect to the protocol description. In steps in which messages are received, verification of the message sender identity is performed. This is not

restrictive because the attacker is still able to spoof the message sender via the attacker API. In the third step, the server checks the concatenation N_{cid}, A, B in the two encrypted parts. This was required in the protocol description to ensure that initiator and responder agree on the principals trying to establish a key between them. We deliberately omit checks involving the concatenation N_{cid}, A, B in the unencrypted part, since nothing is mentioned in this regard. In the fourth and fifth step, the initiator and responder respectively check the conversation identifier and verify their challenges. According to Otway and Rees, the purpose of N_{cid} is "to associate all the messages in the same authentication sequence" (p. 9). Since N_{cid} is not encrypted in the third and fourth message, we cannot get any assurance from it regarding security. We can however ensure the correctness of the protocol. The challenge verification is explicitly mentioned by Otway and Rees in the description and is thus definitely required.

4.3.4 Security Properties

Having developed a model of the Otway-Rees protocol in DY* and having established its correctness as well as its coherence with the original protocol specification, we can now define and prove symbolic security properties for the protocol in DY* based on the security objectives that were informally described in Section 4.3.1. We extend our Otway-Rees model with a module `OYRS.SecurityProps` for the specification of security properties and their verification via F*. For properties that cannot be proven, we identify reasons why the proof fails and argue whether an attack is feasible. The security goals of the Otway-Rees protocol are divided into *secrecy* and *authentication* goals. To capture the secrecy goals, we have expressed the following two requirements regarding the exchanged key:

(S1) the session key is kept secret between the users – the initiator and responder – and the authentication server; and

(S2) the key received and accepted by the users is in fact the session key generated by the server.

The authentication goals can also be divided into two requirements:

(A1) for users A and B that accept a session key k_{ab}, there is a corresponding successful protocol run with initiator A and responder B; and

(A2) A and B shall be convinced that they were talking to each other.

Secrecy

The secrecy requirement S1 is expressed by the following lemma in terms of the DY* model of the Otway-Rees protocol:

```
val session_key_stored_in_auth_server_state_is_secret:
    (server:principal) ->
    (set_state_idx:nat) ->
    (vv:version_vec) ->
    (server_state:state_vec) ->
    (state_session_idx:nat) ->
    (initiator:principal) ->
    (responder:principal) ->
```

```
        (sess_key:bytes) ->
        LCrypto unit (pki oyrs_preds)
        (requires (fun in_tr ->
            principals_and_session_key_stored_in_auth_server_state server set_state_idx vv
server_state state_session_idx initiator responder sess_key /\
            set_state_idx < trace_len in_tr
        ))
        (ensures (fun in_tr _ out_tr ->
            in_tr == out_tr /\
            (
                is_unknown_to_attacker_at (trace_len in_tr) sess_key \/
                (
                    corrupt_id (trace_len in_tr) (P server) \/
                    corrupt_id (trace_len in_tr) (P initiator) \/
                    corrupt_id (trace_len in_tr) (P responder)
                )
            )
        ))
```

The result of lemmas capturing security properties of protocol models in DY* is generally `LCrypto unit`. The `LCrypto` effect is needed because security properties of protocol models often depend on properties of DY*'s global trace, and the result type `unit` coupled with the property `in_tr == out_tr` guaranteed by the function indicates that the function is indeed a lemma that neither produces a meaningful result nor modifies the trace, but only establishes properties of the trace and hence the protocol in F*'s proof context. In the pre-condition of the lemma specified by the `requires` clause, the server's state `server_state` is assumed to contain a state session with index `state_session_idx` in which the principals `initiator` and `responder` and the session key `sess_key` are stored. The respective state session must furthermore be stored on the input trace `in_tr`, which is a snapshot of the trace at the time the lemma is instantiated. The `ensures` clause contains, among other things, the property to prove. As mentioned above, the first property ensured guarantees that the global trace is not altered by the function. It follows the actual secrecy property, which states that `sess_key` remains unknown to the attacker as long as none of the state session identifiers `P server`, `P initiator`, or `P responder`, is corrupted at `trace_len in_tr`. The state session identifiers respectively denote arbitrary state sessions of the server, initiator, and responder. Therefore, an alternative formulation of the secrecy property becomes: the session key is secret if there is no arbitrary state session of the server, initiator, or responder that is corrupted.

To verify `session_key_stored_in_auth_server_state_is_secret` in F*, we need to instantiate the generic `secrecy_lemma` defined in the labeled security layer of DY* in its implementation:

```
    let session_key_stored_in_auth_server_state_is_secret
        server
        set_state_idx
        vv
        server_state
        state_session_idx
        initiator
        responder
        sess_key
    =
        match LabeledPKI.parse_session_st server_state.[state_session_idx] with
        | Success (APP ser_st) -> (
            match OYRS.Sessions.parse_session_st ser_st with
            | Success (AuthServerSentMsg3 a b cid n_a n_b k_ab) ->
                secrecy_lemma #(pki oyrs_preds) sess_key
            | _ -> ()
        )
        | _ -> ()
```

We first parse the state session of the server using the `parse_session_st` functions of `LabeledPKI`, which is part of DY*'s labeled layer and models generic state sessions to store private keys, for example, and `OYRS.Sessions`, which specifically models the stateful parts of the Otway-Rees protocol. As required by `principals_and_session_key_stored_in_auth_server_state`, the state session has the structure `AuthServerSentMsg3 a b cid n_a n_b k_ab` with `a = initiator`, `b = responder`, and `k_ab = sess_key`. In the third protocol step, the server labels the session key with `readers [P srv; P a; P b]`, which we now know is equal to `readers [P server; P initiator; P responder]`. Given that the label of `sess_key` is `readers [P server; P initiator; P responder]`, the secrecy lemma provides that if the attacker knows `sess_key`, then one of the readers, i.e., `P server`, `P initiator`, or `P responder`, must be corrupted. From this, the property ensured by `session_key_stored_in_auth_server_state_is_secret` is easily derived and can be verified in F*. We conclude that the session key generated by the authentication server is secure if the server as well as the users obtaining the key from the server are honest.

This does howbeit not mean that the key obtained by initiator and responder at the end of the protocol is necessarily the session key generated by the server as required by the secrecy requirement S2. In fact, there are several well-known attacks on the Otway-Rees protocol that violate this objective. Consequently, we cannot proof lemmas in DY* that ensure the secrecy of the key stored in the respective state sessions of initiator and responder.

Authentication

The fact that the attacker is able to convince the users to accept a key as session key that has not been chosen by the authentication server does not only break secrecy, but also authentication. In particular, requirement A1 is violated by one of the attacks we present, and in another attack, only one user accepts a session key that it thinks it shares with the other user, so A1 is also violated. Requirement A2 is violated by two attacks we present, i.e., there are at least two occasions where at least one of the users is not convinced that he or she has talked to the other. Interestingly, there is one attack that does not violate any of the authentication requirements, meaning that there is a successful protocol run between an honest initiator and responder, where both accept the same session key and both are convinced to talk to each other, but the key is known to the adversary, thus not secret.

4.3.5 Attacks

We extend our DY* model of Otway-Rees with attacks related to unfulfilled secrecy and authentication goals of the protocol described in Section 4.3.4. This ultimately leads to the proposal of an improved version of the Otway-Rees protocol that is able to achieve all remaining security objectives not achievable by the original protocol. We implement the attacks with the attacker API provided by the symbolic runtime layer of DY*. The steps executed by the attacker in order to carry out the attacks are defined and implemented in a separate module `OYRS.Attacker`. The attacker can then combine these steps with steps executed by honest principals from the module `OYRS.Protocol` in order to realize the attacks. We implement the attacks in the module `OYRS.Debug` in a similar way as the function `benign_attacker`, which simulates a protocol run where the attacker remains passive.

Impersonation Attack

In Section 3.3.1, we mentioned the attack of Boyd and Mao [7] on the Otway-Rees protocol. In this attack, the attacker makes use of the fact that the server does not check whether the concatenation N_{cid}, A, B in the plaintext part of the second message matches with the copies in the two ciphertexts. This enables the adversary to impersonate the responder by replacing the second protocol message with a message $N_{cid}, A, E, \{N_a, N_{cid}, A, B\}^s_{k_{as}}, \{N_e, N_{cid}, A, B\}^s_{k_{es}}$, where E is a principal controlled by the attacker. The server believes that initiator and responder both agree about who wants to establish communication with whom because the concatenations N_{cid}, A, B in the encrypted parts match. As a result, it does not abort the third protocol step and issues certificates to the users containing the session key. The initiator A did however not actually agree to talk to E, but instead wanted to talk to B. To carry out the attack, the attacker must impersonate the responder in the second and in the fourth protocol step. The attacker logic for the second step is implemented by a function `attacker_send_mal_msg_2`:

```
    module A = AttackerAPI

    let attacker_send_mal_msg_2 (#i:nat) (eve srv:principal) (msg1_idx:nat) (k_es:A.pub_bytes
i) :
        Crypto (timestamp * g_trace)
        (requires (fun t0 -> i <= trace_len t0 /\ msg1_idx < trace_len t0))
        (ensures (fun t0 r t1 ->
            match r with
            | Error _ -> (t0 == t1 \/
                (A.attacker_modifies_trace t0 t1 /\ later_than (trace_len t1) (trace_len t0)))
            | Success (n, t01) ->
                trace_len t1 = trace_len t0 + 3 /\ later_than (trace_len t1) (trace_len t0) /\
                n = trace_len t1 - 1 /\ A.attacker_modifies_trace t0 t01 /\
                A.attacker_modifies_trace t01 t1))
    = // receive and parse first message
        let (|t_m1,ser_msg1|) = A.receive_i msg1_idx eve in

        match
            A.split ser_msg1 `bind` (fun (tag_bytes, rest) ->
            A.pub_bytes_to_string tag_bytes `bind` (fun tag ->
            match tag with
            | "msg1" ->
                A.split rest `bind` (fun (cid, rest) ->
                A.split rest `bind` (fun (a_bytes, rest) ->
                A.split rest `bind` (fun (b_bytes, ev_a) ->
                Success (cid,a_bytes,b_bytes,ev_a))))
            | t -> Error ("attacker_send_mal_m2: wrong message: " ^ t)
            ))
        with
        | Success (cid,a_bytes,b_bytes,c_ev_a) ->
            // generate "responder" nonce
            let (|now,n_e|) = A.pub_rand_gen (nonce_usage "nonce_attacker") in
            let intermediate_trace = get () in

            // create and send malicious second message
            let cid = A.pub_bytes_later t_m1 now cid in
            let a_bytes = A.pub_bytes_later t_m1 now a_bytes in
            let b_bytes = A.pub_bytes_later t_m1 now b_bytes in
            let e_bytes = A.pub_bytes_later 0 now (A.string_to_pub_bytes eve) in
            let c_ev_a = A.pub_bytes_later t_m1 now c_ev_a in

            let ev2 = A.concat n_e (A.concat cid (A.concat a_bytes b_bytes)) in
            let k_es = A.pub_bytes_later i now k_es in
            let iv = A.pub_bytes_later 0 now (A.string_to_pub_bytes "iv") in
            let ad = A.pub_bytes_later 0 now (A.string_to_pub_bytes "ev_r") in
            let c_ev2 = A.aead_enc #now k_es iv ev2 ad in

            let msg2_tag = A.pub_bytes_later 0 now (A.string_to_pub_bytes "msg2") in
            let ser_msg2 = A.concat msg2_tag (A.concat cid (A.concat a_bytes (A.concat e_bytes
 (A.concat c_ev_a c_ev2)))) in

            let send_mal_m2_idx = A.send #now eve srv ser_msg2 in

            (send_mal_m2_idx, intermediate_trace)
      | Error e -> error e
```

The prefix `A` on function names indicates that the respective functions are part of the attacker API and are not to be confused with their possible counterparts from the regular API for cryptographic functions provided by the symbolic runtime layer. Input parameters of the function are the principals `eve` (the principal controlled by the attacker) and `srv` (the authentication server), the trace index of the first message from the initiator `msg1_idx`, and the long-term key of `eve` and the server `k_es`. Since the attacker API is part of the symbolic runtime layer, we annotate the function with the `Crypto` effect instead of the `LCrypto` effect. The output type is a tuple `timestamp * g_trace`, where the first part is the trace index of the second message from the attacker to the server, and the second part is the intermediate trace after the attacker generates the nonce `n_e`. The intermediate trace is needed in the `ensures` clause to ensure the validity of the output trace via the `A.attacker_modifies_trace` predicate. The attacker first receives the first message from the initiator as `eve` using the `A.receive_i` function from the attacker API. Next, the attacker parses the message, which results in the tuple `(cid,a_bytes,b_bytes,c_ev_a)` if the message was correctly built by the initiator. `a_bytes` and `b_bytes` are respectively the serialized principals `a` (the initiator) and `b` (the actual responder), `cid` is the conversation identifier, and `c_ev_a` is the ciphertext $\{N_a, N_{cid}, A, B\}^s_{k_{as}}$ encrypted by the initiator. If the attacker could successfully parse the first message, it generates a nonce `n_e` with `A.pub_rand_gen` and stores a view of the current trace in `intermediate_trace`. Then, the malicious second message is created. The `A.pub_bytes_later` function is used to ensure that the attacker knows all terms contained in the created message at creation time (the trace length when the message is created). The attacker creates its own encryption value matching that of the initiator from the first message by concatenating `n_e`, `cid`, `a_bytes`, and `b_bytes`, and encrypts the resulting `bytes` value `ev2` under `k_es` to receive the ciphertext `c_ev2`. Since the attacker needs to ensure that the server can decrypt c_ev2, it replaces b_bytes from the unencrypted part of the first message with e_bytes (the bytes representation of the principal eve) in the unencrypted part of the second message. The attacker then appends the ciphertext c_ev_a from the first message and the ciphertext c_ev2 it just created to the modified plaintext part of the first message and stores the resulting message in ser_msg2. Finally, the attacker, on behalf of eve, sends the malicious message ser_msg2 to the server via A.send. The message looks perfectly fine to the server because it can successfully decrypt the two ciphertexts and it finds two matching encryption values with corresponding conversation identifier cid, and principals a and b. The function outputs the trace index of the sent message and the snapshot of the trace directly after n_e was generated.

The function attacker_send_msg_4 implements the attacker logic of the fourth protocol step based on a similar pattern. It takes four parameters: the three principals eve, bob and alice, and the trace index of the third message from the server msg3_idx. Again, eve is the principal that the adversary controls; bob and alice are, respectively, the actual responder and the initiator. The first action performed by the attacker in this step is to receive the third message that the server sent as response to the attacker's malicious second message. The adversary then parses the message, which should result in a tuple (cid,c_ev_a,c_ev_e). It decrypts the ciphertext c_ev_e using eve's long-term key k_es – which it can because it controls eve – and obtains the challenge n_e, which it ignores, as well as the session key k_ae generated by the server. Finally, the adversary creates the fourth protocol message by concatenating cid and c_ev_a, and sends it to alice for her to receive k_ae as well. Since alice intended to talk to bob, she believes that k_ae is intended for secret communication with bob. However, as this attack demonstrates, eve is the one that alice actually shares a key with. The function attacker_send_msg_4 returns the trace index of the fourth message and the obtained session key, which the attacker can use to masquerade as bob to alice.

We schedule the individual steps of the attack in a function impersonate_resp_to_init_attacker in the module OYRS.Debug:

```
let impersonate_resp_to_init_attacker () =
    let a:principal = "alice" in
    let b:principal = "bob" in
    let srv:principal = "server" in
    let e:principal = "eve" in

    let ((|t_as,us_as,k_as|), a_ii) = initiator_init a srv b in
    let ((|t_bs,us_bs,k_bs|), b_ii) = responder_init b srv in
    let before_idx_state_e = global_timestamp () in
    let ((|t_es,us_es,k_es|), e_ii) = responder_init e srv in
    install_sk_at_auth_server #t_as #us_as srv a k_as;
    install_sk_at_auth_server #t_bs #us_bs srv b k_bs;
    install_sk_at_auth_server #t_es #us_es srv e k_es;

    let idx_comp_e = compromise e e_ii 0 in

    let now = global_timestamp () in
    let k_es = query_secret_key (before_idx_state_e + 1) idx_comp_e now e e_ii 0 in

    let (a_si, msg1_idx) = initiator_send_msg_1 a a_ii in
    let (msg2_idx, _) = attacker_send_mal_msg_2 e srv msg1_idx k_es in
    let (srv_si, msg3_idx) = server_send_msg_3 srv msg2_idx in
    let (msg4_idx, sess_key) = attacker_send_msg_4 e b a msg3_idx k_es in
    initiator_recv_msg_4 a msg4_idx a_ii a_si;

    attacker_knows_session_key_stored_in_initiator_or_responder_state a a_si sess_key;
    // responder "bob" was not involved in protocol run
    initiator_believes_talking_to_responder a a_si b
```

We define four principals: `a`, `b`, `srv`, and `e`. `a` takes the role of the initiator, `b` and `e` that of the responder, and `srv` that of the authentication server. We set up the initial knowledge of the principals accordingly. Since the initiator `a` wishes to talk to `b`, its initial knowledge is set up via `initiator_init a srv b`. This means that `a` generates a long-term key for communication with `srv`, and stores `srv` along with the key and the principal `b` it wishes to communicate with in a state session. The initial knowledge of the two responders includes only the server `srv` and a long-term key to communicate with `srv`. After the server has installed the long-term keys of `a`, `b` and `e`, the attacker compromises `e`'s long-term key state session in order to take control over `e`. It queries the secret key `k_es` of `e` and the server stored in the long-term key state session. Being in control of `e`, the adversary can carry out the attack by scheduling the protocol steps of the initiator `a` and the server `srv` as intended and by impersonating `b` in the second and fourth step. The server is aware that the second message comes from `e` but does not know that `a` actually wanted to establish a key with `b` instead. With `attacker_send_msg_4` the adversary obtains the session key `sess_key` and forwards it to the initiator, which accepts it in the last step `initiator_recv_msg_4`. The last two expressions are instantiations of two lemmas that fail if certain conditions are not met after the attack. With `attacker_knows_session_key_stored_in_initiator_or_responder_state a a_si sess_key`, we explicitly require that the attacker knows the session key stored in the initiator state as a result of the attack. `initiator_believes_talking_to_responder a a_si b` requires that the initiator believes it shares a key with `b`.

Like `benign_attacker`, we wrap `impersonate_resp_to_init_attacker` in a function `impersonate_resp_to_init` , which additionally prints the trace. The output trace of the attack is depicted in Appendix A.2. It shows that the attack is successful, which implies that the attacker has access to a session key generated by the server for `a` and `e`, but `a` believes it shares the key with `b`. The responder `b` was not involved in the protocol at all and therefore neither knows the session key nor does it believe it has recently talked to `a`. Thus, the secrecy objective S2, as well as the authentication objectives A1 and A2 from Section 4.3.4 are not fulfilled.

Intercept and Replay Attacks

Besides the impersonation attack described and modeled above, the Otway-Rees protocol is also vulnerable to attacks in which the attacker intercepts and replays messages from the initiator and possibly the responder to make them accept a session key known to the attacker. These attacks make use of the fact that the encryptions $\{N_a, N_{cid}, A, B\}^s_{k_{as}}$ of the initiator and $\{N_b, N_{cid}, A, B\}^s_{k_{bs}}$ of the responder in the second message could be confused with the encryptions $\{N_a, k_{ab}\}^s_{k_{as}}$ and $\{N_b, k_{ab}\}^s_{k_{bs}}$ in the response from the server if k_{ab} and N_{cid}, A, B had the same length. Although this type of attack is easy to avoid at the implementation level, the Otway-Rees protocol specification proposes an implementation that does not perform the necessary checks to prevent it, so we consider this to be a vulnerability of the protocol.

The easiest way for the adversary to carry out an attack would be to simply replay the initiator's encryption from the first message and remove the two principal identities after the conversation identifier in the unencrypted part. We realize this attack in terms of our Otway-Rees model in a function `intercept_msg_1_attacker` in the `OYRS.Debug` module:

```
let intercept_msg_1_attacker () =
    let a:principal = "initiator" in
    let b:principal = "responder" in
    let srv:principal = "server" in

    let ((|t_as,us_as,k_as|), a_ii) = initiator_init a srv b in
    let ((|t_bs,us_bs,k_bs|), b_ii) = responder_init b srv in
    install_sk_at_auth_server #t_as #us_as srv a k_as;
    install_sk_at_auth_server #t_bs #us_bs srv b k_bs;

    let (a_si, msg1_idx) = initiator_send_msg_1 a a_ii in
    let (msg4_idx, sess_key) = attacker_intercept_msg_1 b a msg1_idx in
    initiator_recv_msg_4 a msg4_idx a_ii a_si;

    attacker_knows_session_key_stored_in_initiator_or_responder_state a a_si sess_key;
    // responder was not involved in protocol run
    initiator_believes_talking_to_responder a a_si b
```

The attacker logic is again implemented in the module `OYRS.Attacker` via a function `attacker_intercept_msg_1`. In this kind of attack, the adversary does not require an additional principal under its control. Therefore, we allocate roles to principals and set up their initial knowledge similarly as in `benign_attacker` where we simulated a regular protocol run. The attack begins again with the initiator `a` executing the first step and sending the first message to the responder `b`. The message is however intercepted by the adversary, who executes `attacker_intercept_msg_1` and replays `a`'s encryption $\{N_a, N_{cid}, A, B\}^s_{k_{as}}$ in a message to `a` in the name of `b`. `attacker_intercept_msg_1` also outputs the `bytes` value representing N_{cid}, A, B taken from the unencrypted part of the first message that will later be accepted as session key by `a`. We then skip to the last step where `a` receives the message and interprets N_{cid}, A, B (the second part of the encryption) as the session key coming from the server. The outcome is the same as in the impersonation attack, that the attacker knows the session key and that the initiator `a` believes it has exchanged a key with the responder `b`, leaving the security requirements S2, A1 and A2 unfulfilled.

If we go further, we can even launch an attack in which both the initiator `a` and the responder `b` accept the above term as session key. Therefore, we must await the second message of the responder before we intercept and craft a message $N_{cid}, \{N_a, N_{cid}, A, B\}^s_{k_{as}}, \{N_b, N_{cid}, A, B\}^s_{k_{bs}}$ that `a` and `b` will interpret as the third message from the server. This attack is carried out in another function `intercept_msg_2_attacker` in `OYRS.Debug`:

```
let intercept_msg_2_attacker () =
    let a:principal = "initiator" in
    let b:principal = "responder" in
    let srv:principal = "server" in

    let ((|t_as,us_as,k_as|), a_ii) = initiator_init a srv b in
    let ((|t_bs,us_bs,k_bs|), b_ii) = responder_init b srv in
    install_sk_at_auth_server #t_as #us_as srv a k_as;
    install_sk_at_auth_server #t_bs #us_bs srv b k_bs;
```

```
let (a_si, msg1_idx) = initiator_send_msg_1 a a_ii in
let (b_si, msg2_idx) = responder_send_msg_2 b msg1_idx b_ii in
// third message from auth server is discarded
let (msg3_idx, sess_key) = attacker_intercept_msg_2 srv b msg2_idx in
let msg4_idx = responder_send_msg_4 b msg3_idx b_ii b_si in
initiator_recv_msg_4 a msg4_idx a_ii a_si;

attacker_knows_session_key_stored_in_initiator_or_responder_state a a_si sess_key;
attacker_knows_session_key_stored_in_initiator_or_responder_state b b_si sess_key;
initiator_and_responder_talk_to_each_other a b a_si b_si
```

The set up phase is completely the same as in `benign_attacker` or `intercept_msg_1_attacker`. However, we schedule the first two steps of the initiator `a` and the responder `b` as if we would run the protocol regularly. Then, after the second step, the adversary executes `attacker_intercept_msg_2` (again defined in `OYRS.Attacker`) in which it proceeds in the same way as in the first interception attack. It replays the second message without the two principal identities of `a` and `b` in the unencrypted part to `b` on behalf of the server. From this point, the protocol execution continues regularly and both `a` and `b` will interpret the second part of their respective encryption as the session key. The result of this attack is that the initiator `a` as well as the responder `b` both accept N_{cid}, A, B as session key and believe that they talk to each other. The particularity here is that both authentication objectives A1 and A2 are fulfilled, since both `a` and `b` have completed a protocol run in which `a` was the initiator and `b` the responder, and they are convinced that they share the exchanged key with each other. The attacker, who knows the session key, can therefore not only impersonate either of them to the other, but also eavesdrop the communication between them. Appendix A.3 shows the respective output traces of the two described *intercept and replay* attacks.

Unusual Message Model

In Section 4.3.2, we pointed out that messages and encryption values are tagged during serialization such that they can be correctly reconstructed when parsed. For messages, this tag is directly included in the serialized message and can be reconstructed during the parsing process. Since the messages are publishable, this does not increase security by any means because the adversary can simply recreate the message with a different tag and hence trick another principal into interpreting the message in a different way than it was intended. The encryption values, on the other hand, are protected by encryption; had we included the tag in the serialized value, the attacker would not be able to use it in another context. Instead, the tag is added to the serialized value afterwards and discarded when the corresponding message itself is serialized. The receiver of the message simply infers the tag of the encryption value from the surrounding context (the tagged serialized message) during the parsing process. Also, we cannot include information about the context of an encryption value in the associated data of an AEAD ciphertext, since this would also lead to the detection of encryption values used in the wrong context. These two features of the model allow the attacker to replay messages in a different context than the one for which they were originally created, thus making the two replay attacks possible.

Significance of the Attacks

While the replay attacks are hardly feasible in practice unless the concatenation N_{cid}, A, B has the same length as the session key k_{ab}, the impersonation attack described first is more severe, because it depends only on the insecure implementation of the authentication server in the Otway-Rees protocol, which, according to the protocol description, does not check whether N_{cid}, A, B in the plaintext part of the second message matches the two copies in the ciphertexts of the initiator and responder. Even if the authentication server would perform said check, this would still not be ideal. We emphasize the importance of the robustness principle proposed by Boyd and Mao [7] when designing cryptographic protocols and therefore propose an improved Otway-Rees protocol in the next section that suffices to achieve remaining security goals that were defeated by the attacks described in this section. Later, we show why the fix provided by Boyd and Mao themselves is not in accordance with their robustness principle and consequently does not prevent the impersonation attack.

4.3.6 Improved Protocol and Model

Our goal is to achieve the remaining security objectives from Section 4.3.4 with as little alterations of the protocol as possible. In this section we propose several changes to the structure and implementation of the Otway-Rees protocol in order to achieve the secrecy goal S2, which states that the session key accepted by the users must be chosen by the server, and the authentication goals A1 and A2, which require a corresponding protocol run with one user as initiator and one user as responder that convinces both of the correctness of each other's identity.

Achieving Secrecy

For the secrecy objective S2, we propose the following modification of the original Otway-Rees protocol depicted in Section 3.3.1:

1. $A \rightarrow B$: $N_{cid}, A, B, \{N_{cid}, A, B, N_a\}^s_{k_{as}}$
2. $B \rightarrow AS$: $N_{cid}, A, B, \{N_{cid}, A, B, N_a\}^s_{k_{as}}, \{N_{cid}, A, B, N_b\}^s_{k_{bs}}$
3. $AS \rightarrow B$: $N_{cid}, \{N_a, B, k_{ab}\}^s_{k_{as}}, \{N_b, A, k_{ab}\}^s_{k_{bs}}$
4. $B \rightarrow A$: $N_{cid}, \{N_a, B, k_{ab}\}^s_{k_{as}}$

In particular, the changes made to the protocol messages are

- a restructuring of the encryption values in the first and second message so that the challenges are swapped with the conversation identifier (e.g., $\{N_a, A, B, N_{cid}\}^s_{k_{as}}$ becomes $\{N_{cid}, A, B, N_a\}^s_{k_{as}}$); and
- the inclusion of the respective key peers in the ciphertexts in the third and fourth message (e.g., $\{N_a, k_{ab}\}^s_{k_{as}}$ becomes $\{N_a, B, k_{ab}\}^s_{k_{as}}$).

The first modification prevents the two *intercept and replay* attacks described in Section 4.3.5. If we take A as example, the adversary cannot use the encryption $\{N_{cid}, A, B, N_a\}^s_{k_{as}}$ from the first message to replace $\{N_a, k_{ab}\}^s_{k_{as}}$ in the fourth message anymore, as both encryptions have a fundamentally different structure now. In case the adversary tries to replay one of the ciphertexts from the first two messages in the context of the third and fourth message, this will be detected by the users. The second modification, on the other hand, prevents the impersonation attack in which the intruder controls a malicious principal E and impersonates B to A. In the third step, the server emits a message of the form $N_{cid}, \{N_a, E, k_{ae}\}^s_{k_{as}}, \{N_e, A, k_{ae}\}^s_{k_{es}}$. The attacker as E is able to obtain the key and then forwards $N_{cid}, \{N_a, E, k_{ae}\}^s_{k_{as}}$ to A, who thinks the message comes from B. When A receives the last message, it decrypts $\{N_a, E, k_{ae}\}^s_{k_{as}}$, realizes that the actual key peer of the obtained key k_{ae} is E, and aborts. If the protocol is realized with AEAD as operating mode for encryption, we could realize the inclusion of the key peer identities in the server message at the implementation-level by including them in the associated data attached to the AEAD ciphertext instead of the ciphertext itself. This is however an implementation detail not addressed by Otway and Rees in the protocol specification, so we felt that including them directly in the ciphertext was the more general solution.

In addition to the changes made to the message structure, we also propose an implementation-level measure to further improve the security of the Otway-Rees protocol: the server checks whether N_{cid}, A, B in the unencrypted part of the second message matches the two copies in the encrypted parts. With this change, we further ensure that the attacker cannot perform the impersonation attack from Section 4.3.5 and that the secrecy goal S2 is met.

Achieving Mutual Authentication

The authentication goals A1 and A2 heavily depend on the secrecy of the session key. We can thus show that the changes proposed so far in addition with two small modification of the events suffice to proof mutual and timely authentication in the improved version of the protocol – which was the main goal of Otway and Rees with their work. The first adaption is for all events to include the identity of the authentication server, who plays an important role in the authentication process; the second adaption is to include users' nonces in the events *forward key* and *receive key*, since they are relevant for timeliness guarantees when completing the protocol.

Adapting the DY* Model

We adapt our DY* model of the Otway-Rees protocol from Section 4.3.2 to incorporate the modifications proposed above, and then define and prove the remaining security properties from Section 4.3.4 for the improved model.

Messages The modifications to the message structure reflect in the type `encval` defined in the `OYRS.Messages` module. `encval` represents the structure of the encrypted parts that appear in the messages of the protocol.

```
noeq type encval =
    | EncMsg1: cid:bytes -> a:string -> b:string -> n_a:bytes -> encval
    | EncMsg2: cid:bytes -> a:string -> b:string -> n_b:bytes -> encval
```

```
        | EncMsg3_I: n_a:bytes -> b:principal -> k_ab:bytes -> encval
        | EncMsg3_R: n_b:bytes -> a:principal -> k_ab:bytes -> encval
```

In `EncMsg1` and `EncMsg2`, the parameter `cid`, which represents the conversation identifier, has been swapped with the parameters `n_a` and `n_b`, which represent the initiator and responder challenge, respectively. In `EncMsg3_I` and `EncMsg3_R`, parameters `b` and `a` for the respective key peers of `k_ab` have been added.

State To show that the session key is secure from the initiator's and responder's point of view, we need to adjust the `valid_session` predicate defined in `OYRS.Sessions` in the `match` arms of the initiator's and responder's respective final states accordingly:

```
    let valid_session (i:nat) (p:principal) (si vi:nat) (st:session_st) =
        // other state sessions remain the same
        | ResponderSentMsg4 srv a k_ab ->
           is_msg i k_ab (readers [P p]) /\
           (corrupt_id i (P p) \/ corrupt_id i (P srv) \/ corrupt_id i (P a) \/
           is_labeled i k_ab (readers [P srv; P a; P p]))
        | InitiatorRecvedMsg4 srv b k_ab ->
           is_msg i k_ab (readers [P p]) /\
           (corrupt_id i (P p) \/ corrupt_id i (P srv) \/ corrupt_id i (P b) \/
           is_labeled i k_ab (readers [P srv; P p; P b]))
```

In Section 4.3.4, we used the secrecy lemma to prove the secrecy of `k_ab` stored in the state session of the server, which is essentially a proof of the secrecy objective S1. For a similar proof regarding `k_ab` stored in the state sessions `ResponderSentMsg4` of the responder and `InitiatorRecvedMsg4` of the initiator, we require a statement on the label of the key. In case of the responder's state session, the statement we require is `corrupt_id i (P p) \/ corrupt_id i (P srv) \/ corrupt_id i (P a) \/ is_labeled i k_ab (readers [P srv; P a; P p])`. This says that either one of the state session identifiers `P p`, `P srv` or `P a` is corrupted, or the label of `k_ab` is `readers [P srv; P a; P p]`. Given the label, we can later show that the adversary cannot learn `k_ab` unless one of its readers is corrupted. The principal `p` here is the owner of the state session `ResponderSentMsg4`, i.e. the responder. To derive these statements on `k_ab` in the initiator and responder state sessions, we must implement the usage predicate for AEAD encryption in `OYRS.Messages` and the event predicate in `OYRS.Sessions`.

Events The first three events are adapted to include the identity of the server, denoted by `srv`:

```
    let event_initiate (cid:bytes) (a b srv:principal) (n_a:bytes) : event =
        ("initiate",[cid;(string_to_bytes a);(string_to_bytes b);(string_to_bytes srv);n_a])
    let event_request_key (cid:bytes) (a b srv:principal) (n_b:bytes) : event =
        ("req_key",[cid;(string_to_bytes a);(string_to_bytes b);(string_to_bytes srv);n_b])
    let event_send_key (cid:bytes) (a b srv:principal) (n_a n_b k_ab:bytes) : event =
        ("send_key",[cid;(string_to_bytes a);(string_to_bytes b);(string_to_bytes srv);n_a;n_b
;k_ab])
```

The modified events *forward key* and *receive key* also contain the server's identity, but additionally include the respective challenges `n_a` of the initiator and `n_b` of the responder:

```
    let event_forward_key (cid:bytes) (a b srv:principal) (n_b k_ab:bytes) : event =
        ("fwd_key",[cid;(string_to_bytes a);(string_to_bytes b);(string_to_bytes srv);n_b;k_ab
])
    let event_recv_key (cid:bytes) (a b srv:principal) (n_a k_ab:bytes) : event =
        ("recv_key",[cid;(string_to_bytes a);(string_to_bytes b);(string_to_bytes srv);n_a;
k_ab])
```

We show later that this is required to proof mutual authentication using relations of certain events on the trace. The challenges have the purpose of assuring each user of the timeliness of the conversation and therefore play an important role in authentication. The server, on the other hand, is the central component in the authentication process and must hence also be included.

Protocol Specific Usage and Trace Predicates As already pointed out, we must adapt some of the usage and trace predicates in `OYRS.Messages` and `OYRS.Sessions`. The AEAD usage predicate is important to pass on statements about values protected by encryption to the receiver of the respective message. We adapt it to pass on statements about the label of the key between the third step executed by the server, and the fourth and fifth step executed by initiator and responder, as well as statements about events that occurred prior to the creation of the respective ciphertext:

```
    let can_aead_encrypt i s k ev ad =
        exists p srv. get_label oyrs_key_usages k == readers [P p; P srv] /\
        (match _parse_encval ev with
        | Success (EncMsg1 cid a b n_a) ->
            did_event_occur_before i a (event_initiate cid a b srv n_a)
        | Success (EncMsg2 cid a b n_b) ->
            did_event_occur_before i b (event_request_key cid a b srv n_b)
        | Success (EncMsg3_I n_a b k_ab) ->
            was_rand_generated_before i k_ab (readers [P srv; P p; P b]) (aead_usage "sk_i_r")
            /\ (exists cid n_b.
            did_event_occur_before i srv (event_send_key cid p b srv n_a n_b k_ab))
        | Success (EncMsg3_R n_b a k_ab) ->
            was_rand_generated_before i k_ab (readers [P srv; P a; P p]) (aead_usage "sk_i_r")
            /\ (exists cid n_a.
            did_event_occur_before i srv (event_send_key cid a p srv n_a n_b k_ab))
        | _ -> False)
    let oyrs_aead_pred i s k b ad =
        forall (t:string{bytes_to_string ad = Success t}). can_aead_encrypt i s k (|t,b|) ad
```

The encryption predicate is denoted `oyrs_aead_pred` and depends on an internal helper predicate `can_aead_encrypt`. The parameters `i`, `s`, `k`, `b`, and `ad`, are respectively the trace index at which the predicate holds, the usage string of the key used for encryption, the encryption key, the serialized but unencrypted encryption value, and the associated data of the AEAD ciphertext. We remember that the tag identifying an encryption value is not included in the serialized value and thus discarded before encryption. However, the parse function requires the tag to infer the structure of the serialized value such that it can correctly recreate it. The associated data `ad` is hence used to associate a ciphertext with the context in which it originally appeared. In particular, `ad` is the serialized tag of the corresponding serialized encryption value. We thus define `oyrs_aead_pred` for a serialized encryption value `b` and associated data `ad` such that `can_aead_encrypt` must hold for the tagged serialized encryption value `(|t,b|)`, where `t` is the tag resulting from deserializing `ad`. In `can_aead_encrypt`, we require that there are principals `p` and `srv` such that the key used for encryption, i.e., the long-term key of `p` and `srv`, has the label `readers [P p; P srv]`. Furthermore, we require that if parsing `ev` results in `EncMsg1 cid a b n_a` from the first message, then the initiator `a` has triggered `event_initiate cid a b srv n_a` before encrypting `ev` at trace index `i`. In case of `EncMsg2 cid a b n_b` from the second message, the responder `b` has triggered `event_request_key cid a b srv n_b` before `i`. For `EncMsg3_I` and `EncMsg3_R` from the third message, we have two similar conditions. The first condition ensures that the session key `k_ab` was generated before trace index `i` with usage `aead_usage "sk_i_r"` and, respectively, with label `readers [P srv; P p; P b]` or `readers [P srv; P a; P p]`, depending on the key peer `p` of `k` from the server's point of view. This essentially means that `k_ab` is dedicated to be used in AEAD encryption and only the server, the initiator and the responder should have access to it. The second condition ensures that the server triggered, respectively, `event_send_key cid p b srv n_a n_b k_ab` or `event_send_key cid a p srv n_a n_b k_ab` prior to the encryption.

With the event predicate in `OYRS.Sessions`, we specify the conditions to be met before a principal can trigger a certain event. Combined with the usage predicates for cryptographic primitives, principals can thus infer information about the state of other principals that trigger events and about the data associated with these events.

```
    let epred idx s e =
        match e with
        | ("initiate",_) | ("req_key",_) -> True
        | ("send_key",[cid;a_bs;b_bs;s_bs;n_a;n_b;k_ab]) -> (
            match (bytes_to_string a_bs, bytes_to_string b_bs, bytes_to_string s_bs) with
            | (Success a, Success b, Success srv) ->
                srv = s /\
                (did_event_occur_before idx a (event_initiate cid a b srv n_a) /\
                did_event_occur_before idx b (event_request_key cid a b srv n_b) \/
                corrupt_id idx (P a) \/ corrupt_id idx (P b) \/ corrupt_id idx (P srv)) /\
                was_rand_generated_before idx k_ab (readers [P srv; P a; P b]) (aead_usage "
sk_i_r")
            | _ -> False
        )
        | ("fwd_key",[cid;a_bs;b_bs;s_bs;n_b;k_ab]) -> (
            match (bytes_to_string a_bs, bytes_to_string b_bs, bytes_to_string s_bs) with
            | (Success a, Success b, Success srv) ->
                b = s /\
                (exists cid' n_a'.
                did_event_occur_before idx srv (event_send_key cid' a' b srv n_a n_b k_ab)) \/
                corrupt_id idx (P a) \/ corrupt_id idx (P b) \/ corrupt_id idx (P srv)
            | _ -> False
        )
        | ("recv_key",[cid;a_bs;b_bs;s_bs;n_a;k_ab]) -> (
            match (bytes_to_string a_bs, bytes_to_string b_bs, bytes_to_string s_bs) with
            | (Success a, Success b, Success srv) ->
                a = s /\
                (exists cid n_b.
                did_event_occur_before idx srv (event_send_key cid a b srv n_a n_b k_ab)) \/
                corrupt_id idx (P a) \/ corrupt_id idx (P b) \/ corrupt_id idx (P srv)
            | _ -> False
        )
        | _ -> False
```

For the *initiate* and *request key* events, no constraints on the associated data are specified. To trigger *send key*, the triggering principal `s` must play the role of the authentication server `srv` in the corresponding protocol run. Moreover, it is required that the initiator `a` has previously triggered `event_initiate cid a b srv n_a` and the responder `b` has previously triggered `event_request_key cid a b srv n_b`, or at least one of the principals involved in the protocol is corrupted. The third constraint concerns the session key `k_ab` and corresponds to the statements about `k_ab` in the AEAD usage predicate for `EncMsg3_I` and `EncMsg3_R`. The *forward key* event can only be triggered by the responder `b`. Furthermore, there must be a previous event `event_send_key cid' a' b srv n_a n_b k_ab` with arbitrary conversation identifier `cid'` and initiator `a'` triggered by the server, or at least one of the principals `a`, `b` or `srv` is corrupt. Similarly, *receive key* is restricted to the initiator role and also requires a previous *send key* event by the server, but for arbitrary `cid'` and `b'`, in case all principals in the protocol are honest.

Protocol Steps Finally, the protocol steps are adapted to account for the changed message structure and events. We must ensure that the verification of the protocol steps still succeeds with the refined encryption and event predicates. The third step performed by the server is extended with the additional check comparing the conversation identifier and user identities in the unencryprted and encrypted parts. As indicated above, we also use the associated data of AEAD ciphertexts to store the tags of underlying encryption values. This considerably facilitates proving the desired security properties via the AEAD usage predicate by uniquely identifying the encryption value referenced by a particular instance of the predicate, since the structure of the encryption values is symmetric in the second and third message. The symmetry however only holds for message parts encrypted or decrypted with different keys, such that the context can be inferred easily in practice. Therefore, the security properties we will prove for the improved protocol also hold when simple symmetric encryption is used instead of AEAD.

Correctness of the Improved Model

The correctness of the improved Otway-Rees model must be verified in a similar way as that of the original model. With the improved model, we have introduced several refinements, such as refinements in the usage predicate for AEAD on the encrypted data, in the event predicate on the data we associate with events, or in the `valid_session` predicate in `OYRS.Sessions` on the data stored in state sessions. These refinements assist in formulating and proving the remaining security goals from Section 4.3.4. To establish the soundness of the refined model, we must once again perform a type check of the modified F* modules. In order for the type check to be successful, F* may in some cases need manual assistance with the type derivation of expressions that are subject to refinements such as those described above. We must particularly prove that the ciphertexts encrypted, events triggered and data stored during the improved protocol comply with the refined encryption and event predicates. Later, we can use these predicates to establish remaining security properties.

One such refinement is a statement added in `OYRS.Sessions.valid_session (i:nat) (p:principal) ( si vi:nat) (st:session_st)` about the label of the session key `k_ab` stored in the initiator's final state session `InitiatorRecvedMsg4 srv b k_ab`:

```
(corrupt_id i (P p) \/ corrupt_id i (P srv) \/ corrupt_id i (P b)
      \/ is_labeled i k_ab (readers [P srv; P p; P b]))
```

This statement is passed on by the server in the third step when it encrypts the value EncMsg3_I n_a b k_ab with the long-term key k_as shared with the initiator, who is p in this case. For the initiator to be able derive this statement when it receives the corresponding ciphertext in the fifth step, we must show that the AEAD usage predicate holds at the time the encryption is performed. In case of EncMsg3_I n_a b k_ab, the encryption predicate consists essentially of the following statement:

```
          exists p srv. get_label oyrs_key_usages k == readers [P p; P srv]
    /\ was_rand_generated_before i k_ab (readers [P srv; P p; P b]) (aead_usage "sk_i_r")
```

The first part of the statement says that the key k used for encryption can be read by arbitrary state sessions of principals p and srv. Since k = k_as, p must be the initiator a and srv the server. The second part implies that k_ab is labeled readers [P srv; P p; P b]. When the initiator receives and decrypts the ciphertext in the fifth step, it knows from the aead_dec_lemma that either the key used for encryption (k_as) is publishable or that the encryption predicate holds true. In DY*, we have that if a value of type bytes labeled with a readers label is publishable, it means that one of the readers is corrupted. The initiator knows the label of k_as and can thus infer that the encryption predicate holds if neither P a nor P srv are corrupted. Further, it knows that if the encryption predicate holds true, then k_ab must be labeled readers [P srv; P p; P b] with p = a. This leads to the statement about the label of k_ab that we added in valid_session. F* is not capable of performing the entire proof chain automatically, so we need to guide it through the proof with useful assertions at places where manual help is required. In this manner, we are able to prove all the refinements in the modified Otway-Rees model, so that the model passes the type check.

Regarding trace verification, we can simply reuse the scheduler function that simulates a benign attacker in the original protocol, since the protocol API is unchanged. The resulting trace is depicted in Appendix A.4. The trace of the modified protocol slightly differs from that of the original protocol but results in the same outcome – the key chosen by the server is stored in the respective state sessions of initiator and responder. Besides the benign environment, we can also simulate the three attacks presented in Section 4.3.5 in the improved model. The output in Appendix A.4 shows that all attacks fail when executed in the improved model, which suggests that the changes suffice to prevent them. It is howbeit still left to show that the protocol's intended security goals are also achieved in the presence of an arbitrary attacker.

Security Proof

We show that the new Otway-Rees protocol satisfies all secrecy and authentication goals defined in Section 4.3.4. Therefore, we extend the module OYRS.SecurityProps with security properties for these goals in terms of the improved model.

The original protocol did not guarantee that the initiator and responder receive the key generated by the authentication server as required by the secrecy objective S2. We have described and modeled attacks in the course of this thesis that violate this exact property. In order to ensure it in the improved model, we have to show that the key stored in the final state sessions of initiator and responder is secret, similarly as in the secrecy proof from the server's point of view. Two new security lemmas session_key_stored_in_initiator_state_is_secret and session_key_stored_in_responder_state_is_secret are hence defined. The property that both lemmas ensure is the same property as that ensured by the previously defined server key secrecy lemma but from the perspective of the initiator and responder. This works because of the discussed refinements in the encryption and event predicates, as well as in the valid_session predicate. Using the example of the initiator, we have explained how a statement about the label of k_ab stored in the initiator's final state session is derived from a message originally coming from the server. The derivation for k_ab in the responder's state session is similar. Unlike in the proof from the server's perspective, we can only be sure about the label of the session key in cases where none of the principals involved in the key exchange is corrupted. Since we can only prove the key secrecy for non-corrupted readers using the secrecy lemma anyways, the result is the same in both cases. We are able to verify both lemmas via F* and can thus show that the session key in the improved Otway-Rees protocol remains secret if the principals involved in the key exchange are honest.

Mutual authentication was defined in terms of objectives A1 and A2. A1 requires that users sharing a session key have fulfilled their designated roles in the protocol, and A2 requires that they trust each other about their identities. We can prove both objectives in one lemma but need to differentiate between initiator and responder authentication. If both properties hold simultaneously, we have mutual authentication of initiator and responder. We define authentication of the initiator to the responder in a lemma initiator_authentication:

```
val initiator_authentication: i:nat ->
LCrypto unit (pki oyrs_preds)
(requires (fun t0 -> i < trace_len t0))
(ensures (fun t0 _ t1 -> forall cid a b srv n_b k_ab.
did_event_occur_at i b (event_forward_key cid a b srv n_b k_ab)
==> ((exists cid' n_a. did_event_occur_before i a (event_initiate cid' a b srv n_a)
    /\ did_event_occur_before i b (event_request_key cid' a b srv n_b))
    \/ corrupt_id i (P a) \/ corrupt_id i (P b) \/ corrupt_id i (P srv))))
```

The natural number i must be a valid trace index. The lemma ensures that for arbitrary conversation identifier cid, initiator a, responder b, server srv, responder challenge n_b, and session key k_ab, if the responder has triggered event_forward_key cid a b srv n_b k_ab at trace index i, then there must be corresponding events event_initiate cid' a b srv n_a and event_request_key cid' a b srv n_b with arbitrary cid' and n_a that were triggered, respectively, by the initiator and by the responder prior to the *forward key* event unless at least one of the principals involved in the run has been corrupted. We do not know whether the conversation identifiers cid and cid' are the same. However, we know that the conversation identifier matches in the events *initiate* triggered by the initiator and *request key* triggered by the responder and that the responder challenge n_b matches in the events *request key* and *forward key* both triggered by the responder. That is, if the responder b receives its correct challenge n_b from the server srv and forwards the session key k_ab to a, then a has previously initiated a conversation in which the responder has chosen n_b. In other words, this means that if a user b as responder accepts a key k_ab with another user a, then there is a corresponding protocol run initiated by a. This is exactly the authentication goal A1 from the responder's perspective, who requires the initiator to authenticate to him. Moreover, the correspondence of a and n_b among the events *initiate*, *request key* and *forward key* establishes the responder's trust in the identity a of the initiator, which was required by the authentication objective A2. The initiator_authentication property therefore perfectly captures the authentication goals intended for the protocol from the responder's point of view. We prove that initiator authentication holds for the improved Otway-Rees protocol by verifying the lemma in the context of the modified model in F*.

Having established initiator authentication, we can express responder authentication with a similar lemma but different events:

```
val responder_authentication: i:nat ->
LCrypto unit (pki oyrs_preds)
(requires (fun t0 -> i < trace_len t0))
(ensures (fun t0 _ t1 -> forall cid a b srv n_a k_ab.
did_event_occur_at i a (event_recv_key cid a b srv n_a k_ab)
==> ((exists cid' n_b. did_event_occur_before i b (event_request_key cid' a b srv n_b)
    /\ did_event_occur_before i a (event_initiate cid' a b srv n_a))
    \/ corrupt_id i (P a) \/ corrupt_id i (P b) \/ corrupt_id i (P srv))))
```

The property states that if an arbitrary initiator `a` triggers `event_recv_key cid a b srv n_a k_ab` for arbitrary `cid`, `b`, `srv`, `n_a`, and `k_ab`, at trace index `i`, then there are prior events `event_request_key cid' a b srv n_b` triggered by the responder `b` and `event_initiate cid' a b srv n_a` triggered by `a`. Again, we cannot say anything about the conversation identifiers `cid` and `cid'` themselves, but we have the correspondence of `cid'` in the events *initiate* and *request key*, and of `n_a` in the events *initiate* and *receive key*, which implies that there is a relation between the events *request key* of the responder and *receive key* of the initiator. The precise implication for the initiator is the following: the user `b` from whom he receives the key `k_ab` is actually the responder who requested the key from the server prior to that in a conversation initiated by `a`, in which it chose the challenge `n_a`. The `responder_authentication` lemma thus captures the authentication goals A1 and A2 from the initiator's perspective and is – similarly as the initiator authentication property – verified in the improved model in F*. Together with the verified initiator authentication property, we fully capture the authentication objectives for the Otway-Rees protocol and conclude that our improved Otway-Rees protocol achieves mutual authentication.

F* is able to verify both authentication lemmas automatically based on the information we provide in the model of the improved protocol. We used the event predicate to specify for each event what events must have preceded it. The *initiate* and *request key* events of initiator and responder are not subject to any restrictions, the *send key* event is triggered by the server after the initiator has started a conversation in which the responder has requested a key, and the final events of initiator and responder again depend on the *send key* event. The event predicate thus imposes relations on user events before and after key distribution by the server. The authentication lemmas defined above rely on these relations. Because the event predicate is not capable of passing statements between protocol steps, this is done via the AEAD predicate. The soundness of the improved model – including all usage and trace predicates – has already been proved in F*, so the soundness of the authentication lemmas follows.

4.3.7 Discussion

So far, we have elaborated essential properties of the Otway-Rees protocol, developed a model of the protocol in DY*, and shown that the protocol does not satisfy all the intended security goals within the model. We then discussed possible attacks, extended our model accordingly, and proposed an improved version of the protocol preventing the attacks and satisfying strong security goals. Finally, we have modeled the improved Otway-Rees protocol in DY* and proven its security in the new model. Section 3.3.1 summarized existing literature on the Otway-Rees protocol including different approaches to analyse the protocol regarding its security. In this section, we discuss the results of our analysis in the context of other research on the Otway-Rees protocol with the aim to explain possible inconsistencies and differences in the results.

The first formal analysis of the Otway-Rees protocol was performed by Burrows et al. [11] with the BAN logic. With their analysis, they come to the conclusion that the protocol is well designed and establishes the users' trust in the secrecy of the exchanged key. The BAN analysis argues in terms of *believes* rather than *knowledge* – the above statement should therefore not be misunderstood as an actual claim of secrecy of the exchanged key. Howbeit, the attacks provided in Section 4.3.5 show that the conclusion of the BAN authors is nevertheless dangerous, since the analysis is based on a passive attacker setting. An active attacker can bring the users to accept a key consisting only of publicly known terms, which has neither been chosen by the server nor originated from a message emitted by the server. Therefore, the key that the users accept does not necessarily provide the guarantees assumed in the BAN analysis. Even if we disregard the type flaw attacks, which heavily depend on implementation details of the protocol, there is still the impersonation attack in which the final belief of the initiator is to share a secret key with a user that has not been involved in the corresponding protocol run at all. In fact, the server does not provide any guarantee regarding the identity of the respective key peer of the key it distributes to the users. The users thus have no way to know with whom they actually share the key they receive from the server. Regarding authentication, the BAN authors state that the responder authenticates to the initiator via the second message in the protocol, in which it forwards an encrypted part from the initiator to the server containing the initiator's challenge. Again, if we do not consider type flaw attacks, it is true that an initiator A who initiates a protocol run with another honest user B beliefs to talk to B after completing the run. The authentication goals derived from the protocol description are however violated because – as mentioned above – B was not necessarily involved in the protocol. Burrows et al. account for the fact that the server does not verify the unencrypted part of the second message, and it is not mentioned anywhere that they assume the users to perform type checks on the exchanged key. All attacks on the Otway-Rees protocol we describe and incorporate in our model are thus also possible in the setting described by the BAN authors. The problem of the BAN analysis is on the one hand that the principals in the protocol are considered honest while there is no such thing as an active attacker that can tamper with the protocol messages, and on the other hand that the abstractions made by the idealization process preceding the analysis are dangerous.

The limitations of the BAN logic and the idealization process were first mentioned by Boyd and Mao [7], who find the impersonation attack on the original Otway-Rees protocol that we described and modeled in the course of our analysis. Based on their research, they present the concept of robust protocols whose messages can be analyzed independently of each other, and propose a fix to prevent the attack and make the protocol robust (see Section 3.3.1). However, the fix does not actually prevent the attack. The adversary can still carry out the same impersonation attack as before:

1. $A \rightarrow I(B): \quad N_{cid}, A, B, \{N_a, N_{cid}, A, B\}^s_{k_{as}}$
2. $I(B) \rightarrow AS: \quad N_{cid}, A, E, \{N_a, N_{cid}, A, B\}^s_{k_{as}}, \{N_e, N_{cid}, A, B\}^s_{k_{es}}$
3. $AS \rightarrow I(B): \quad N_{cid}, \{N_a, A, k_{ae}\}^s_{k_{as}}, \{N_e, E, k_{ae}\}^s_{k_{es}}$
4. $I(B) \rightarrow A: \quad N_{cid}, \{N_a, A, k_{ae}\}^s_{k_{as}}$

Boyd and Mao tried to obviate the attack by making the key peer identities explicit in the third message from the AS. Though, they make the mistake to include the identity of the respective long-term key owner to whom the ciphertext is destined instead. For the latter, however, the information is of no value because it is already implied by successful decryption. When A receives the fourth message from the attacker, it decrypts the ciphertext using k_{as}, sees its own identity A, and accepts. From the successful decryption, it already knows that the ciphertext was intended for it, and therefore the identity A in the encrypted part is obsolete. As a result of our analysis, we propose an improved protocol based on Boyd and Mao's robust design principle that actually prevents the attack. Our fix includes the correct key peer identities in the encrypted parts of the third message from the AS, so that the encrypted part that A receives in the last message would contain E's identity if the adversary attempted to carry out the attack. In addition, we extended the protocol step executed by the server with a check comparing the unencrypted and encrypted parts of the second message (more precisely, the concatenation N_{cid}, A, B in these parts), so that the server would already abort in case it detects inconsistencies. Nonetheless, it is always better to eliminate attack vectors on the message structure level, rather than on the implementation level, in order to make the properties of the protocol as clear as possible and enforce secure implementations of the protocol, rather than leaving the responsibility to the implementor. The problem with the improved Otway-Rees protocol presented by Boyd and Mao is hence not to be found in the robust principle itself, which we have shown to obviate the attack and ease the analysis, but instead in their implementation of the principle in the instance of the Otway-Rees protocol, i.e., that they still miss to make the key peers explicit in the third message.

Another improved version of the Otway-Rees protocol was proposed by Chen [12] and proven secure in the SVO logic. The protocol of Chen was explicitly designed to obviate the impersonation attack of Boyd and Mao and it also prevents the other attacks described in this thesis. However, Chen's protocol comes with stronger security goals than those that we derived from the protocol description by Otway and Rees. The stronger security properties particularly require an exchange of ciphertexts encrypted under the session key k_{ab} to assure the users that the respective key peer knows the key and also trusts it to be secret. In our analysis, we demonstrated that this exchange is not necessary to achieve the original security goals of the protocol that we divided into four objectives regarding secrecy and authentication in Section 4.3.4. The users get this assurance as they send and receive messages in our improved protocol without deviating too much from the original structure. Chen's protocol, on the other hand, heavily deviates from the original structure and includes an additional fifth message to achieve its stronger security goals. We have also mentioned the use of EAT to investigate the security of the Otway-Rees protocol [37]. Multiple improved versions of the protocol have been proposed based on EAT. They do not deviate as much from the original protocol and come with the same strong security implications as Chen's improved version. Howbeit, they still include an additional ciphertext encrypted with k_{ab} to achieve this level of security, which is not required to ensure that the key remains secret or that the users mutually authenticate. With only little improvements to the Otway-Rees protocol, we were already able to ensure that the users only accept a session key that originates from the server. The secrecy of the key then depends on the behavior of the users as well as the server. As long as none of the principals leaks the key to the adversary, it remains secret. Moreover, our improvements make sure that the attacker cannot impersonate an honest party to break authentication anymore. We hence claim that our improvements are perfectly sufficient to fix the vulnerabilities of the Otway-Rees protocol without moving too far from the original structure, which results in a totally new protocol.

Contrarily to the other approaches, Backes [4] provides a detailed security proof of the Otway-Rees protocol based on an abstract cryptographic library. Their proof stands under the assumption that the adversary cannot carry out the type flaw attacks that are possible in our DY* model. As we already mentioned, the type flaw attacks are heavily dependent on implementation details that are out of scope of the work of Otway and Rees, who simply propose a blueprint of the protocol messages and provide a description of the actions performed by the principals involved in the protocol. Everything beyond that, e.g., the length of the long-term keys or the session key, depends on factors that cannot be determined in advance. The assumption made by Backes is thus reasonable. Though, we again emphasize the importance of preventing attacks by design, rather than implementation, which is why we considered type flaws in our model and proposed an alternative message structure that rules out attacks of this kind. Besides the assumption regarding type flaw attacks, Backes also assumes that the server performs the additional check on the second message from the responder, which is however not mentioned by Otway and Rees in their description of the actions performed by the server. Since Otway and Rees describe the actions performed by the principals clearly, we argue that the missing check imposes a flaw that is part of the original specification of the Otway-Rees protocol and therefore definitely relevant. The resulting threat is also easily resolved by design by making the key peers explicit in the third message from the server. Backes uses the absence of this vulnerability to prove their consistency property, which is concerned with the consistency of the users' views on the respective key peer of the accepted session key and intuitively results in the secrecy of the session key as long as the long-term keys remain secret. In our model, the original protocol does not fulfill this property, which is also demonstrated by the impersonation attack where A's view on the key peer is inconsistent with that of E (the adversary).

Whether an analysis of the Otway-Rees protocol concludes that it is secure and meets its secrecy and authentication objectives, or that it is insecure and vulnerable to various types of attacks, depends primarily on the assumptions made in the approach used for the analysis and the defined security goals. If we assume a robust and secure implementation of the protocol that performs sufficient checks on the data in messages, then there is no way for an adversary to break security without compromising long-term keys. Regarding authentication, we must trust the authentication server to distribute the session key to the correct users, and based on that, get timeliness guarantees. On the other hand, if we build our assumptions on the original protocol specification by Otway and Rees, we have that the server does not sufficiently check the user identities in the second message, which imposes a security vulnerability and enables an attacker to break authentication. Also, if the users do not further check whether the key they receive – supposedly from the server – has the properties of a key generated by the server, an adversary can possibly replay the ciphertexts from the first two messages as key certificates from the server to the users to make them accept a session key which it can derive from its own knowledge. Besides the assumptions on which an analysis is based and the security goals defined for the analysed protocol, the result of an analysis also depends on the abstractions made in the used approach. For example, the BAN logic abstracts very aggressively from the original form of a protocol to bring it into a form that serves as starting point for the analysis. Boyd and Mao elaborate how the abstractions made by the BAN analysis make it impossible to detect certain flaws in protocols on the example of the Otway-Rees protocol. In our analysis, we have considered all the possible implementation flaws that the Otway-Rees protocol has and thus also modeled all the resulting attacks. We further proposed an improved protocol with small deviations in the message structure and in the behavior of the authentication server to make the Otway-Rees protocol secure against the modeled attacks and also in the presence of an arbitrary attacker. The security of the improved protocol still depends on the security of the long-term keys; if an adversary is able to compromise them, it completely breaks the security of the protocol.

4.4 Yahalom

We continue our analysis with the Yahalom protocol, which is an authentication protocol formalized by Burrows et al. [11] in their paper on the BAN belief logic. The protocol is similar to Otway-Rees insofar as both protocols rely on symmetric keys pre-shared with an authentication server that performs a key exchange between two users. However, the Yahalom protocol comes with some subtle properties and, as our analysis will show, achieves its designated secrecy and authentication goals that we derive from the protocol specification and from the assumptions and goals stated by Burrows et al. in their BAN analysis of the protocol. With our security proof, we also argue against research that labels the protocol flawed and insecure and show that it fulfills its original purpose based on what the BAN authors provide in their research. Our analysis of Yahalom is based on the message structure given in Section 3.3.2.

4.4.1 Properties and Goals

Again, we start by capturing essential properties and goals of the protocol required for the development of a fitting model in DY*.

Principal Roles

In the Yahalom protocol, we again have two users A and B that want to exchange a key and a trusted authentication server AS that is in charge of key distribution. A is the initiator who starts a protocol run via the first message to B. B is the responder who receives the first message from A and requests a key from the AS. The AS responds by generating and then distributing a key for secret communication between A and B. The roles reverse for the last two messages when the AS first sends the key to A, who then forwards it to B.

Initial Knowledge

The initial knowledge of the principals is exactly the same as in the Otway-Rees protocol. Each of the users shares a long-term key with the AS and therefore also knows its identity. The initiator additionally knows B so that it can initiate the protocol. The AS stores the long-term keys and identities of both users.

Generated Nonces

Each user generates one challenge for the authentication server. The challenge N_a of the initiator is public, while N_b generated by the responder only appears encrypted under the long-term keys k_{bs} and k_{as}, as well as under the session key k_{ab} and therefore has the properties of a shared secret. Despite the use of N_b as shared secret, both challenges have the same purpose as in the Otway-Rees protocol – to provide the users with timeliness guarantees regarding the key exchange.

Exchanged Keys or Secrets

Similarly as in the Otway-Rees protocol, the authentication server generates and distributes the session key as a response to the second message from the responder. The server protects k_{ab} by encrypting it under k_{as} for the initiator and k_{bs} for the responder, so that only initiator and responder can obtain it given the long-term keys are not corrupted. The protocol shall also ensure that the users actually receive the key from the server, which they trust. Besides the session key, the Yahalom protocol also results in the exchange of a shared secret – the nonce N_b generated by the responder. The nonce is intended as challenge for the server who delegates this task to the initiator by encrypting it under k_{as} in the certificate for the session key together with the session key k_{ab} itself. In the fourth message, the initiator then encrypts it under the session key k_{ab} to assure the responder that the communication is timely. As a result, N_b is a shared secret between the users.

Authentication Goals

The Yahalom protocol strives to mutually and timely authenticate two users A and B in a similar way as the Otway-Rees protocol. Again, a session key k_{ab} accepted by the users shall result from a recent successful protocol run with initiator A and responder B, and both users shall be assured that they recently talked to each other. The fact that A already uses k_{ab} during the protocol, however, results in one additional implication compared to Otway-Rees: at the end of the protocol, B already learns that A accepted the same key as itself and therefore has the same beliefs about the key as B. All assurances may not hold in case some principals act maliciously.

4.4.2 Modeling the Protocol in DY*

We now develop a model of the Yahalom protocol in DY*, based on its message structure and the previously discussed properties and goals. We follow a similar approach to the model of the Otway-Rees protocol.

Messages

```
/// Fromat of encrypted message parts
noeq type encval =
    | EncMsg2: a:principal -> n_a:bytes -> n_b:bytes -> encval
    | EncMsg3_I: b:principal -> k_ab:bytes -> n_a:bytes -> n_b:bytes -> encval
    | EncMsg3_R: a:principal -> k_ab:bytes -> encval
    | EncMsg4: n_b:bytes -> encval

noeq type message (i:nat) =
    | Msg1: a:principal -> n_a:bytes -> message i
    | Msg2: b:principal -> ev_b:msg i public -> message i
    | Msg3: ev_a:msg i public -> ev_b:msg i public -> message i
    | Msg4: ev3_b:msg i public -> ev4_b:msg i public -> message i
```

We again have a module `YLM.Messages` with types `encval` representing the encrypted parts of messages and `message (i:nat)` for the messages themselves. `EncMsg2` is the responder's encryption in the second message. `EncMsg3_I` and `EncMsg3_R` are the encryptions performed by the server to distribute the session key `k_ab` to the users. `EncMsg4` is encrypted by the initiator in the last message to assure the responder of the timeliness of the conversation. The messages are constructed from unencrypted terms, such as the identifiers of the two users required by the server, and the ciphertexts resulting from the encrypted parts. The serialized and encrypted ciphertexts in the constructors of `message i` are denoted by the type `msg i public`, so that the constructors only accept valid and publishable values of `bytes`. We no longer need the wrapper types for tagging encryption values from the Otway-Rees model because we add the tag directly to the serialized value in the Yahalom model.

```
val serialize_encval: i:nat -> ev:encval -> l:label{valid_encval i ev l} -> sev:(msg i l)
val parse_encval: #i:nat -> #l:label -> sev:(msg i l) -> r:(result encval)
    {
        match r with
        | Success ev -> valid_encval i ev l
        | Error _ -> True
    }

val serialize_msg: i:nat -> m:(message i){valid_message i m} -> msg i public
val parse_msg: #i:nat -> sm:(msg i public) -> r:(result (message i))
{
    match r with
    | Success m -> valid_message i m
    | Error _ -> True
}
```

This also eases the serialization and parsing process. `serialize_encval` outputs values of type `msg i l` coming from encryption values `ev` that are valid with respect to `valid_encval i ev l`. Similarly, `serialize_msg` outputs values of type `msg i public` given valid messages. The parse functions get serialized values with an implicit tag and return either valid encryption values or messages, or an error if parsing fails. Besides the differences in how tags are handled, serialization and parsing work exactly as in the Otway-Rees model, and `valid_encval i ev l` again ensures that the terms in an encryption value `ev` are valid at trace index `i` and that their label flows to `l`. `valid_message` does the same for messages and the `public` label.

State

```
noeq type session_st =
    (* Auth server session for secret keys shared with principals *)
    | AuthServerSession: p:principal -> k_ps:bytes -> session_st
    (* Principal secret key session with auth server *)
```

```
        | PKeySession: srv:principal -> k_ps:bytes -> session_st
        (* Protocol states *)
        | InitiatorSentMsg1: b:principal -> n_a:bytes -> session_st
        | ResponderSentMsg2: a:principal -> srv:principal -> n_b:bytes -> session_st
        | AuthServerSentMsg3: a:principal -> b:principal -> k_ab:bytes -> session_st
        | InitiatorSentMsg4: b:principal -> srv:principal -> k_ab:bytes -> session_st
        | ResponderRecvedMsg4: a:principal -> srv:principal -> k_ab:bytes -> session_st
```

The protocol specific state sessions of the Yahalom protocol are modeled in the `session_st` type in `YLM.Sessions`. The three state sessions for storing the principals' initial knowledge from the Otway-Rees model have been replaced with only two state sessions for the long-term key bindings in the Yahalom model. The first state session `AuthServerSession p k_ps` represents a long-term key session from the server's point of view, in which it stores the respective principal `p` with whom it shares a key, and the key `k_ps` itself. Likewise, `PKeySession` defines a user key session, where a user stores server identity and long-term key. The other state sessions are protocol state sessions and thus only store data associated with a particular protocol run. The naming of the state sessions has also been adapted from the Otway-Rees model. The name of a state session constructor indicates who is supposed to use it and in which context it is to be used. For example, `InitiatorSentMsg1 b n_a` is used by the initiator after sending the first message to store the identity of the responder `b` – with whom it wants to exchange a key –, and its challenge `n_a`, which it verifies when receiving the third message from the server. The responder similarly stores the name of the initiator `a` and its own nonce `n_b` in `ResponderSentMsg2`. However, it also needs the server's identity `srv`, because it must be able to express via `n_b`'s label that the server can read the nonce. The remaining three states all store the session key `k_ab` and the peers of the key from the perspective of the state session owner.

```
    let is_lt_key i b p srv = is_aead_key ylm_global_usage i b (readers [P p; P srv]) "YLM.
lt_key"
    let is_comm_key i b srv p q = is_aead_key ylm_global_usage i b (readers [P srv; P p; P q])
 "YLM.comm_key"

    let valid_session (i:nat) (p:principal) (si vi:nat) (st:session_st) =
        match st with
        | AuthServerSession pri k_pri_srv ->
          is_msg i k_pri_srv (readers [P p]) /\ is_lt_key i k_pri_srv pri p
        | PKeySession srv k_ps -> is_lt_key i k_ps p srv
        | InitiatorSentMsg1 b n_a -> is_labeled i n_a public
        | ResponderSentMsg2 a srv n_b -> is_labeled i n_b (readers [P p; P a; P srv])
        | AuthServerSentMsg3 a b k_ab -> is_comm_key i k_ab p a b
        | InitiatorSentMsg4 b srv k_ab ->
          is_msg i k_ab (readers [P p]) /\
          (is_labeled i k_ab (readers [P srv; P p; P b]) \/ corrupt_id i (P srv) \/
corrupt_id i (P p))
        | ResponderRecvedMsg4 a srv k_ab ->
          is_msg i k_ab (readers [P p]) /\
          (is_labeled i k_ab (readers [P srv; P a; P p]) \/ corrupt_id i (P srv) \/
corrupt_id i (P p))
```

Serialization and parsing are completely adapted from the Otway-Rees model. The `valid_session` predicate is used to refine the terms stored in the state sessions, and like in the improved Otway-Rees model, serves as basis for the secrecy proof of the session key. We also added predicates `is_lt_key` and `is_comm_key` to express that a term is a long-term key or a communication (session) key. The states relevant for the secrecy proof are again the final state sessions of the principals, i.e., `AuthServerSentMsg3`, `InitiatorSentMsg4`, and `ResponderRecvedMsg4`. From the final state session of the server, we get that `k_ab` is a communication key for AEAD and labeled with `readers [P p; P a; P b]` (with server `p`). In the final state sessions of initiator and responder, the label of `k_ab` depends on the honesty of the readers of the respective long-term key used to transmit it, which does howbeit not affect the secrecy proof.

Events

```
let event_initiate (a b srv:principal) (n_a:bytes) =
    ("initiate",[string_to_bytes a;string_to_bytes b;string_to_bytes srv;n_a])
let event_req_key (a b srv:principal) (n_a n_b:bytes) =
    ("req_key",[string_to_bytes a;string_to_bytes b;string_to_bytes srv;n_a;n_b])
let event_send_key (a b srv:principal) (n_a n_b k_ab:bytes) =
    ("send_key",[string_to_bytes a;string_to_bytes b;string_to_bytes srv;n_a;n_b;k_ab])
let event_fwd_key (a b srv:principal) (n_a n_b k_ab:bytes) =
    ("fwd_key",[string_to_bytes a;string_to_bytes b;string_to_bytes srv;n_a;n_b;k_ab])
let event_recv_key (a b srv:principal) (n_b k_ab:bytes) =
    ("recv_key",[string_to_bytes a;string_to_bytes b;string_to_bytes srv;n_b;k_ab])
```

We define the same five events for the Yahalom protocol that we previously defined for Otway-Rees. The main differences are in the associated data, which consists of different terms due to the different message structure, and in the principals triggering the last two events. The *initiate* event is still triggered by the initiator before it sends the first message to `b`. Afterwards, the responder signals that it requests the key from the server and the server signals that it distributes the key to the users. The *forward key* event is however now triggered by the initiator, who receives the session key directly from the server, while the responder triggers *receive key* when it indirectly gets the key from the initiator. In each event, we associate all data that is relevant to the current protocol run at the time of the event.

Protocol Specific Usage and Trace Predicates

While we specifically omitted the relevant usage and trace predicates for the Otway-Rees protocol because of several attacks violating its security objectives, we explicitly require them in the Yahalom model. We want to prove the security of the Yahalom protocol and therefore require statements regarding the label of the session key in all state sessions in which it is stored as a result of the protocol. Particularly, we want to derive the label of the session key in the state sessions `InitiatorSentMsg4` and `ResponderRecvedMsg4`, as required by the `valid_session` predicate in `OYRS.Sessions`. To be able to serialize and store state sessions, they must fulfill this predicate, so we must derive these statements in the respective protocol steps where they will be stored. Therefore, we again need the usage predicate for AEAD to pass statements about properties of encrypted terms – such as the session key – between message sender and receiver.

```
let can_aead_encrypt i s k sev ad =
    match parse_encval_ sev with
    | Success (EncMsg2 a n_a n_b) ->
        exists b srv. get_label ylm_key_usages k == readers [P b; P srv] /\
        did_event_occur_before i b (event_req_key a b srv n_a n_b)
        | Success (EncMsg3_I b k_ab n_a n_b) ->
          exists a srv. get_label ylm_key_usages k == readers [P a; P srv] /\
          did_event_occur_before i srv (event_send_key a b srv n_a n_b k_ab)
        | Success (EncMsg3_R a k_ab) ->
          exists b srv. get_label ylm_key_usages k == readers [P b; P srv] /\
          (exists n_a n_b. did_event_occur_before i srv (event_send_key a b srv n_a n_b k_ab
))
        | Success (EncMsg4 n_b) ->
          exists a b srv n_a. did_event_occur_before i a (event_fwd_key a b srv n_a n_b k)
        | _ -> False
```

Since the tag of encryption values is included in the serialized value, we only need a single predicate in the Yahalom model. The predicate divides into four branches for the different encryption values that are encrypted in the protocol. The first encryption value is `EncMsg2 a n_a n_b`, for which we require that there are principals `b` and `srv` such that the label of the key `k` used for encryption is `readers [P b; P srv]`. This ensures that `EncMsg2` is encrypted under the long-term key shared between a responder `b` and the server `srv`. Secondly, we require that the responder triggered `event_req_key a b srv n_a n_b` prior to the encryption. The responder event associates the principals involved in the protocol run and the challenges of initiator and responder. For `EncMsg3_I b k_ab n_a n_b`, we similarly require that it is encrypted with the long-term key of initiator and server, and that the server signaled `event_send_key a b srv n_a n_b k_ab` before `i`. The server event – in addition to the terms already included in the previous responder event – also associates the just generated session key `k_ab` with the protocol run, which is important to prove the authentication property later. `EncMsg3_R a k_ab` is in turn encrypted under the long-term key of responder and server, and also requires that `event_send_key a b srv n_a n_b k_ab` was triggered before. The encryption value does however not contain the nonces `n_a` and `n_b`, so that the responder does not get the timeliness guarantees it needs from this statement alone. The last encryption value `EncMsg4 n_b` is the second part of the fourth message, which is encrypted with the session key. Therefore, we require that the initiator triggers `event_fwd_key a b srv n_a n_b k` before encrypting the value, signaling that it forwards the session key `k` to the responder and uses it to encrypt the responder challenge `n_b`, which gives the responder its timeliness guarantees. We deliberately omitted statements about the label of the session key in the AEAD predicate, since we already include them in the event predicate and wanted to avoid redundancies. The sole purpose of the AEAD predicate hence is to ensure that encryption values are encrypted under the correct key and to establish relations between ciphertexts and events. The event predicate, on the other hand, has the purpose to ensure all refinements we make regarding the data that is relevant to a protocol run, including principal identities, nonces, and the session key.

```
let epred idx s e =
    match e with
    | ("initiate",[a_bytes;b_bytes;srv_bytes;n_a]) ->
      bytes_to_string a_bytes == Success s
    | ("req_key",[a_bytes;b_bytes;srv_bytes;n_a;n_b]) -> (
      match (bytes_to_string a_bytes,
             bytes_to_string b_bytes,
             bytes_to_string srv_bytes) with
      | (Success a, Success b, Success srv) ->
        b = s /\
        is_msg idx n_a public /\
                was_rand_generated_before idx n_b (readers [P b; P a; P srv]) (nonce_usage "
YLM.nonce_b")
            | _ -> False
        )
        | ("send_key",[a_bytes;b_bytes;srv_bytes;n_a;n_b;k_ab]) -> (
            match (bytes_to_string a_bytes,
                   bytes_to_string b_bytes,
                   bytes_to_string srv_bytes) with
            | (Success a, Success b, Success srv) ->
                srv = s /\
                was_rand_generated_before idx k_ab (readers [P srv; P a; P b]) (aead_usage "
YLM.comm_key") /\
                (did_event_occur_before idx b (event_req_key a b srv n_a n_b) \/
                (corrupt_id idx (P b) \/ corrupt_id idx (P srv)) /\ is_msg idx n_b public)
            | _ -> False
        )
        | ("fwd_key",[a_bytes;b_bytes;srv_bytes;n_a;n_b;k_ab]) -> (
            match (bytes_to_string a_bytes,
                   bytes_to_string b_bytes,
                   bytes_to_string srv_bytes) with
            | (Success a, Success b, Success srv) ->
                a = s /\
                (did_event_occur_before idx srv (event_send_key a b srv n_a n_b k_ab) \/
                (corrupt_id idx (P a) \/ corrupt_id idx (P srv))
                /\ is_publishable ylm_global_usage idx n_b)
            | _ -> False
        )
        | ("recv_key",[a_bytes;b_bytes;srv_bytes;n_b;k_ab]) -> (
            match (bytes_to_string a_bytes,
                   bytes_to_string b_bytes,
                   bytes_to_string srv_bytes) with
            | (Success a, Success b, Success srv) ->
                b = s /\
                ((exists n_a n_b.
                did_event_occur_before idx srv (event_send_key a b srv n_a n_b k_ab)) \/
                corrupt_id idx (P b) \/ corrupt_id idx (P srv)) /\
                ((exists n_a.
                did_event_occur_before idx a (event_fwd_key a b srv n_a n_b k_ab)) \/
                corrupt_id idx (P b) \/ corrupt_id idx (P srv) \/ corrupt_id idx (P a))
            | _ -> False
        )
        | _ -> False
```

The event predicate of the Yahalom model is slightly more complex than that of improved Otway-Rees. This stems from the fact that the properties of the Yahalom protocol are more subtle with respect to the unusual order and structure of messages, and that we capture all properties of the protocol in the event predicate, while the encryption predicate merely imposes relations between ciphertexts and events. The only restriction of the *initiate* event is that it must be triggered by the initiator associated with the event. Similarly, *request key* must be triggered by the responder, but further requires that the initiator's nonce `n_a` can flow to the `public` label and therefore be publishable, and that the responder generated a nonce `n_b` prior to the event, labeled with `readers [P b; P a; P srv]` (for initiator `a`, responder `b`, and server `srv`) and annotated with a usage identifying it as nonce. The principal triggering *send key* must be the authentication server. To trigger the event, the server must have generated the session key `k_ab`, which receives the label `readers [P srv; P a; P b]`. The usage of the key indicates that it is destined to be used in AEAD. Moreover, we require that the responder has requested a key for the same combination of principals as those associated in the event, or that one of the state sessions of the responder or server is corrupted and `n_b` is publishable. The initiator subsequently receives the key directly from the server and forwards it to the responder. In this course, it triggers the *forward key* event to signal that it accepts the key and is about to pass it on. The initiator requires that the server has previously triggered *send key* and agrees on the principals as well as the nonces associated with the protocol run, unless the initiator or server is corrupted and `n_b` is publishable. The last event *receive key* indicates completion of the protocol and acceptance of the key by the responder. Its pre-condition is more complex due to the subtlety of the last protocol step, where the responder must combine the information it gets from the two separately encrypted parts of the last message. Two conditions must hold for the responder to accept the key. The first condition is basically the equivalent of the condition ensured by the initiator when forwarding the key regarding the occurence of *send key* minus the timeliness guarantee provided by the nonces. Timeliness is implied by the second condition, which ensures that the initiator has forwarded the key `k_ab` and matches the responder's nonce `n_b`.

Protocol Steps

The protocol steps in the Yahalom model are again realized in form of five independent and arbitrarily schedulable functions in a module `YLM.Protocol`. As in the Otway-Rees model, we also include an API for long-term key installation in the protocol API. However, since there are no longer state sessions that directly model the initial knowledge of principals, we only require two functions `new_lt_key_session` and `install_lt_key_at_auth_server` for this purpose. Instead, we model terms that count to the initial knowledge of principals other than the long-term keys as direct inputs to the protocol functions.

First Step (Initiator) In the first step of the Yahalom protocol, the initiator starts a run of the protocol by sending the first message.

```
val initiator_send_msg_1:
    a:principal ->
    kas_idx:nat ->
    b:principal ->
    LCrypto (msg_idx:timestamp * sess_idx:nat) (pki ylm_preds)
    (requires (fun t0 -> True))
    (ensures (fun t0 (mi, si) t1 ->
        mi < trace_len t1 /\ trace_len t0 < trace_len t1))
```

The initiator is denoted by `a`. Its initial knowledge comprises the long-term key session with the server stored at `kas_idx` in the initiator's state and the identity of the responder `b`. The function outputs a message trace index of the first protocol message and a session index pointing to a new state session created by the initiator. The initiator first retrieves the long term key session of the form `PKeySession srv k_as` containing the server `srv` and the long-term key `k_as` from its state and generates a nonce `n_a` labeled `public`. Next, it triggers `event_initiate a b srv n_a`, signaling the start of a protocol run involving the three principals `a`, `b` and `srv` in their respective roles and the challenge n_a for the server. After triggering the event, the initiator creates the first message Msg1 a n_a and sends it to the responder in plaintext. Finally, it stores a new state session InitiatorSentMsg1 b n_a with the identity of the responder and the challenge it just generated.

Second Step (Responder) The second step is executed by the responder, who requests a session key from the server based on the first message from the initiator.

```
val responder_send_msg_2:
    b:principal ->
    kbs_idx:nat ->
    msg1_idx:timestamp ->
    LCrypto (msg_idx:timestamp * sess_idx:nat) (pki ylm_preds)
    (requires (fun t0 -> msg_idx < trace_len t0))
    (ensures (fun t0 (mi, si) t1 ->
        mi < trace_len t1 /\ trace_len t0 < trace_len t1))
```

The first function parameter b denotes the responder. kbs_idx points to the long-term key session of the responder with the server that was established prior to the protocol. The first message from the initiator lies at msg1_idx on the trace. Similarly as the first step, the second step also outputs a message trace index – of the second message – and an index of a new state session in the responder's state. The responder begins by receiving the first message, revealing the identity of the initiator a. Next, it parses the received message, which it expects to be of the form Msg1 a' n_a. It checks if a = a' and, if yes, goes on to retrieve the long-term key state session PKeySession srv k_bs from its state. The responder generates its own challenge n_b, which is supposed to be secret among the principals involved in the protocol, and therefore gets the label readers [P b; P a; P srv]. With that, it has all it needs to request the key from the server, so it triggers event_req_key a b srv n_a n_b, adding its nonce n_b to the data associated with the protocol run. The responder creates EncMsg2 a n_a n_b and encrypts it under k_bs for the server. It then prepends its own identity to the resulting ciphertext in Msg2 b c_ev2, and sends the message to the server. After sending the message, it stores ResponderSentMsg2 a srv n_b in a new state session corresponding to the protocol run. As already pointed out, the server identity is required here to make a statement about the label of n_b.

Third Step (Server) The key generation and distribution performed by the server as response to the request from the responder is implemented in a function server_send_msg_3.

```
val server_send_msg_3:
    srv:principal ->
    msg2_idx:timestamp ->
    LCrypto (msg_idx:timestamp * sess_idx:nat) (pki ylm_preds)
    (requires (fun t0 -> msg_idx < trace_len t0))
    (ensures (fun t0 (mi, si) t1 ->
        mi < trace_len t1 /\ trace_len t0 < trace_len t1))
```

The first parameter srv is again the principal executing the step – the authentication server in this instance. msg2_idx is the trace index of the second message coming from the responder. Again, the protocol step returns a message trace index and a state session index. The first action of the server is to receive the second message. From the receive function, it gets the serialized message and the identity of the responder b. The message is then parsed, expectedly resulting in

Msg2 b' c_ev_b. The server verifies the identity of the responder b with the one in the message and decrypts the ciphertext c_ev_b encrypted by the responder. Therefore, the server first retrieves the server key session AuthServerSession b k_bs via b. Successful decryption should then reveal the encryption value EncMsg2 a n_a n_b. The server now knows the initiator a and can hence retrieve the corresponding key session AuthServerSession a k_as for a and generate the session key k_ab. The session key is annotated with the label readers [P srv; P a; P b] to express the requirement that the key shall remain secret between a, b, and srv, as well as usage aead_usage "YLM.comm_key" to identify it as AEAD key for direct communication between users. The server is now ready to distribute the key and signals this via event_send_key a b srv n_a n_b k_ab, further extending the data associated with the protocol run in the *request key* event with the session key k_ab. After that, it creates two encryption values EncMsg3_I b k_ab n_a n_b and EncMsg3_R a k_ab and encrypts them under the long-term keys k_as of the initiator and k_bs of the responder. The third message is simply the concatenation of the ciphertexts for initiator and responder, i.e., Msg3 c_ev3_i c_ev3_r, and is sent to the initiator. In order to be able to prove key secrecy from the server's perspective, the server must again store the key and the principals that get a handle to it in a state session AuthServerSentMsg3 a b k_ab.

Fourth Step (Initiator) In the fourth step, the initiator receives the session key from the server, and forwards it to the responder.

```
val initiator_send_msg_4:
    a:principal ->
    kas_idx:nat ->
    msg3_idx:timestamp ->
    a_sess_idx:nat ->
    LCrypto (msg_idx:timestamp) (pki ylm_preds)
    (requires (fun t0 -> msg_idx < trace_len t0))
    (ensures (fun t0 mi t1 ->
        mi < trace_len t1 /\ trace_len t0 < trace_len t1))
```

The parameters a and kas_idx are the same as in the first step. msg3_idx denotes the trace index of the third message from the server. The state session of the initiator from the first protocol step associated with the current protocol run is stored in the initiator's state at index a_sess_idx. At the end of the fourth step, the initiator updates the state session from the first step, so that the function only outputs the trace index of the fourth message. The initiator first gets the state session InitiatorSentMsg1 b n_a from the first step. It then receives and parses the third message Msg3 c_ev_a c_ev_b and gets the long-term key session PKeySession srv' k_as. The server identity returned by the receive function is compared with the identity stored in the key session. If they match, the initiator decrypts the first encrypted part c_ev_a and expects a value EncMsg3_I b' k_ab n_a' n_b. It checks whether the server generated a key for the correct responder b by checking that b = b' and whether the server message originated from the current protocol run by checking that n_a = n_a'. After having validated the message, the initiator triggers event_fwd_key a b srv n_a n_b k_ab to show that it accepted k_ab and now forwards the second encrypted part – also containg the key – to the responder. Since the responder does not get timeliness guarantees from the part encrypted by the server and therefore cannot deduce that k_ab is not a replay, the initiator creates a second encryption value EncMsg4 n_b and encrypts it with the session key k_ab. The ciphertext from the server and the ciphertext c_ev4 from the initiator are concatenated in the fourth message Msg4 c_ev_b c_ev4, which is sent to the responder. Finally, the initiator updates its state session corresponding to the protocol run to InitiatorSentMsg4 b srv k_ab, storing the session key and its peers.

Fifth Step (Responder) The final step is the responder receiving the fourth message from the initiator, from witch it gets the session key and can deduce that the key is not a replay.

```
val responder_recv_msg_4:
    b:principal ->
    kbs_idx:nat ->
    msg4_idx:timestamp ->
    b_sess_idx:nat ->
    LCrypto unit (pki ylm_preds)
    (requires (fun t0 -> msg_idx < trace_len t0))
    (ensures (fun t0 _ t1 -> trace_len t0 < trace_len t1))
```

The first two parameters b and kbs_idx are again the responder and the index of the long-term key session with the server. The fourth message from the initiator is stored at msg4_idx on the trace. b_sess_idx is the index of the responder's state session from the second step. Since the responder – similar to the initiator – only updates its previous state session and also does not send any further message in the step, the function just outputs unit. The last step begins with the responder retrieving and parsing the state session ResponderSentMsg2 a srv n_b it stored at the end of the second step. If it finds the expected session at b_sess_idx, the responder receives the fourth message from the trace and verifies whether the actual sender returned by the receive function is the initiator a stored in its state. The fourth message is then parsed, which should yield Msg4 c_ev3_b c_ev4_b. Given the parsed message, the responder retrieves its long-term key session PKeySession srv' k_bs. If the responder followed the protocol correctly, the server stored in its long-term key session srv' and the server srv it associates with the protocol run should be the same. The responder can finally start to decrypt the ciphertexts c_ev3_b and c_ev4_b. It starts with the first ciphertext from the server, which it can decrypt with k_bs. The responder obtains EncMsg3_R a'' k_ab, and once again verifies the identity of the initiator, this time included in the key certificate from the server as a''. Besides this, the responder receives the session key k_ab, which it uses to decrypt the second ciphertext. The decryption results in EncMsg4 n_b' containing the responder's nonce, for which we must check that n_b = n_b' to deduce the timeliness of the fourth message. Now that the responder has the session key and is assured of the timeliness of the message, it updates its state session associated with the protocol run to ResponderRecvedMsg4 a srv k_ab, similarly as the initiator did in the previous step.

Execution

The Yahalom model can be executed similarly to Otway-Rees in a benign environment implemented via a scheduler function in which a passive adversary schedules the protocol steps of honest principals in the correct order. This way, we can verify the correctness of the Yahalom model by checking if the protocol runs successfully and achieves the desired effects based on its output trace. The scheduler function simulating a benign adversary is again named benign_attacker and is defined in a module YLM.Debug:

```
let benign_attacker () =
    let a:principal = "alice" in
    let b:principal = "bob" in
    let srv:principal = "server" in

    let ((|t_kas,k_as|), kas_idx) = new_lt_key_session a srv in
    let ((|t_kbs,k_bs|), kbs_idx) = new_lt_key_session b srv in
    install_lt_key_at_auth_server #t_kas srv a k_as;
    install_lt_key_at_auth_server #t_kbs srv b k_bs;

    let (msg1_idx, a_sess_idx) = initiator_send_msg_1 a kas_idx b in
    let (msg2_idx, b_sess_idx) = responder_send_msg_2 b kbs_idx msg1_idx in
    let (msg3_idx, srv_sess_idx) = server_send_msg_3 srv msg2_idx in
    let msg4_idx = initiator_send_msg_4 a kas_idx msg3_idx a_sess_idx in
    responder_recv_msg_4 b kbs_idx msg4_idx b_sess_idx;
    ()
```

We start as usual, by initializing the principals a, b, and srv that are going to run the protocol together. a and b are respectively the initiator and responder in the protocol, while srv is the authentication server. The principal initialization is followed by the generation and installation of the long-term keys between the users and the server, resulting in indices kas_idx and kbs_idx pointing to the long-term key sessions of a and b. Note that the resulting state sessions do not necessarily model the complete initial knowledge of the users at protocol start. To be able to initiate a protocol run, the initiator must however know the responder. Therefore, b is input to the first protocol function initiator_send_msg_1, which starts the protocol. The other protocol steps follow in the intended order and the protocol completes with responder_recv_msg_4. benign_attacker in the Yahalom model is also wrapped in a function benign that prints the trace after execution and is called from the model's main function in YLM.Debug.

4.4.3 Correctness and Coherence of the Model

To establish the correctness of our Yahalom model, we proceed in the same way as for Otway-Rees. First thing, we perform a type check of the individual modules in F*. If the type check is successful, we can compile our model into an executable. The entry point of the executable is the main function in YLM.Debug, which executes the scenario of a benign Yahalom attacker defined above and prints the resulting trace. Appendix B contains the trace produced by the benign attacker. It shows the successful simulation of a regular protocol run with the desired outcome that initiator and responder share a session key generated and distributed by the authentication server, which means that our model works as expected. Knowing that our developed model runs successfully, it is left to show that it also coheres with the description of the Yahalom protocol from Section 3.3.2. The coherence of the Yahalom model can be demonstrated based on the same six points as that of the Otway-Rees model:

- The model correctly represents the principals' initial knowledge. In comparison to the Otway-Rees model, the Yahalom model does not incorporate explicit state sessions for the initial knowledge of principals. Instead, we have state sessions for long-term keys only and model the remaining terms that are initially known to principals directly as input parameters to the respective protocol functions. In case of the Yahalom protocol, this only concerns the identity of the responder, which is publishable and must be known to the initiator in advance. The semantics of passing public initial knowledge directly as input parameters to protocol functions or storing it in a state session and passing the index of the state session as input parameter instead are equivalent. From the semantic equivalence and the coherence of the initial knowledge in the Otway-Rees model, the coherence in the Yahalom model follows.

- The origin of nonces and keys corresponds in the model and in the protocol description. Two nonces N_a and N_b are generated during the protocol, respectively, by the initiator and by the responder. The server generates the session key k_{ab}. Like in the Otway-Rees model, the long-term keys k_{as} and k_{bs} are generated by the users, as it is not made clear in the description of the Yahalom protocol where they originate from.
- Nonces and keys are labeled with their correct intended audience. Intuitively, the long-term keys are labeled `readers [P a; P srv]` and `readers [P b; P srv]`, since they form the secure communication channels of the respective user and server. N_a is transmitted in plaintext in the first message and is thus `public`. The responder nonce serves as challenge for the server, first encrypted under k_{bs}, and then indirectly sent back to the responder encrypted under the session key k_{ab} by the initiator. The session key is generated by the server as a means for direct and secure communication between the users and is thus labeled `readers [P srv; P a; P b]`. Since N_b is encrypted under k_{ab}, it can only be so secret and gets the label `readers [P b; P a; P srv]`.
- The module `YLM.Messages` correctly represents the actual message structure. Similar to Otway-Rees, we have two types `encval` and `message i`. One models encrypted parts of messages and one models the messages as a whole. The encrypted parts are the responder's encryption `EncMsg2` under k_{bs} (`k_bs`) in the second step, the two encryptions `EncMsg3_I` under k_{as} (`k_as`) and `EncMsg3_R` under k_{bs} performed by the server in the third step, and the initiator's encryption `EncMsg4` under k_{ab} (`k_ab`) in the fourth step. Together, the two types can be used to form messages that reflect the actual message structure.
- The stateful parts of the protocol are correctly represented in the module `YLM.Sessions`. The `session_st` type provides a means to store terms related to a protocol run in state sessions affiliated with principals. The main goal of the Yahalom protocol is – similar to Otway-Rees – the exchange of a session key, which can then be used for direct communication between the users. Therefore, we again store the exchanged session key in the final state sessions of initiator and responder, where they can access it at a later time. Other terms like nonces are exclusively relevant to a specific protocol run and can thus be abandoned when the run completes. As in the Otway-Rees model, the server stores the session key as well, so when proving the security of the protocol, we can express the key secrecy property from the server's perspective based on its state. The model is hence valid in that it correctly captures the protocol's outcome via the principals' state sessions.
- The checks and message content verification performed in the protocol steps in `YLM.Protocols` are reasonable and not too restrictive. Again, we perform verification of the message sender identity whenever we can. As already explained for the Otway-Rees model, this does not restrict the attacker in its capabilities. Besides this, in the fourth and fifth steps, the initiator and responder verify each other's identity contained in the key certificates issued by the server in the third step. The inclusion of the key peer identity in the key certificates was crucial for the key secrecy in the improved Otway-Rees protocol. So the fact that the Yahalom protocol

includes them only makes sense if they are also verified by the respective recipient; otherwise they would be useless. The verification of the challenges N_a by the initiator and N_b by the responder – also in the fourth and fifth steps – is taken directly from the protocol description. In the fifth step, the responder additionally checks whether the server identity stored in its key session matches with the one stored in its state session affiliated with the protocol run. If the responder executes the protocol correctly, this is already ensured by the implementation of the protocol functions. The check is nevertheless required to make this information explicit to F*. As demonstrated, we only perform verification of message contents if it is reasonable or if the protocol description demands it, resulting in the validity of the model from a higher-level implementation perspective.

4.4.4 Security Properties

In this section, we prove the security of the Yahalom protocol with respect to the security goals elaborated in Section 4.4.1 by formulating concrete security objectives and defining corresponding security lemmas for these objectives in an additional module `YLM.SecurityProps`, which we verify together with the rest of the coherent Yahalom model in F*. Given the similarities in the procedures of the protocols Yahalom and Otway-Rees, it is no surprise that they also have very similar security goals. In fact, the secrecy objectives S1 and S2, and the authentication objectives A1 and A2 expressed for the Otway-Rees protocol in Section 4.3.4 also apply to the Yahalom protocol. Howbeit, we have one additional authentication objective for the Yahalom protocol resulting from the use of the session key k_{ab} to encrypt the responder's challenge N_b in the last protocol message:

(A3) the responder is assured that the initiator has accepted and therefore trusts the session key.

Secrecy

The secrecy objectives S1 and S2 can be shown exactly as in the Otway-Rees model by proving that the key stored in the respective final state sessions of server, initiator, and responder, is unknown to the attacker unless at least one of their state sessions is corrupted. The property from the server's perspective here proves objective S1, concerned with the secrecy of the generated key itself, while the property from the initiator's and responder's perspective ensures objective S2, i.e. that the users receive this secret key and accept it as session key as a result of the protocol, instead of some other term that the attacker may possibly know. We omit the details of the secrecy proof in the Yahalom model and refer to the identical secrecy proof of the (improved) Otway-Rees protocol in Section 4.3.4 and Section 4.3.6, respectively.

Authentication

Due to the unusual message order and structure in the Yahalom protocol, the authentication objectives result in slightly different lemmas than in the improved Otway-Rees model. We again divide the mutual authentication property into two separate lemmas, `initiator_authentication` and `responder_authentication`, both of which satisfy the objectives A1 and A2 from the perspective of the respective user. The `initiator_authentication` lemma further ensures the new authentication goal A3, specific to the Yahalom protocol. The authentication properties of the Yahalom protocol are also specified in terms of trace event relations based on the events we defined in the model in Section 4.4.2.

```
val initiator_authentication: i:nat ->
LCrypto unit (pki ylm_preds)
(requires (fun t0 -> i < trace_len t0))
(ensures (fun t0 _ t1 -> forall a b srv n_b k_ab.
    did_event_occur_at i b (event_recv_key a b srv n_b k_ab)
    ==> (exists n_a. did_event_occur_before i a (event_fwd_key a b srv n_a n_b k_ab)
    \/ corrupt_id i (P a) \/ corrupt_id i (P b) \/ corrupt_id i (P srv))))
```

The parameter `i` again denotes some valid trace index. For arbitrary initiator `a`, responder `b`, server `srv`, responder challenge `n_b`, and session key `k_ab`, the lemma ensures that if the responder triggered `event_recv_key a b srv n_b k_ab` at trace index `i`, the initiator must have previously triggered `event_fwd_key a b srv n_a n_b k_ab` with arbitrary `n_a`, or at least one state session of initiator, responder or server is corrupted. Disregarding corruption, this means that if the responder `b` receives back its correct challenge `n_b` and accepts the session key `k_ab` in the fifth protocol step, then it has been the initiator `a` who previously forwarded `n_b` and `k_ab` to the responder and both users have the same view regarding the principals involved in the protocol run. From the correspondence of `n_b` among the events, the responder gets the timeliness assurance contained in objective A2. Furthermore, the correspondence of `k_ab` means the responder learns that the initiator has accepted the same session key as a result of the protocol run and therefore also trusts it to be secret, essentially proving objective A3. Finally, the relation imposed on the events and the principals matching in the events imply that `a` executed the protocol as initiator and `b` executed the protocol as responder, and both see each other's correct identity. This shows, on the one hand, the remaining part of the authentication objective A2, which is to convince the users that they have talked to each other, and, on the other hand, the objective A1, which is for the users to comply with the principal roles in the protocol. The lemma is automatically verified within the DY* model with F*, which proves it once and for all.

Responder authentication is similarly proven, but based on a different event relation.

```
val responder_authentication: i:nat ->
LCrypto unit (pki ylm_preds)
(requires (fun t0 -> i < trace_len t0))
(ensures (fun t0 _ t1 -> forall a b srv n_a n_b k_ab.
    did_event_occur_at i a (event_fwd_key a b srv n_a n_b k_ab)
    ==> (did_event_occur_before i b (event_req_key a b srv n_a n_b)
    \/ corrupt_id i (P a) \/ corrupt_id i (P b) \/ corrupt_id i (P srv))))
```

In particular, responder authentication is expressed as relation between the events *forward key* and *request key* of initiator and responder. The property states that if the initiator triggered `event_fwd_key a b srv n_a n_b k_ab` at trace index `i` for the same arbitrary principals and terms as in initiator authentication, and arbitrary initiator challenge `n_a`, then there has been a prior event `event_req_key a b srv n_a n_b` triggered by the responder. Essential are here the correspondence of the principals and the initiator challenge among the events, which ensures that the initiator forwarding a key `k_ab` to the responder is preceded by a request of the responder to the server including `n_a` and the same principals with the same role distribution. The initiator can conclude that the request must have occurred in the same protocol run based on the timeliness and locality guarantees it gets from verifying its challenge. The fact that both users agree on the principals and roles involved in the protocol run and that the initiator can deduce the timeliness of the request by the responder covers the authentication objectives A1 and A2 from the initiator's perspective. Since the responder does not use `k_ab` for encryption in the course of the protocol, the initiator has no way of knowing whether the responder will eventually obtain the same session key. Thus, the slightly weaker responder authentication lemma already completes the mutual authentication property of the Yahalom protocol. To prove the lemma's soundness, it is also verified in F*.

4.4.5 Discussion

During the analysis, security related properties and goals of the Yahalom protocol have been detailed and incorporated in a coherent model of the protocol developed in DY*. The security of the protocol has been analysed based on the protocol's properties and concrete secrecy and authentication objectives were defined. Finally, we formalized the objectives through several security lemmas in the DY* model and verified them using F*, resulting in a full Yahalom security proof. In Section 3.3.2, different methods to analyse the security of the Yahalom protocol and their results, as well as attacks on the protocol, have been summarized. Similarly as for Otway-Rees, we discuss these analyses and attacks in the context of our own security proof to explain possible differences in the outcome, but also to draw parallels with other security proofs.

Besides the protocol of Otway and Rees, the Yahalom protocol has also been analysed with the BAN logic of Burrows et al. [11]. The BAN analysis shows that the protocol results in strong security implications for honest users, but has a minor flaw that would allow the initiator to replay an old session key without the responder noticing. For the BAN authors, this does not impose a major problem because they assume principals to not act maliciously in the context of a BAN analysis. With the active attacker model present in DY*, we cannot assume that principals do not deviate from the protocol steps. In Section 4.3.4, we were however able to prove security objectives S2, which states that the users must obtain a session key generated by the server as a result of the protocol, and A2, which requires that the users recently talked to each other (particularly, in the protocol run in which the key has been exchanged). To proof S2, we deduced a statement about the label of the key stored in the state sessions of initiator and responder from the respective key certificates issued by the server. Since the DY* adversary can just replay an old key certificate from the server on behalf of the initiator, this alone is not enough to ensure that the key is not a replay. This is where the authentication objective A2 comes into play. The proof of A2 bases on the timeliness and locality implications of the challenges generated by the users during their respective first steps in a protocol run. Verifying the challenge encrypted with the session key in the last step assures the responder that the session key is fresh, because it has been used to encrypt a randomly generated nonce that the responder associates with the current protocol run. The initiator itself could still trick the responder into accepting an old key if it simply uses that key to encrypt the challenge of the responder in the fourth message, but this is viewed as adversarial behavior in DY* and hence requires the adversary to corrupt the initiator's long-term key session in advance. While an analysis with the BAN logic only considers the case where the protocol is run by honest principals, DY* defines the adversary as a concept that covers all malicious behavior, directly or indirectly through principals under its control. If a principal deviates from the protocol, it must be made explicit by corruption, which in turn reflects in the provable security properties for the protocol. In case of the Yahalom protocol, the result is the same, that this detail does not make the protocol any less secure, as a malicious initiator could just as easily leak a fresh session key to the adversary and therefore break security.

Still, the flaw pointed out by the BAN authors prompted Chen and Shi [13] to perform an elaborate analysis of the Yahalom protocol with the SVO logic. They come to the result that the protocol does not achieve its authentication goals. Thus, they propose an improved version of the protocol and use the SVO logic to prove that it achieves the level of security they intend. While the BAN analysis merely aims to study the authentication properties of the Yahalom protocol between honest parties, Chen and Shi consider the possibility of a malicious initiator in their analysis. Therefore, they slightly weaken their initial assumptions compared to the BAN analysis by omitting assumptions that let the responder deduce the freshness of the session key. In the SVO derivation for the responder, they are thus not able to show that the responder believes in the freshness of the session key, which they identify as flaw in the protocol that could be exploited by an attacker to make the responder accept an old key. On the other hand, Chen and Shi also define stronger security goals for the protocol in terms of the SVO logic. Both users shall be convinced that the other shares its believes about the session key. Intuitively, this is not possible because the initiator has no way of knowing whether the responder ever receives the session key, as the responder never encrypts a message under it during the protocol. This also reflects in the results of our analysis in a slightly weaker responder authentication property compared with the initiator authentication property we were able to prove. For this matter, Chen and Shi propose several modifications to the protocol to make up for the deficits. The most important change is an additional fifth message from the responder to the initiator encrypted with the session key accepted by the responder. This way, the initiator is reassured that the responder actually received and accepted the session key forwarded by the initiator with the fourth message, and can therefore assume that the responder trusts the session key to be secret. Moreover, even a malicious initiator cannot replay an old session key in the fourth message anymore, since the server includes the responder's challenge in its key certificate, which cannot be manipulated by the initiator. This finally lets the responder deduce the freshness of the key, which the initiator also learns when it receives the fifth message. The modifications therefore have the desired effects, but achieve a much stronger level of security than originally intended. We have explained above, that the original protocol already assures the responder that the session key is fresh as long as the initiator is not controlled by a malicious party (the DY* adversary). Further, the intention of the original protocol was clearly not to reassure the initiator that the responder accepts the session key; otherwise, the protocol would already contain a handshake between initiator and responder where they both exchange messages encrypted under the session key. Like in our improved version of Otway-Rees, the initiator gets this information when receiving the first message from the responder after completing the protocol. Another attack on the Yahalom protocol – besides the replay attack detected by the BAN authors – is described in [14] and has already been discussed in Section 3.3.2. In the attack, the initiator replays the responder's encryption from the second message as key certificate from the server to make the responder accept the concatenation of the two challenges as session key. In a way, this attack is the equivalent of the type flaw attacks on the Otway-Rees protocol presented in Section 4.3.5. We have clarified that this attack – similarly to the replay attack from above – requires the corruption of the initiator, who must somehow obtain the challenge of the responder, and does thus not contradict our security proof in DY*. Instead, the attack once more stems from different assumptions about the intended level of security that the protocol shall achieve. As we have already precisely argued, our security proof attempts to show that the Yahalom protocol achieves the level of security targeted by Burrows et al. in the formal description of the protocol and with the BAN analysis, and is therefore not in contradiction with other research that tries to prove stronger or different security properties.

Not only that, but our security proof is also backed up by the results of analyses like that of Ryan et al. [31] with the CSP approach or Paulson [28] with their inductive trace based method. Ryan et al., for example, define a general secrecy property stating that a message claimed secret by its sender is in fact secret if both sender and receiver are honest. Among other things, Ryan et al. prove that this also applies to the session key in the Yahalom protocol. This is similar to our secrecy proof in that the server annotates the session key with a label and passes on a statement about this label to the users via the encrypted key certificates. The encrypted certificates can therefore be understood as a secrecy claim by the server. In our proof, the users are able to deduce the label of the session key from their respective key certificate if the long-term key used to encrypt it is secret, which is exactly the case if the state session of the server or the respective user storing the long-term key has not been corrupted by the attacker. Assuming this is the case, the users can use their knowledge about the label of the key to derive a statement about its secrecy. The authentication property is – like in our proof in DY* – divided into an initiator and responder property. Another similarity is that Ryan et al. define the properties using relations of trace events or points passed during protocol execution. In particular, they say that either party must emit a running signal before the respective other party commits to the run, in order to mutually authenticate both parties. For the authentication to be meaningful, the parties must agree on certain terms like principals involved in the run, nonces, or the exchanged key, and associate them with the signaled event. In our DY* model, the running signal of the initiator is the event *forward key*, and the responder uses the event *receive key* to commit to the run. In turn, the responder's running signal is *request key*, while the initiator also commits with *forward key*. With this definition of authentication, Ryan et al. are even able to account for the fact that the initiator authentication property is slightly stronger in that initiator and responder agree on the principals involved in the run, their nonces, and the exchanged key, while the initiator does not get reassurances about the key from the responder through responder authentication. Paulson's secrecy proof is quite similar insofar as it relies on the fact that an honest server distributes the session key on secret channels accessible only to the users, and the users accept only a key that must have come from the server. Initiator and responder authentication are also similar because they ultimately depend on matching key peers in the key certificates, as well as the initiator's nonce in the corresponding key certificate, and the responder's nonce in the second encryption of the last message under the session key by the initiator. We thus have two independent analyses that come to a conclusion comparable to ours, that the Yahalom protocol is secure with respect to its formal specification given in the BAN paper.

4.5 Denning-Sacco

The third analysed protocol is the Denning-Sacco protocol with public keys [16]. Denning and Sacco studied the use of timestamps in key distribution protocols, which resulted in a family of protocols with interesting properties. Particularly, protocols with timestamps can protect against replays of old session keys and make the handshake between the users exchanging a key obsolete. In our analysis, we focus on one specific protocol from Denning and Sacco with the purpose of exchanging a symmetric communication key via public keys and a trusted third party that functions as certification authority (see Section 3.3.3). We prove that the protocol is secure under certain assumptions insofar as that it has the properties described by its authors and achieves basic security goals a simple key exchange protocol should achieve, including, inter alia, key secrecy and basic authentication. The concrete properties are defined, formalized and finally verified in DY* as part of the analysis. This thesis provides the first elaborate security proof of the Denning-Sacco protocol with public keys and strives to give a more detailed insight into the security related properties of the protocol than previous literature could.

4.5.1 Properties and Goals

We start once more by outlining the properties and goals of the protocol taken from the work of Denning and Sacco and their description of the protocol. Based on that, we can later build a representative DY* model of the protocol, which has the discussed properties and achieves the defined goals.

Principal Roles

Like the other protocols, the Denning-Sacco protocol is run between two users A and B with the ultimate goal to exchange a secret key for direct communication between the users. Therefore, the protocol also involves a trusted third party – an authentication server AS – that serves as certification authority for the users' public keys. One speciality of the protocol is that while A again takes the role of the initiator who starts the protocol by requesting the public key certificates of itself and B from the AS, the other user B is only involved in the protocol as receiver of the last message containing the communication key, so that A can never actually know whether B has been online until B sends its first message encrypted under the key. This highlights the importance of the role of the AS in the protocol, since all guarantees that the initiator gets build on its trust in the AS.

Initial Knowledge

The initiator is in possession of a public and private key and knows the pair (AS, P_{AS}), where P_{AS} denotes the public key of the AS, and the user B with whom it wishes to exchange a key. The AS must also possess a public and private key, and must furthermore store all pairs of users and public keys, including the pairs (A, P_A) and (B, P_B). Finally, we have the user B who also posses a public and corresponding private key and knows (AS, P_{AS}) as well. B is however not necessarily aware of A's existence before it receives the key.

Timestamps

Denning and Sacco refrain from using nonces for timeliness and locality guarantees in their protocol, and suggest the use of timestamps in their place. Similar to nonces, timestamps shall ensure that messages are fresh and not replays from previous protocol runs. They trade off the additional locality guarantee from nonces (i.e., the guarantee that a message originates from a particular protocol run) for the omission of the obligatory handshake for nonce verification. To reliably use timestamps, the principals must synchronize their respective local clocks to a reliable global source before running the protocol. The server then obtains a timestamp from its clock and includes it in the certificates issued to the initiator. The initiator receives the certificates and checks if they are fresh by obtaining a timestamp from its own local clock and verifying that it falls within an acceptable tolerance range of the timestamp in the certificates. If they turn out to be valid, the initiator forwards them to the responder, who also validates their freshness. Along with the certificates, the initiator also sends a signed and encrypted part containing the communication key and the same timestamp as in the certificates. This allows the responder to verify that the communication key is fresh.

Exchanged Keys or Secrets

The purpose of the Denning-Sacco protocol with public keys is the exchange of a symmetric key k_c for direct and secure communication between two users with the only commonality that they know an authentication server AS and the corresponding public verification key P_{AS}. The authentication server knows the public keys of a set of users including the keys P_A of A and P_B of B and issues certificates for those keys using its private sign key S_{AS}. The users can then use P_{AS} to verify the certificates containing the keys. A proceeds by choosing a communication key k_c intended for secret communication with B, which it signs using its private sign key S_A and then encrypts using B's public encryption key P_B. B in turn decrypts the message using its private decryption key S_B and then uses A's public verification key P_A to verify the signature. The signature with S_A is necessary such that B is able to verify the origin of the key, and the encryption under P_B shall ensure that only the possessor of S_B, i.e. B, obtains k_c.

Authentication Goals

Denning and Sacco are not precise regarding the authentication properties of their protocol. However, a careful look at the message structure gives us an idea of how exactly the two users authenticate. The AS plays a central role in mutual authentication, as it issues certificates that prove the authenticity of the users' public keys. If the certificates are valid, and A and B honest, the users can be sure about the origin of subsequent messages encrypted under the communication key k_c. A because it has transmitted k_c encrypted under P_B certified by the AS, and B because, as explained above, it can verify the origin of k_c via P_A, also certified by the AS. B is provided with even stronger guarantees in that it is assured that A emitted a message containing k_c and intended for B. The authentication properties are extended with timeliness guarantees via the timestamps in the certificates and in the signature of k_c. As explained above, the timestamps ensure that the certificates and the signature of k_c are current and therefore prevent possible replays of compromised session keys.

4.5.2 Modeling Time-based Properties

One important goal of this thesis was to look for ways to model the described time-based or time-dependent properties of the Denning-Sacco protocol in the DY* framework. In this section, we explain the difficulties of modeling time-based properties in symbolic provers like DY*. We present a possible generic solution to the problem and demonstrate it with an implementation in the DY* model of the Denning-Sacco protocol. Finally, we discuss the implications of our solution and the tradeoffs that must be made when abstracting from time-based properties of cryptographic protocols in the real world.

Problem and Goal

Although the timestamps are supposed to protect from replays of old compromised session keys, they cannot protect against such attacks with full certainty. If an adversary is able to compromise a key while the server's timestamp is still valid, the responder would accept the replayed key. This tolerance window is necessary to ensure that the responder does not erroneously reject legitimate key certificates. Denning and Sacco suggest a tolerance between one and two minutes to account for discrepancies between the global time sources used by the principals to synchronize their local clocks plus some additional time for the expected network delay. This results in a vague definition of security due to the small remaining chance of a replay. Because it is very unlikely that a single session key will be compromised in such a short time after the key exchange (given the protocol sufficiently protects the key otherwise), this chance is considered negligible in reality. In contrast, symbolic provers like DY* usually aim to prove security properties of cryptographic protocols in an ideal world that does not allow uncertainty. Our goal is thus to abstract from the uncertainty of time-based properties in the real world and model the property in DY* such that it clearly defines when timestamps protect against replays.

Idea

First and foremost, to formulate the time-dependent security properties as ideal properties in DY*, we must eliminate factors that bring non-determinism into our model. This requires certain abstractions in our model from the real world, and from the way Denning and Sacco describe the use of timestamps in their protocol. For example, we must neglect the discrepancy between clocks in our model and instead use a single global source of time for all principals. DY* already uses the global trace length to describe how certain events during protocol execution relate in time. In that sense, the trace length is an ideal global source of time because of its monotonically increasing property and the fact that it is globally accessible via the `global_timestamp` function in the symbolic runtime layer. Another necessary abstraction is to determine the exact delay between messages instead of estimating it. Since protocols are modeled in DY* as a set of fixed protocol steps that are intended to execute in a particular order resulting in the same trace each time, this would generally be possible using the length of the trace as measure of time. The problem with this approach is that a DY* model of a protocol should generally allow to run multiple instances of the protocol concurrently. If we use the current trace length as timestamp in the Denning-Sacco protocol and run multiple instances of the protocol in parallel, the increase in trace length reflects in all protocol sessions and thus invalidates timestamps, causing the protocol to fail in cases where the initiator actually sent a fresh key. Hence, we must somehow track time in different protocol sessions independently, while the only global measure of time we have is the trace length. This problem is solved by using the trace length as global clock that returns absolute timestamps, and then measuring the time in the corresponding protocol run relative to the point marked by the timestamp and independent of other runs. How this relative and protocol session independent measure of time is defined, and how it changes the way a timestamp is validated, is discussed in detail in the following section.

Implementation

We implement our solution in a module `DS.Clock` as part of the DY* model of the Denning-Sacco protocol. In the module, we introduce a type `clock` with the purpose of measuring the time elapsed in a protocol run relative to an absolute timestamp. We define `clock` as `Type0` in the interface of `DS.Clock` to abstract from its concrete implementation.

```
val clock:Type0
```

Internally, `clock` is implemented by a type `clock_`, which is defined as a tuple of a natural number `counter` – the clock counter representing the elapsed time – and the timestamp for which the elapsed time is measured.

```
let clock_ = (counter:nat * timestamp)
let clock = clock_
```

By keeping the implementation private, it is hidden from other modules, thereby preventing an adversary from creating clocks for arbitrary timestamps and setting the clock counter to zero to trick honest principals into accepting invalid timestamps. Instead, clocks are created using a function `clock_new` that obtains the current trace length as timestamp, appends to the trace to prevent the instantiation of a second clock with the same timestamp, and initializes the clock counter to zero.

```
val clock_new: #pr:preds -> principal ->
    LCrypto (c_new:clock & ts:timestamp) (pki pr)
    (requires (fun t0 -> True))
    (ensures (fun t0 (|c_new,ts|) t1 ->
        trace_len t1 == (trace_len t0) + 1  /\
        clock_get c_new == 0 /\
        ts == trace_len t0))
```

The function accepts two arguments: the first argument `pr` is a set of usage and trace predicates defined at the individual protocol level – in this instance the Denning-Sacco protocol –, and the second argument is the principal instantiating the clock. Returned is a tuple consisting of the new clock `c_new` and the timestamp `ts` at which `c_new` has been instantiated. `ts` can then be used as timestamp in protocol messages, for example of the Denning-Sacco protocol, and `c_new` tracks the elapsed time and is used for validation of `ts` later. Since `clock_new` implicitly effects the trace, we annotate the result with the `LCrypto` effect. The desired properties of the function are captured with the `requires` and `ensures` predicates. In particular, the function takes an arbitrary input trace `t0` and results in an output trace `t1` with one additional entry. Moreover, the counter of the new clock returned by `clock_get` is zero, and the returned timestamp is equal to the trace length of the input trace. A similar function `att_clock_new` is defined for usage by the DY* adversary.

```
module A = AttackerAPI

val att_clock_new: unit ->
    Crypto (c_new:clock & ts:timestamp)
    (requires (fun t0 -> True))
    (ensures (fun t0 r t1 ->
        match r with
        | Success (|c_new,ts|) ->
            A.attacker_modifies_trace t0 t1 /\
            trace_len t1 == (trace_len t0) + 1 /\
            later_than (trace_len t1) (trace_len t0) /\
            clock_get c_new == 0 /\
            ts == trace_len t0
        | Error _ -> t0 == t1
    ))
```

The attacker function is defined in terms of the `Crypto` effect instead of the `LCrypto` effect and thus integrates well with other attacker functions defined in `AttackerAPI`. The two input arguments of `clock_new` are omitted in case of `att_clock_new` and are replaced by a single input of type `unit`. The validity of the trace, which is implicit in the `LCrypto` effect, is explicitly ensured by `A.attacker_modifies_trace` for `att_clock_new`.

```
let clock_new #pr p =
    let ts = global_timestamp () in
    let _ = send #pr #ts p p (nat_to_bytes #pr.global_usage #ts 0 ts) in
    (|(0, ts),ts|)

let att_clock_new () =
    let ts = A.global_timestamp () in
    let _ = A.send #ts "*" "*" (A.pub_bytes_later 0 ts (A.nat_to_pub_bytes 0 ts)) in
    (|(0, ts),ts|)
```

Since the implementation of the `clock` type is private, the implementations of `clock_new` and `att_clock_new` must also be internal to the `DS.Clock` module. Both implementations are quite similar. To create a new clock, both first obtain the current length of the trace via `global_timestamp`, or `A.global_timestamp`, in the adversary's case. After that, a symbolic message containing the serialized timestamp is sent and stored on the trace, resulting in a new trace entry. The new entry on the trace ensures that the returned clock is unique with respect to its associated timestamp. In `clock_new`, we use the `send` function from `LabeledPKI` for this purpose, which is defined in terms of the `LCrypto` effect and thus takes the predicates `pr` as input. We define `p`, the principal instantiating the clock, to be both sender and receiver of this symbolic message. On the other hand, `att_clock_new` uses the `send` function of `AttackerAPI` and respectively replaces sender and receiver with wildcard strings. Both functions finally return a tuple consisting of the new clock – which is itself a tuple `(0, ts)` – and the timestamp `ts`. Returning `ts` separately is required, since we have no way to extract the timestamp from the clock outside of the private `DS.Clock` implementation.

Given a timestamp and a corresponding clock, validation of the timestamp can be performed using a function `clock_lte`.

```
val clock_lte: ts:timestamp -> max_delay:nat -> c:clock -> r:(result bool)
    {
        match r with
```

```
          | Success true -> clock_get c <= max_delay
          | _ -> True
      }
```

The function accepts parameters `ts`, the timestamp to validate, `max_delay`, the maximum expected network delay, and `c`, the clock used to validate the timestamp. The returned result `r` is a refinement type of `result bool`, and ensures that the clock counter is less or equal to `max_delay` in case that `r == Success true`.

```
let clock_lte ts max_delay c =
    let (cnt_c, ts_c) = c in
    if ts_c = ts then Success (cnt_c <= max_delay)
    else Error "[clock_lte] Timestamps do not match"
```

The implementation of `clock_lte` is again part of the private implementation file of the clock module. We start by extracting the counter and timestamp of `c` into bindings `cnt_c` and `ts_c`. Then, we check that `ts` is equal to the timestamp `ts_c` stored by the clock to make sure that the clock was created at trace length `ts` and not at some other time. Matching timestamps are required such that a comparison of the clock's counter `cnt_c` and the expected network delay are meaningful. The property we want to ensure is that the clock has not ticked more often since its creation than specified by `max_delay` with `ts` as starting point for the measurement. If the timestamps match, the comparison is meaningful and we return either `Success true` indicating that the timestamp is valid, or `Success false` for an invalid timestamp. Otherwise, an error is returned stating that the timestamps do not match.

Because network delay describes the time it takes to transmit messages over the network, we define a clock tick in our model as the time it takes to send or receive a message. The clock counter therefore describes the number of sends and receives of messages containing the timestamp for which the time is measured by the clock. We only require honest principals to increment the clock and hence define another module `DS.SendRecv` in the Denning-Sacco model, in which we wrap the functions `send` and `receive_i` from `LabeledPKI` and simultaneously increment the clock counter. The adversary, on the other hand, does not perform a clock tick when sending or receiving messages, and thus can use the `send` and `receive_i` functions in `AttackerAPI` directly.

```
val send: (#i:timestamp) -> (c_in:clock) -> (sender:principal) -> (receiver:principal) ->
        (message:msg i public) -> LCrypto (si:timestamp & c_out:clock) (pki ds_preds)
    (requires (fun t0 -> i <= trace_len t0))
    (ensures (fun t0 (|si,c_out|) t1 ->
        si == trace_len t0 /\
        trace_len t1 = trace_len t0 + 1 /\
        was_message_sent_at (trace_len t0) sender receiver message /\
        clock_get c_out = (clock_get c_in) + 1))
```

The interface of the new `send` function in `DS.SendRecv` has an additional parameter `c_in` of type `clock`, the current clock before a message is sent. Similarly, the output type is changed to a dependent tuple consisting of the trace index `si` of the sent message, and additionally the new clock `c_out` after the message has been sent. The increment of the clock counter when sending a message is tracked by the additional clause `clock_get c_out = (clock_get c_in) + 1` in the `ensures` predicate of the `LCrypto` effect.

```
    val receive_i: (index_of_send_event:timestamp) -> (c_in:clock) -> (receiver:principal) ->
        LCrypto (now:timestamp & c_out:clock & sender:principal & msg now public) (pki
ds_preds)
        (requires (fun t0 -> True))
        (ensures (fun t0 (|now,c_out,sender,t|) t1 ->  t0 == t1 /\
            now = trace_len t0 /\
            index_of_send_event < trace_len t0 /\
            (exists sender receiver. was_message_sent_at index_of_send_event sender receiver t
) /\
            clock_get c_out = (clock_get c_in) + 1))
```

In the same way, we modify `receive_i`. The new function has an additional input parameter `c_in` and the dependent tuple returned by the function has an additional member `c_out`, both of type clock. The `ensures` predicate is again extended to track the clock counter before and after a message is received. To be able to increment the counter of a clock, the `DS.SendRecv` module needs access to the `clock` implementation. Therefore, we need to split the module into an interface and an implementation file, similar to `DS.Clock`, and then friend the `DS.Clock` module inside the implementation file.

Implications

The implication of the presented approach and its concrete implementation is that a communication key exchanged over a secure channel cannot be replayed under any circumstances if compromised later. This presupposes that we model the protocol steps such that the `send` and `receive_i` functions from `DS.SendRecv` are used to send and receive messages with timestamps and that proper validation of timestamps is performed. To still allow an adversary to carry out timely replay attacks, which are typically not prevented by the lax timeliness guarantees of timestamps, we do not assume that the adversary increments the clock counter when sending or receiving messages. In this way, the adversary can easily compromise principals before or during a protocol run and replay keys within the small time window in which timestamps are valid. However, if the attacker compromises and replays a previously securely exchanged key, the honest recipient will perform a clock tick, invalidating the corresponding timestamp and therefore rejecting the key. Hence, we have ideal time-based security implications in protocol runs of honest principals, without limiting the DY* attacker's capability to compromise principals and successfully replay keys in the validity window of the corresponding timestamps. We demonstrate the soundness of our approach in the following sections by proving that the timestamps in the Denning-Sacco protocol do indeed prevent replays of old compromised communication keys, and by modeling an attack on the protocol in which a compromised server issues a fake certificate containing a valid timestamp accepted by the initiator, allowing the adversary to break authentication and further replay the key in the name of the initiator to any other principal while the timestamp is still valid.

4.5.3 Modeling the Protocol in DY*

With the presented solution for modeling and proving time-based properties of the Denning-Sacco protocol or similar protocols, we are now able to develop a complete model of the Denning-Sacco protocol in DY*. We presuppose the message structure from Section 3.3.3 and use an architecture for the model similar to that of the Otway-Rees and Yahalom models.

Messages

Again, a module `DS.Messages` defines and implements types modeling the message structure and functions for serializing and parsing messages.

```
/// Format of signed message parts
noeq type sigval =
    | CertA: a:principal -> pk_a:bytes -> t:timestamp -> sigval
    | CertB: b:principal -> pk_b:bytes -> t:timestamp -> sigval
    | CommKey: ck:bytes -> t:timestamp -> sigval

noeq type message (i:nat) =
    | Msg1: a:principal -> b:principal -> message i
    | Msg2: cert_a:msg i public -> sig_cert_a:msg i public -> cert_b:msg i public ->
sig_cert_b:msg i public -> message i
    | Msg3: cert_a:msg i public -> sig_cert_a:msg i public -> cert_b:msg i public ->
sig_cert_b:msg i public -> enc_sig_ck:msg i public -> message i
```

While Otway-Rees and Yahalom used exclusively symmetric encryption – modeled as AEAD in their respective DY* models – the Denning-Sacco protocol uses different cryptographic primitives: a combination of digital signatures and public key encryption. Since public key encryption is only used to encrypt a signature at one point in the protocol, we do not need to model it as an additional sum type with multiple constructors. Howbeit, digital signatures are used in several places in the protocol and are modeled by the `sigval` type (remember that we had an `encval` type for encrypted values in the Otway-Rees and Yahalom models). Similar to `encval`, the type `sigval` reflects the structure of the message parts that appear in the form of signatures in the protocol messages. There are three signatures in total in the messages of the Denning-Sacco protocol. The two public key certificates $C_A = \{A, P_A, T\}^a_{S_{AS}}$ and $C_B = \{B, P_B, T\}^a_{S_{AS}}$ are issued by the authentication server for users A and B, and are constructed via `CertA a pk_a t`, and `CertB b pk_b t`, respectively, in the model. The third signature is the key certificate $\{k_c, T\}^a_{S_A}$ issued by A for B, which is additionally encrypted under B's public key because k_c is confidential. To create the key certificate, the constructor `CommKey ck t` may be used. Like in the other models, the message model is completed by a type `message i` for entire messages consisting of signatures, ciphertexts, and terms appearing in plaintext. In the Denning-Sacco protocol, there are three messages respectively constructed via the constructors `Msg1 a b` for the first message $A \longrightarrow AS : A, B$, `Msg2 cert_a sig_cert_a cert_b sig_cert_b` for the second message $AS \longrightarrow A : C_A, C_B$, and `Msg3 cert_a sig_cert_a cert_b sig_cert_b enc_sig_ck` for the third message $A \longrightarrow B : C_A, C_B, \{\{k_c, T\}^a_{S_A}\}^a_{P_B}$. The constructors for the second and third message accept two arguments for each of the two certificates, the plaintext and the corresponding signature or tag. Denning and Sacco do not differentiate between public and private keys for public key cryptography, and keys used for creating and verifying digital signatures. Instead, they simply define signing and verification as the inverse of encryption and decryption under the same key pair. For the Denning-Sacco protocol by itself, this does not impose a security risk, as neither party uses its public-private-key-pair for both, public key encryption and digital signatures. However, DY* only offers a secure symbolic implementation of signatures, which requires an extra key pair for signing and verifying messages, and where the plaintext is not part of the signature returned by the sign function. The verification algorithm, howbeit, requires the plaintext to verify a signature, assuming that the verifying party either already knows the signed term or is able to derive it (which is generally possible, since signatures by themselves should not contain confidential data). Since the server is in charge of public key distribution in the Denning-Sacco protocol, and users are not supposed to know the public keys of all other users in advance, we simply include the certificates in plaintext in the second and third message, such that the receiving party can easily verify the certificates and obtain the public keys and peer identities, as well as the timestamp.

```
    val encval_comm_key: i:nat -> ser_ck:bytes -> sig_ck:bytes -> l:label{
valid_encval_comm_key i ser_ck sig_ck l} -> enc_sig_ck:msg i l

    val parse_encval_comm_key: #i:nat -> #l:label -> enc_sig_ck:msg i l -> r:(result (bytes *
bytes))
        {
            match r with
            | Success (ser_ck, sig_ck) -> valid_encval_comm_key i ser_ck sig_ck l
            | _ -> True
        }
```

For the key certificate encrypted under the responder's public key in the last message, we introduce a function `encval_comm_key` that accepts as input the key certificate in plaintext form and in signed form, both of type `bytes` and valid according to `valid_encval_comm_key`. The values are concatenated, and the resulting `msg i l` is returned, which can then be encrypted. The corresponding parse function accepts a single value of type `msg i l` and if parsing succeeds, returns a tuple consisting of a valid key certificate in plaintext and its signature tag. Serialization and parsing of `sigval` and `message i` values is implemented similarly as in the Yahalom model.

State

We also once more define a state session type `session_st` in a module `DS.Sessions` to construct states for storing protocol relevant data.

```
    noeq type session_st =
        | InitiatorSentMsg1: b:principal -> session_st
        | AuthServerSentMsg2: a:principal -> b:principal -> session_st
        | InitiatorSentMsg3: b:principal -> ck:bytes -> session_st
        | ResponderRecvedMsg3: a:principal -> ck:bytes -> session_st
```

Since the Denning-Sacco protocol uses signatures and public key encryption, we use the API for private key generation and public key installation of `LabeledPKI`, which implicitly stores these keys in state sessions defined in its own `session_st` type. The `session_st` type in `DS.Sessions` is thus explicitly used for modeling protocol states and not for storing long-term keys. We have four constructors corresponding to four protocol steps in which the Denning-Sacco protocol can be divided. The constructor names contain again the role of the principal that stores the state session and can access it, and the context or protocol step in which it has been created. After sending the first message, the initiator creates and stores `InitiatorSentMsg1 b`, where `b` is the responder or the party receiving the key at the end of the protocol, respectively. Similarly, the authentication server constructs and stores `AuthServerSentMsg2 a b` after the second message, with `a` and `b` being the users (initiator and responder) for which public key certificates were issued. The initiator receives the certificates first and forwards them together with the communication key it generates, in the third message, after which it then stores `InitiatorSentMsg3 b ck`, where `b` is again the key recipient or responder and `ck` the communication key. At the end, the responder should be in a similar state and therefore stores `ResponderRecvedMsg3 a ck` for initiator `a`.

```
    let is_comm_key i b p q = is_aead_key ds_global_usage i b (join (readers [P p]) (readers [
P q])) "DS.comm_key"

    let valid_session (i:nat) (p:principal) (si vi:nat) (st:session_st) =
        match st with
        | InitiatorSentMsg3 b ck ->
            is_msg i ck (readers [P p]) /\
            (is_comm_key i ck p b \/ corrupt_id i (P auth_srv))
        | ResponderRecvedMsg3 a ck ->
            is_msg i ck (readers [P p]) /\
            (is_labeled i ck (join (readers [P a]) (readers [P p])) \/ corrupt_id i (P a) \/
            corrupt_id i (P auth_srv))
        | _ -> True
```

The smaller amount of state sessions in the Denning-Sacco model also leads to a simpler valid_session predicate. Since the first two state sessions only store principals or principal names, there is nothing to refine, and any state session constructed with their respective constructors is valid. For InitiatorSentMsg3 b ck, we require that the initiator p can read ck, and that ck is a communication key for the initiator and the responder b unless the authentication server is corrupted. Even though the server is not involved in the key exchange directly, it still plays an important role because it certifies the public key used to keep the communication key confidential between the initiator and the party the initiator wants to communicate with. The adversary can corrupt the server and claim that the public key of a principal controlled by the adversary belongs to the responder. The is_comm_key predicate in the Denning-Sacco model tests for bytes b and for principals p and q, whether b is an AEAD key labeled with the union of the labels readers [P p] and readers [P q]. The initiator must make sure that the key certificate can be decrypted with the private key corresponding to the public key that the server claims to be the responder's, thus joining the labels of its own private key (which it knows) and of the responder's private key, which is readers [P b] given that the server issues an honest certificate. If the server is corrupted and issues a wrong certificate for the responder's public key to the initiator, we cannot deduce the label of the corresponding private key and therefore not conclude that ck is a communication key for the initiator and responder. A state session ResponderRecvedMsg3 a ck is valid if the responder can read ck, and if its label is the union of readers [P a] and readers [P p], with initiator a and responder p, unless there is an arbitrary corrupted state session of the initiator or the server. The statements about the key's label in the respective state sessions are used – similarly as in the other analyses – to prove that the key is secret as long as the authentication server and the users are honest.

The valid_session predicate involves a principal auth_srv that is neither an input argument of the function nor stored in one of the state sessions. This is due to the problem that we cannot prove the secrecy of the communication key in the responder's final state for arbitrary authentication servers in DY*. As elaborated, the label of the communication key depends on the honesty of the server and we are not able to unambiguously associate a particular server with the key certificate for the communication key given the information it contains. This is certainly not a problem in reality, since the users must know the identity of the authentication server as well as the server's public verification key in advance, and impersonation by the attacker is not possible without corrupting the servers signing key. If the users accept fake certificates, they must originate from the server owning the verification key that the users used to verify them, and the respective signing key must have been corrupted by the adversary. Otherwise, the server is honest and the secrecy of the communication key depends on the authenticity of its origin, which can be assumed if the signing key is secret. To circumvent this problem, we restrict our model to a single authentication server, which we define statically in `DS.Messages`. In all places where the server would be declared as a variable, e.g. as an input argument of a function or as a variable bound to a quantifier in a predicate, we use the globally defined principal `auth_srv` instead. For completeness, we also include a model of the Denning-Sacco protocol in our repository that does not define the authentication server as a singleton, but instead allows arbitrary principals to play the role of the server. However, some of the security properties that we prove for the model presented here cannot be proved for the alternative model.

Events

```
let event_initiate (a b:principal) =
    ("initiate",[string_to_bytes a;string_to_bytes b])
let event_certify (a b:principal) (pk_a pk_b:bytes) (t:timestamp) (clock_cnt:nat) =
    ("certify",[string_to_bytes a;string_to_bytes b;pk_a;pk_b;nat_to_bytes 0 t;
                nat_to_bytes 0 clock_cnt])
let event_send_key (a b:principal) (pk_a pk_b ck:bytes) (t:timestamp) (clock_cnt:nat) =
    ("send_key",[string_to_bytes a;string_to_bytes b;pk_a;pk_b;ck;nat_to_bytes 0 t;
                 nat_to_bytes 0 clock_cnt])
let event_accept_key (a b:principal) (pk_a pk_b ck:bytes) (t:timestamp) (clock_cnt:nat) =
    ("accept_key",[string_to_bytes a;string_to_bytes b;pk_a;pk_b;ck;nat_to_bytes 0 t;
                   nat_to_bytes 0 clock_cnt])
```

For the Denning-Sacco protocol, we define four events. The first event is again the *initiate* event marking the start of a protocol run between the initiator `a` and the intended communication partner or responder `b`. When the server issues the public key certificates and sends them to the initiator as response to the first message, it triggers the *certify* event. With the *certify* event, we associate all the data contained in the two certificates, i.e. the initiator, the responder, their respective public keys `pk_a` and `pk_b`, and the timestamp `t`. After having received and verified the certificates and generated the communication key `ck`, the initiator triggers the event *send key* to indicate that it sends the public key certificates and a certificate for the communication key to the responder. The final protocol event is the *accept key* event, which the responder triggers when it has verified the server certificates and accepted the communication key it received from the initiator. The *certify*, *send key*, and *accept key* events, which include the server timestamp in their associated data, also capture the value of the clock counter at the time of each event, which we later use to prove that the key accepted by the responder is not a replay.

Protocol Specific Usage and Trace Predicates

As mentioned, we aim to provide the first elaborate security proof for the Denning-Sacco protocol with public keys based on our DY* model. In order to derive the statements in `valid_session` about the label of the communication key stored in the final state sessions of initiator and responder required to prove that the key is secret, and to make statements about relations of protocol events for proving time-based authentication and replay mitigation properties, we need to refine the data transmitted by messages and associated with events in the usage and trace predicates of our model. As for the data contained in messages, we can only refine data that is protected by some cryptographic primitive. Most of the data in the Denning-Sacco protocol is public knowledge whose integrity is protected by digital signatures. In addition, the serialized communication key certificate is protected by public key encryption for confidentiality. However, the parsed data is only available in the parsed certificate and therefore the signature predicate is sufficient to pass on statements about keys ans relations of events between protocol steps. Before we can implement the signature predicate, we have to define the network delays expected by the initiator when receiving the second message from the server and by the responder when receiving the last message from the initiator. Recall that the network delay specifies the validity window of a timestamp in terms of clock ticks since the time represented by the timestamp.

```
let recv_msg_2_delay:nat = 2
let recv_msg_3_delay:nat = recv_msg_2_delay + 2
```

We define two values: the maximum delay expected by the initiator when receiving the second message we denote by `recv_msg_2_delay` and set it to two clock ticks, one when the message is sent and one when it is received, and the delay expected by the responder when receiving the third message we denote by `recv_msg_3_delay` and set it to the sum of the expected delay of the second message plus two more clock ticks for sending and receiving the third message. Having defined the network delays for timestamp validation, we can implement the signature usage predicate of the Denning-Sacco protocol.

```
let can_sign (i:nat) s k ssv =
    exists p. get_signkey_label ds_key_usages k == readers [P p] /\
    (match parse_sigval_ ssv with
    | Success (CertA a pk_a t) ->
        (t+1) < i /\
        (exists b pk_b. did_event_occur_at (t+1) p (event_certify a b pk_a pk_b t 0))
    | Success (CertB b pk_b t) ->
        (t+1) < i /\
        (exists a pk_a. did_event_occur_at (t+1) p (event_certify a b pk_a pk_b t 0))
    | Success (CommKey ck t) ->
        i > 2 /\
        (exists b pk_b.
        was_rand_generated_before i ck (join (readers [P p]) (get_sk_label ds_key_usages
pk_b)) (aead_usage "DS.comm_key") /\
        (exists clock_cnt.
        clock_cnt <= recv_msg_2_delay /\ did_event_occur_at (i-3) p (event_send_key p b k
pk_b ck t clock_cnt)))
    | _ -> False)
```

The signature predicate accepts as inputs a natural number `i` representing the trace index at which the signature is created, which is also the trace index at which the predicate holds. The other input parameters are a usage string `s` belonging to the used signing key, the verification key `k` corresponding to the signing key, and the serialized `bytes` value `ssv` that is signed. The statement in the first line means that the signing key is secret to some arbitrary principal `p`, whose concrete identity depends on the key used for signing and the signed term. The rest of the predicate divides into three branches for the three different signatures appearing in the protocol messages. For the two server certificates `CertA a pk_a t` and `CertB b pk_b t` that confirm the key bindings (`a,pk_a`) and (`b,pk_b`), we require that the next timestamp after `t`, i.e., `t+1`, is smaller than the timestamp of the signature, and that `p`, being the authentication server, has triggered `event_certify a b pk_a pk_b t 0` at `t+1`, either for arbitrary responder `b` and public key `pk_b` in case of `CertA`, or for arbitrary initiator `a` and public key `pk_a` in case of `CertB`. Because events are logged in the trace, two different events cannot occur at the same trace index, so the two statements in the respective `match` arms for `CertA` and `CertB` refer to the same occurrence of the *certify* event at index `t+1`. This allows us to combine the two statements into one statement with concrete values for both key pairs, (`a`,`pk_a`) as well as (`b`,`pk_b`). The occurrence of the *certify* event is central to proving mutual authentication, although in a slightly weaker form than in the models of the improved Otway-Rees protocol or the Yahalom protocol. The third branch for `CommKey ck t` contains one helper clause ensuring that the index at which the signature was created is greater than two, which is relevant in a following clause. The core statements for the communication key certificate are, on the one hand, that the communication key `ck` must have been generated by `p` – the initiator – before trace index `i`, labeled with the union of `readers [P p]` and the secret key label of some arbitrary public key `pk_b` and intended as AEAD key between two parties who completed the Denning-Sacco protocol together, and on the other hand, that there exists some value for `clock_cnt` less than or equal to `recv_msg_2_delay` such that the initiator triggered `event_send_key p b k pk_b ck t clock_cnt` at index `i-3` (which is a valid index due to `i > 2`) for some arbitrary responder `b` owning `pk_b`. That `pk_b` is the public key of `b` is implied by the structure of the events, but we still need the event predicate to further refine the terms associated with events, including the values of `b` and `pk_b` in the *send key* event. For a secrecy proof of the communication key `ck`, we must be able to deduce the label of the secret key corresponding to `pk_b`.

```
let is_pub_enc_key i b p =
    is_public_enc_key ds_global_usage i b (readers [P p]) "DS.pke_key"
let is_ver_key i b p =
    is_verification_key ds_global_usage i b (readers [P p]) "DS.sig_key"

let epred idx s e =
    match e with
    | ("initiate",[a_bytes;b_bytes]) ->
        bytes_to_string a_bytes == Success s
    | ("certify",[a_bytes;b_bytes;pk_a;pk_b;t_bytes;clock_cnt_bytes]) -> (
        match (bytes_to_string a_bytes, bytes_to_string b_bytes, bytes_to_nat t_bytes,
               bytes_to_nat clock_cnt_bytes) with
        | (Success a, Success b, Success t, Success clock_cnt) ->
            s = auth_srv /\
            clock_cnt = 0 /\
            is_ver_key idx pk_a a /\ is_pub_enc_key idx pk_b b
        | _ -> False
    )
    | ("send_key",[a_bytes;b_bytes;pk_a;pk_b;ck;t_bytes;clock_cnt_bytes]) -> (
        match (bytes_to_string a_bytes, bytes_to_string b_bytes, bytes_to_nat t_bytes,
               bytes_to_nat clock_cnt_bytes) with
        | (Success a, Success b, Success t, Success clock_cnt) ->
            a = s /\
            clock_cnt <= recv_msg_2_delay /\
            (get_sk_label ds_key_usages pk_b == readers [P b] /\
            (t+1) < idx /\
            did_event_occur_at (t+1) auth_srv (event_certify a b pk_a pk_b t 0) \/
            corrupt_id idx (P auth_srv)) /\
            was_rand_generated_before idx ck (join (readers [P a]) (get_sk_label
ds_key_usages pk_b)) (aead_usage "DS.comm_key")
        | _ -> False
    )
```

```
        | ("accept_key",[a_bytes;b_bytes;pk_a;pk_b;ck;t_bytes;clock_cnt_bytes]) -> (
            match (bytes_to_string a_bytes, bytes_to_string b_bytes, bytes_to_nat t_bytes,
                   bytes_to_nat clock_cnt_bytes) with
            | (Success a, Success b, Success t, Success clock_cnt) ->
                b = s /\
                clock_cnt <= recv_msg_3_delay /\
                ((t+1) < idx /\
                did_event_occur_at (t+1) auth_srv (event_certify a b pk_a pk_b t 0) \/
                corrupt_id idx (P auth_srv)) /\
                ((exists clock_cnt'.
                clock_cnt' <= recv_msg_2_delay /\
                did_event_occur_before idx a (event_send_key a b pk_a pk_b ck t clock_cnt'))
/\
                was_rand_generated_before idx ck (join (readers [P a]) (readers [P b])) (
aead_usage "DS.comm_key") \/
                corrupt_id idx (P a) \/ corrupt_id idx (P auth_srv))
            | _ -> False
        )
        | _ -> False
```

Regarding the *initiate* event, the predicate ensures that the initiator is the one triggering it. For the *certify* event, we require that it is triggered by the authentication server, that the counter of the clock attached to the timestamp `t` is zero, since the event occurs before the server first sends the certificates to the initiator, and that the public key `pk_a` is the verification key of the initiator `a` and `pk_b` the public encryption key of the responder `b`. Based on the correspondence of the public keys, we can also determine the labels of the respective secret keys, like the secret key label of `pk_b` needed for the secrecy proof of `ck`. The *send key* event is again triggered by the initiator. The associated clock counter value must be less than or equal to the expected network delay for receiving the second message such that `t` is considered valid. Furthermore, we now have explicit that the secret key label of `pk_b` is `readers [P b]`, where `b` is the responder, and the server triggered `event_certify a b pk_a pk_b t 0` at trace index `t+1` before `idx`, the index of the *send key* event, unless an arbitrary server state session is corrupted. With the last statement, we ensure the same thing as in the signature predicate for the communication key certificate, that the communication key `ck` was generated by the initiator to be used as AEAD key by anyone with access to the private keys corresponding to `pk_a` and `pk_b`. The private key of an honest initiator is only known to the initiator itself; thus, we can assume that it is labeled with `readers [P a]`. The *accept key* event can only be triggered by the responder, and the clock counter at the time the event is triggered must be smaller than or equal to `recv_msg_3_delay`, which is essential for the replay mitigation proof. Beyond that, the responder is able to derive the same statement about the occurrence of the *certify* event triggered by the server, although we can omit the explicit statement about the secret key label of `pk_b`, which is known to an honest responder. Since the responder receives a certificate containing the communication key `ck` issued by the initiator along with the public key certificates from the server, it can further assume that the initiator has triggered `event_send_key a b pk_a pk_b ck t clock_cnt'` for some value of `clock_cnt'` less than or equal to `recv_msg_2_delay`, meaning that the responder can assume that the initiator trusts the correctness of the responder's public key binding. Although implicitly ensured by *send key*, we explicitly require that the label of the communication key is known to the responder as `join (readers [P a]) (readers [P b])` for honest initiator and server. The clauses in valid_session on the label of ck stored in the final state sessions of the initiator and responder are then derived directly from the statements in the event predicate regarding the events *send key* and *accept key*.

Protocol Steps

The Denning-Sacco protocol can be realized in four protocol steps in which the three protocol messages are exchanged and processed. We therefore follow the example of our previous analyses of Otway-Rees and Yahalom, and define and implement the protocol steps in a module DS.Protocol. Since DY* comes with the necessary functionality to generate and install keys for public key encryption or digital signatures, we do not require additional functions to set up the initial state of the principals in the protocol anymore.

First Step (Initiator) The first step in the Denning-Sacco protocol is run by the initiator, who sends the first message to the authentication server, requesting certificates for the public keys of the initiator and its intended communication partner, called “responder” for consistency.

```
val initiator_send_msg_1:
    a:principal ->
    b:principal ->
    LCrypto (msg_idx:timestamp * sess_idx:nat) (pki ds_preds)
    (requires fun t0 -> True)
    (ensures fun t0 (mi, si) t1 -> mi < trace_len t1 /\ trace_len t0 < trace_len t1)
```

The initiator is denoted by a and the responder by b. We model initial knowledge other than the private and public key sessions similarly as in the Yahalom model, by specifying them as direct input arguments to the respective protocol steps, which is here the case for the responder’s identity known to the initiator in advance. The first step returns a tuple consisting of the trace index of the first message and a session index pointing to the initial protocol state session of the initiator. The initiator initiates the protocol by triggering event_initiate a b, signaling that a protocol run between a and b has been started. Then, it creates the first message Msg1 a b consisting of the identities of initiator and responder in plaintext, and sends it to the authentication server. The first message functions as request for the public key certificates of the users, whose identities are included in the message. After sending the first message, the initiator then stores the responder’s identity in a new state session InitiatorSentMsg1 b in its protocol state and outputs the message index along with the index of the new state session.

Second Step (Server) In the second step, the server receives the first message with the user identities from the initiator and responds with public key certificates for the respective users in the second message.

```
val server_send_msg_2:
    msg_idx:timestamp ->
    LCrypto (msg_idx:timestamp * sess_idx:nat * c_out:clock) (pki ds_preds)
    (requires fun t0 -> msg_idx < trace_len t0)
    (ensures fun t0 (mi, si, c_out) t1 ->
        mi < trace_len t1 /\ trace_len t0 < trace_len t1 /\
        clock_get c_out = 1)
```

Since the server is defined as global instance in this version of the Denning-Sacco model, we do not need to specify it as input parameter to the protocol step. However, we need the trace index of the first message, in order for the server to fetch it from the trace. The function definition of the second step of the protocol is more interesting, because this is where the time-based properties come into play. Similar to the first step, the returned tuple contains the trace index of the sent message – the second message in the protocol – and the session index of the newly created state session stored in the server state, but in addition, the protocol step also outputs a clock `c_out`, which is attached to the timestamp included in the public key certificates from the server. We demand that `c_out` has ticked once since its creation, particularly when the server emitted the second message with the certificates. To demonstrate the usage of the clock API for modeling time-dependent properties of protocols, we put the implementation of the second protocol step below. However, we omit the usage of lemmata and assertions in this step, which have the sole purpose of assisting F* during the verification process.

```
    module SR = DS.SendRecv

    let server_send_msg_2 msg1_idx =
        // receive and parse first message
        let (|now,a,ser_msg1|) = receive_i #ds_preds msg1_idx auth_srv in

        match parse_msg ser_msg1 with
        | Success (Msg1 a' b) -> (
            if a <> a' then error "[srv_send_m2] initiator from received message does not
match with actual initiator"
            else
                // look up public keys of initiator and responder
                let pk_a = get_public_key #ds_preds #now auth_srv a SIG "DS.sig_key" in
                let pk_b = get_public_key #ds_preds #now auth_srv b PKE "DS.pke_key" in

                // obtain timestamp and initialize clock
                let (|c_new,t|) = clock_new #ds_preds auth_srv in

                // trigger event 'certify'
                let event = event_certify a b pk_a pk_b t (clock_get c_new) in
                trigger_event #ds_preds auth_srv event;

                // look up sign key of server and generate sign nonce
                let now = global_timestamp () in
                let (|_,sigk_srv|) = get_private_key #ds_preds #now auth_srv SIG "DS.sig_key"
in
                let (|_,n_sig|) = rand_gen #ds_preds (readers [P auth_srv]) (nonce_usage "
SIG_NONCE") in

                // create and sign initiator certificate
                let cert_a = CertA a pk_a t in
                let now = global_timestamp () in
                let ser_cert_a = serialize_sigval now cert_a public in
                let sig_cert_a = sign #ds_global_usage #now #(readers [P auth_srv]) #public
sigk_srv n_sig ser_cert_a in

                // create and sign responder certificate
                let cert_b = CertB b pk_b t in
                let ser_cert_b = serialize_sigval now cert_b public in
                let sig_cert_b = sign #ds_global_usage #now #(readers [P auth_srv]) #public
sigk_srv n_sig ser_cert_b in

                // create and send second message
                let msg2 = Msg2 ser_cert_a sig_cert_a ser_cert_b sig_cert_b in
                let ser_msg2 = serialize_msg now msg2 in

                let (|msg2_idx,c_out|) = SR.send #now c_new auth_srv a ser_msg2 in

                // create and store server session
                let new_sess_idx = new_session_number #ds_preds auth_srv in
                let st_srv_sent_m2 = AuthServerSentMsg2 a b in
                let now = global_timestamp () in
                let ser_st = serialize_session_st now auth_srv new_sess_idx 0 st_srv_sent_m2
in

                new_session #ds_preds #now auth_srv new_sess_idx 0 ser_st;

                (msg2_idx, new_sess_idx, c_out)
        )
        | _ -> error "[srv_send_m2] wrong message"
```

The server first fetches the first message from the initiator from the trace at `msg1_idx`, using the `receive_i` function from `LabeledPKI`. This returns, among other things, the identity of the sender as well as the serialized first message. Next, the first message is parsed, resulting in the high-level structure `Msg1 a' b`. The server checks if the initiator identity `a` returned by `receive_i` matches with the identity `a'` included in the message. If so, the server looks up the public keys of the initiator `a` and the responder `b`, which should be installed in its state in advance; otherwise, an error indicating a mismatch in the identity of the initiator is returned. After looking up the public keys, the server creates a new clock `c_new` for a timestamp `t` holding the value of the current trace length. Therefore, it uses the `clock_new` function that is part of the clock API provided by the module `DS.Clock`. Now that all information that will be included in the certificates is available, the server continues by triggering `event_certify a b pk_a pk_b t (clock_get c_new)`. `clock_get c_new` outputs the counter of the fresh clock, which is zero initially. Before it can create and sign the certificates, the server must first retrieve its private signing key `sigk_srv` from a respective state session stored in its state and generate a secret nonce `n_sig` used as randomness in the signing algorithm. Then, it creates the certificates `CertA a pk_a t` and `CertB b pk_b t`, serializes them, and finally signs them with `sigk_srv`. The server is now able to create the second message `Msg2 ser_cert_a sig_cert_a ser_cert_b sig_cert_b` containing the certificates for the initiator's and the responder's public key in serialized form only and in signed form so that the users can verify that the certificates really come from the trusted server. The second message is then serialized and sent back to the initiator. Howbeit, instead of the regular labeled `send` function from `LabeledPKI`, we use the respective wrapper function in the `DS.SendRecv` module, which accepts an additional parameter of type `clock` and increments its counter. The server passes the just created clock corresponding to the timestamp in the certificates as argument to the function to account for the symbolic network delay imposed by the sent message. The wrapped send function returns, in addition to the trace index of the sent message, also the resulting clock after incrementing the counter. Finally, the server creates a new state session `AuthServerSentMsg2 a b` containing the identities of the users for whom it just issued public key certificates, and stores it in its state. The function outputs the trace index of the second message, the index of the new state session, and the clock returned by the `SR.send` function.

Third Step (Initiator) In the third step of the protocol, the initiator validates the public key certificates from the authentication server and generates the communication key. The communication key is signed and encrypted, and then sent to the responder along with the public key certificates.

```
val initiator_send_msg_3:
    c_in:clock ->
    a:principal ->
    msg_idx:timestamp ->
    sess_idx:nat ->
    LCrypto (msg_idx:timestamp * c_out:clock) (pki ds_preds)
    (requires fun t0 -> msg_idx < trace_len t0)
    (ensures fun t0 (mi, c_out) t1 -> mi < trace_len t1 /\ trace_len t0 < trace_len t1 /\
                                    clock_get c_out = (clock_get c_in) + 2)
```

The third step takes four input parameters: the input clock, denoted by `c_in`, which must be the clock returned by the server step, the initiator `a`, the trace index of the second message from the server `msg_idx`, and the session index of the state session stored by the initiator at the end of the first step, denoted by `sess_idx`. Returned is a tuple of the message index of the third message sent in this step, and the output clock `c_out`. Again, the `ensures` predicate contains a property of the output clock, stating that the clock must have ticked twice during the protocol step. The first action performed by the initiator is to retrieve and parse the state session stored at `sess_idx` in the initiator state. If the parsed state session matches `InitiatorSentMsg1 b`, the initiator receives the second message stored at `msg_idx` on the trace using the `receive_i` function from `DS.SendRecv`, which increments the clock counter and returns, among other things, the message sender as `auth_srv'`, the serialized message as `ser_msg2`, and the resulting clock as `c_rm2`. The initiator checks that `auth_srv'` is the authentication server defined as global singleton in the `DS.Messages` module, and parses `ser_msg2` next. A message of the form `Msg2 cert_a sig_cert_a cert_b sig_cert_b` is expected, containing the public key certificates for the initiator `a` and the responder `b` in plaintext, as well as the respective signatures. The initiator must verify the signatures `sig_cert_a` and `sig_cert_b` to be sure that they really originate from the server. Therefore, it looks up the server's verification key, which it installed in its own state before initiating the protocol. If the signatures come from the server, the initiator parses the plaintext forms `cert_a` and `cert_b` of the certificates in order to validate its own certificate and to obtain the public key of the responder. The parsed certificates should be of the form `CertA a' pk_a t` and `CertB b' pk_b t'`. The initiator first validates the principal identities in the certificates by checking that `a = a'` and that `b = b'`. Moreover, it compares the timestamps `t` from the initiator certificate and `t'` from the responder certificate, which should be equal. After that, the initiator validates `t` by checking whether the clock `c_rm2` was obtained at `t` and has not ticked more often than specified by `recv_msg_2_delay`, the maximum expected network delay of the second message. If the certificate fulfills the timeliness requirements, the initiator first looks up its sign key `sigk_a`, and then uses the `vk` function from `LabeledCryptoAPI` to obtain the corresponding verification key `verk_a`. Given `verk_a`, it can finish the validation of its own certificate, by checking that `pk_a = verk_a`, i.e., that the public key included in the certificate by the server is really the verification key of the initiator. After the validation of the certificates is completed, the initiator can generate the communication key `ck`, which it labels with `join (readers [P a]) (get_sk_label ds_key_usages pk_b)`. Joining the known secret key label of the initiator's public key with the secret key label of pk_b is necessary so that the communication key certificate can later be encrypted under pk_b, as we need to ensure consistency between the readers of an encrypted message and the key used for encryption. Next, the initiator triggers event_send_key a b pk_a pk_b ck t (clock_get c_rm2) to signal that it is about to forward the public key certificates and ck to the responder. The initiator then generates two randomnesses for signing and for encryption, respectively. Now it can create the third message, for which it first creates the communication key certificate CommKey ck t with t being the same timestamp as in the public key certificates. The certificate for ck is then serialized and signed with sigk_a, resulting in ser_comm_key and sig_comm_key. Afterwards, the responder concatenates the resulting values with the encval_comm_key function from DS.Messages, which yields the final certificate and encryption value ev_comm_key. The certificate is then encrypted under pk_b, which results in the ciphertext c_comm_key. The initiator now creates and sends the third message Msg3 cert_a sig_cert_a cert_b sig_cert_b c_comm_key to b, containing the public key certificates and the encrypted communication key certificate. Finally, the initiator extends its state session from the first step with ck to be able to use it later for communication with b, resulting in an updated state session of the form InitiatorSentMsg3 b ck.

Fourth Step (Responder) It follows the last protocol step, in which the responder receives the public key certificates along with the communication key from the initiator and decides whether to accept the key based on the timeliness guarantees provided by the timestamps in the certificates.

```
val responder_recv_msg_3:
    c_in:clock ->
    b:principal ->
    msg_idx:timestamp ->
    LCrypto (sess_idx:nat * c_out:clock) (pki ds_preds)
    (requires fun t0 -> msg_idx < trace_len t0)
    (ensures fun t0 (si, c_out) t1 -> trace_len t0 < trace_len t1 /\
                                clock_get c_out = (clock_get c_in + 1))
```

The protocol step accepts input parameters c_in, the input clock, b, the responder, and msg_idx, the trace index of the third message. Outputs are the session index of a new state session stored in the responder's state and the output clock c_out with the clock counter advanced by one. The responder proceeds as follows: it first fetches the third message sent by the initiator from the trace, again using the wrapped receive_i function defined in DS.SendRecv. From the function, the responder obtains the serialized message ser_msg3 along with a clock c_out, which is the input clock advanced by one clock tick, and the sender identity a. Next, the responder parses ser_msg3 to obtain the third message Msg3 cert_a sig_cert_a cert_b sig_cert_b c_comm_key sent by the initiator in the third protocol step. In order to verify the public key certificates, the responder retrieves the server's verification key verk_srv, which should be installed in its state. Given verk_srv, the responder verifies the signatures sig_cert_a and sig_cert_b, and parses the respective plaintexts cert_a and cert_b, in case the verification was successful. The parse function returns the two certificates CertA a' pk_a t and CertB b' pk_b t'. The responder validates and checks both certificates in much the same way as the initiator did in the previous step, by verifying the identities a' and b' as well as the timestamps t and t' in the certificates, and also by verifying that pk_b is actually the responder's public key based on its secret key sk_b and the pk function from LabeledCryptoAPI returning the corresponding public key. After verification, the responder continues by decrypting the encrypted communication key certificate c_comm_key with sk_b. This results in a bytes value ev_comm_key that can be split into the serialized form ser_comm_key and signed form sig_comm_key of the certificate. Again, signature verification is performed, this time of sig_comm_key using pk_a. If the communication key turns out to be correct, the responder parses the corresponding plaintext ser_comm_key. The parsed certificate is expected to have the form CommKey ck t''. The timestamp t'' is supposed to be equal to the timestamps in the public key certificates; the responder therefore checks that t = t''. Now that the responder knows that all timestamps in the certificates are equal, it can validate them with a single call to clock_lte, checking that c_out was obtained at t and that the value of the clock counter is at most recv_msg_3_delay. Afterwards, the responder triggers event_accept_key a b pk_a pk_b ck t (clock_get c_out), signaling that it trusts the public key binding of the initiator and accepts the communication key ck. The timestamp t and clock counter of c_out associated with the event can be used to prove that the key accepted by the responder is not a replay if none of the parties involved in the protocol have been corrupted. Finally, the responder stores a state session ResponderRecvedMsg3 a ck – the equivalent to the updated state session of the initiator – in its state, providing it with the ability to use ck for communication with a at some later time.

Execution

Similar to the models of the other protocols, the Denning-Sacco model can also be executed in a benign environment with the protocol steps scheduled in the intended order and without the adversary actively interfering in the execution. This again allows us to verify that our model is correct based on the trace resulting from execution. Since we have modeled time-based properties of a protocol for the first time in DY*, we further demonstrate the correctness of our implementation regarding attacker capabilities with an attack of a corrupted server breaking authentication. All the executable functions of the Denning-Sacco protocol are implemented in a module DS.Debug. The benign adversary is once more simulated via a function benign_attacker:

```
let benign_attacker () =
    let a:principal = "alice" in
    let b:principal = "bob" in

    let now = global_timestamp () in
    let sigk_a_idx = gen_private_key #ds_preds #now a SIG "DS.sig_key" in
    let now = global_timestamp () in
    let sk_b_idx = gen_private_key #ds_preds #now b PKE "DS.pke_key" in
    let now = global_timestamp () in
    let sigk_srv_idx = gen_private_key #ds_preds #now auth_srv SIG "DS.sig_key" in

    let now = global_timestamp () in
    let pk_a_srv_idx = install_public_key #ds_preds #now a auth_srv SIG "DS.sig_key" in
    let now = global_timestamp () in
    let pk_b_srv_idx = install_public_key #ds_preds #now b auth_srv PKE "DS.pke_key" in
    let now = global_timestamp () in
    let verk_srv_a_idx = install_public_key #ds_preds #now auth_srv a SIG "DS.sig_key" in
    let now = global_timestamp () in
    let verk_srv_b_idx = install_public_key #ds_preds #now auth_srv b SIG "DS.sig_key" in

    let (msg1_idx, a_sess_idx) = initiator_send_msg_1 a b in
    let (msg2_idx, srv_sess_idx, c_sm2) = server_send_msg_2 msg1_idx in
    let (msg3_idx, c_sm3) = initiator_send_msg_3 c_sm2 a msg2_idx a_sess_idx in
    let (b_sess_idx, c_end) = responder_recv_msg_3 c_sm3 b msg3_idx in
    ()
```

Since we have defined the authentication server as central instance in `DS.Messages`, we must only initialize two principals `a` and `b`, where `a` will take the role of the initiator, and `b` that of the other party. We then set up the initial knowledge by generating private signing keys for `a` and the server, and a private decryption key for `b`. The public keys of the two users `a` and `b` are installed in the state of the server such that it can issue certificates for these later. Also, the verification key of the server is installed in the user's state for verification of the certificates. Having installed public keys, we can then schedule the four protocol steps in the intended order, beginning with the first step executed by the initiator `a`. In addition to the message and session indices that were also present in the previous models, we additionally have to pass the clocks correctly through the protocol steps if we want the protocol to succeed. The last step returns not only a session index because the responder `b` has not been involved in the protocol before, but also a final clock `c_end` that would have to be passed to further protocol steps in case messages containing timestamps from this protocol session are replayed. If an honest principal receives such a message, it increments the clock counter and hence rejects the message when validating the timestamp. The adversary, on the other hand, shall still be able to carry out attacks in which it compromises principals upfront and replays messages immediately within the validity window of the timestamps. For this reason, we do not require the attacker to perform a clock tick when sending or receiving messages with the attacker API. To demonstrate that the adversary has the capabilities to carry out attacks, even when clocks are involved, we implement a second scheduler function `fake_cert_attacker`, in which a corrupted server breaks authentication by emitting a false certificate that is accepted by the initiator.

```
let fake_cert_attacker () =
    let a:principal = "alice" in
    let b:principal = "bob" in
    let e:principal = "eve" in

    // perform key generation and installation for alice, bob, eve and the server
    // compromise eve's secret decryption key 'sk_e'
    // compromise server's secret sign key 'sigk_srv'
    // query public key's of eve 'pk_e' and alice 'pk_a' stored in the server state

    let (msg1_idx, a_sess_idx) = initiator_send_msg_1 a b in
    let (|msg2_idx,c_sm2|) = attacker_issue_fake_cert e sigk_srv pk_a pk_e msg1_idx in
    let (msg3_idx, c_sm3) = initiator_send_msg_3 c_sm2 a msg2_idx a_sess_idx in
    let (|t_ck,ck|) = attacker_recv_msg_3 e sk_e msg3_idx in

    attacker_knows_comm_key_stored_in_initiator_state a a_sess_idx ck
```

We have left out the steps in the implementation that set up private and public keys and in which the attacker compromises and queries the keys it requires for the attack. The keys for `a`, `b` and the server are set up in the same way as in `benign_attacker`. The principal `e` will be controlled by the attacker and is used to impersonate `b`. Therefore, it gets private and public keys for public key encryption, similar to `b`. Public key installation is then performed just as in `benign_attacker`. At this point, the server has not been compromised yet; we thus install the public key of `e` in the server state as well. The adversary then compromises the secret decryption key of `e`, the server's secret sign key, and the state sessions of the server storing the public keys of `a` and `e`. Now the attacker is ready to carry out the attack. The protocol starts regularly with the initiator a requesting public key certificates for itself and b from the server. The adversary takes this chance and issues a false certificate for b to a, in which it claims that pk_e is the public key of b. Since a trusts the server, it accepts the certificate and thus falsely believes b to be the only one that is able to decrypt the communication key when encrypting it under pk_e. attacker_issue_fake_cert emits, besides the trace index of the second message containing the false certificate, also a clock c_sm2, obtained by the adversary via the function att_clock_new from the DS.Clock module. The attacker does howbeit not increment the clock counter when sending the message. This is not noticed by the initiator, because it only verifies that the time elapsed since the timestamp in the certificates is not greater than the maximum delay specified by recv_msg_2_delay. The initiator hence generates a communication key, signs it with its own private sign key, and encrypts it under pk_e, believing that this is the public key of b. During the protocol step of the initiator, the clock counter of c_sm2 is incremented twice, resulting in a new clock c_sm3. In the last step, the attacker receives the message and is able to decrypt the encrypted communication key certificate using sk_e, which it knows because it controls e. The decryption already reveals the plaintext form of the certificate containing the communication key ck, which is easily obtained by the adversary. As attacker_recv_msg_3 does not accept an input parameter of type clock, c_sm3 is already the final clock resulting from this particular attack, which has only ticked twice during the third step run by the initiator. Since the adversary knows ck and has never ticked the clock, it should be able to replay the key in the name of a to some arbitrary party, e.g., b, who will not notice the replay. Both scheduler functions are wrapped in functions that print the trace after running the protocol, and are called in the main function of the Denning-Sacco model in DS.Debug.

4.5.4 Correctness and Coherence of the Model

The individual modules of the Denning-Sacco model have been successfully verified in DY* and have been compiled to an executable executing the main function in DS.Debug. In particular, the model executes two scheduler functions, one simulating a benign attacker, hence executing a regular protocol run, and one where the attacker compromises the server's sign key and issues a false certificate to break authentication. The output and trace resulting from a regular protocol run are depicted in Appendix C.1. It shows a successful protocol run that leads to a communication key being exchanged and stored in the states of the initiator and responder, and the clock count at the time the responder accepts the key being equal to recv_msg_3_delay, meaning that the timestamp in the certificates is valid, which also shows that the time-dependent properties of the protocol are modeled correctly if the protocol steps are executed in the intended order. Appendix C.2 shows the output and trace of the second scheduler function executing the *fake certificate* attack. The trace shows that the attack is carried out successfully by the adversary, who is able to get the initiator to encrypt the certificate for the communication key with a public key whose corresponding private key is controlled by the attacker, while the initiator believes to share the key with another user. Thus, our implementation for capturing time-based protocol properties does not restrict the attacker in its capabilities to carry out attacks that are not prevented by the lax timeliness guarantees of timestamps. As the trace further shows, this also includes attacks in which the communication key is replayed timely. In our example attack, the clock counter is 1 when the initiator triggers *send key* and is incremented to 2 when it sends the message containing the key. The clock counter is therefore 2 at the time the adversary receives the key. If the adversary replays it to an honest user, the final clock counter will be 3, which is still lower than recv_msg_3_delay – the user will hence accept the key. The attack thus demonstrates that the time-dependent properties of our model correctly integrate with the DY* attacker model and hold in the presence of arbitrary adversaries. The coherence of the developed model with the actual protocol as described in Section 3.3.3 can be reasoned similarly to the previous analyses, but slightly different in that we have timestamps instead of nonces and an additional restriction regarding the principals involved in the protocol:

- Defining the authentication server as a global singleton in the model does not constrain the attacker's capabilities and thus does not diminish the model's power. The fact that the authentication server issues signed certificates to assure the users of the authenticity of the contained data makes it impossible for the adversary to impersonate the server without compromising its secret sign key. In DY*, we are not yet able to link the communication key certificate to the public key certificates from the server without additional information in the former. However, this correspondence is implied by the message structure of the protocol, where the initiator receives and validates the certificates from the second message, chooses a communication key, and then sends the certificates along with a third certificate for the communication key to the party with whom it wants to establish a key. This message exchange will not succeed unless both parties in the key exchange agree on the identity of the authentication server, and thus use the same verification key to check the public key certificates. In our model, we have realized this by encapsulating this behavior in a single protocol step `initiator_send_msg_3`, which models an honest initiator performing exactly the steps described above. We therefore argue that this restriction regarding the authentication server is necessary because there is no way to explicitly express this property of the Denning-Sacco protocol in DY*, not because the protocol is actually insecure.
- The initial knowledge of principals is correctly represented in the model. We took an approach similar to that used in the Yahalom model for representing initial knowledge, where we had established long-term key sessions prior to the protocol and modeled other terms, such as principal identities, as direct inputs to the respective protocol steps. In the Denning-Sacco protocol, we first generate private keys for each of the principals involved in the protocol according to their designated roles, store them in respective state sessions of these principals, and then install the corresponding public keys in state sessions of their initial peers. In particular, the server – as certification authority for public keys – stores all the users' public keys in its state and can certify them as requested based on the identities of the corresponding users given in the first message. The users, on the other hand, must only know the authentication server and its verification key in order to verify the certificates. To start a protocol run, the initiator must also know the identity of the party with whom it wishes to exchange a key in order to request a corresponding public key certificate from the server. As we have already argued with respect to the Yahalom protocol, there is no difference in modeling knowledge intended for the public, speaking principal names for example, as direct input to protocol steps, or in an additional state session for which we need to pass the session index as an input argument to the protocol step.
- The goals the authors had in introducing timestamps to the Denning-Sacco protocol also apply to the model, and the implementation for modeling time-based properties of protocols comes with strict semantics. The main purpose of the timestamps is to prevent replays of old compromised session keys. While in reality the timeliness guarantees of timestamps are lax, since replays are possible in a short window of time when the timestamps are valid, they are clear in the model. In fact, our model prevents the reuse of any key exchanged in a

protocol run of honest principals, while accounting for the fact that compromise can also take place prior to or in the middle of a protocol run. In case of an honest protocol run, the clock attached to the timestamp in the certificates ticks exactly as often as specified by the maximum expected network delay. If one or more compromised principals are involved, they use the attacker API, which does not perform clock ticks when sending or receiving messages, creating a symbolic validity window for the timestamp that allows the adversary to replay the exchanged key. Another property of the Denning-Sacco protocol based on timestamps is the lack of a handshake between initiator and responder. Since the message structure of the protocol directly transfers to our model, the handshake is not present in the model either. Considering that DY* only offers a symbolic indicator of time with the length of the global trace, there is no way to accurately model the real-world properties of the protocol based on timestamps in the model. However, our model defines clear semantics for timestamps that ultimately serve the same goals, to prevent replays of old compromised keys and to eliminate an additional handshake message.

- The origin of keys in the model matches with the origin defined in the protocol description. Private and public keys are set up using the API provided by DY*; the coherence of private and public key origins in the model is thus ensured by DY*, not by the model we have developed. The communication key k_c is signed in the third message by the initiator using its private signing key, and therefore must be generated by the initiator in the third protocol step.
- Labels of keys in the model correctly represent their intended audience. Intuitively, a private key is supposed to be a secret to the principal who generates it. Therefore, the private sign key of the initiator A (denoted by `a` in respective protocol steps), the decryption key of the responder B (denoted by `b`), and the sign key of the authentication server (denoted by `auth_srv` in the model), are labeled, respectively, `readers [P a]`, `readers [P b]`, and `readers [P auth_srv ]`. The corresponding public keys are labeled `public` as defined in DY*. When A generates the communication key in the third protocol step, it labels it with the union of the private key labels of its own private key – which it knows to be `readers [P a]` – and the private key of B – which is `readers [P b]`, unless the server has been compromised and issued a false public key certificate for B. As explained, A must trust the authentication server here, because only then can it use B's supposed public key to encrypt the communication key.
- `DS.Messages` correctly depicts the message structure of the protocol. The high-level message structure of the Denning-Sacco protocol is modeled by the types `sigval` and `message i`. `sigval` represents signed message parts, and includes the two public key certificates `CertA` for the initiator and `CertB` for the responder, issued by the authentication server and thus signed with S_{AS} (`sigk_srv`), as well as the communication key certificate `CommKey`, which the initiator signs with S_A (`sigk_a`). The communication key certificate is additionally encrypted under P_B, but since this is the only encryption performed in the protocol, we do not need an additional type for encrypted message parts. `message i` then models the top-level message structure including the signed and encrypted message parts, and parts sent in plaintext form. A notable characteristic of our model is that we model the certificates in the protocol as pairs of the data to be signed in plaintext form and the corresponding signature tag. This is necessary, since the description of the Denning-Sacco protocol simplifies signing and verifying as the inverse of encryption and decryption under the same key pair, while DY* demands that different keys are used for public key encryption than for digital signatures. Furthermore, the symbolic verification algorithm provided by DY* does not output the contents of the signature, but

instead takes the plaintext data as input and verifies that the signature tag corresponds to the plaintext, returning `true` if the signature is valid or `false` otherwise. Since the keys used in the Denning-Sacco protocol only serve a single purpose anyways and therefore do not impose a security risk, the semantics of both approaches are equivalent.

- `DS.Sessions` correctly models the protocol state and outcome. We again have a type `session_st` modeling the states of principals over the course of a protocol run. Since the Denning-Sacco protocol does not rely on nonces for timeliness guarantees and the authentication server is defined as a central instance in the model, the state sessions of the initiator and responder only store the exchanged communication key and the respective identity of the key peer. Based on the key peer stored in the state session, the user can look up the session and retrieve the key at any later time, allowing them to communicate with the peer. The fact that the server records the identities of users requesting certificates in the certification step has no particular impact on the completeness or security of the protocol and could therefore be omitted. It was simply included for consistency with the other models, where the server stored data required in security proofs.
- The protocol steps in `DS.Protocol` perform proper checks and a proper verification of the contents of messages. Like in the other protocol models, the identity of the message sender is verified based on the information provided in messages or in the context of the protocol steps, which does not restrict the attacker in its ability to spoof the sender. In the third and fourth steps, the initiator and responder respectively validate their own certificate by verifying their identity and the public key contained in the certificate, and they also verify that the other certificate belongs to the intended or expected communication partner. Validating the contents of the public key certificates is deliberate according to the protocol description. Further, the timestamps in the certificates are validated based on the current clock to ensure the timeliness of the messages. It was clarified in Section 3.3.3 that the two public key certificates and the communication key certificate use the same timestamp. Therefore, at each protocol step involving messages with timestamps, we first check that the timestamps are identical, which has the added benefit that we only need to validate one of the timestamps instead of each timestamp separately. Although Denning and Sacco only mention that timestamps should be validated based on the current clock, we reason that the fact that they are supposed to be the same timestamp is sufficient to justify the way we implemented timestamp validation. Finally, there is one more check in the last protocol step, where the initiator A makes sure that A and B are not the same user, which is a property required by F* for the protocol step to pass the type check. Since a key shared with oneself has no actual use, there is no harm in establishing such a property in the model.

4.5.5 Security Properties

This section comprises a detailed security proof of the Denning-Sacco protocol with public keys based on the model developed in DY*. As in the analyses of Otway-Rees and Yahalom, we proceed by formulating security properties of the model in DY*, which we derive from the protocol properties and goals described in Section 4.5.1. For this purpose, we extend the model by an additional module `DS.SecurityProps`, where we define and prove the security properties. This module is also subject to verification in F* to ensure that the security properties and their respective proofs are sound. We divide the protocol's security goals into goals for the exchanged key and authentication goals. The goals of the Denning-Sacco protocol regarding the exchanged key are captured by the following objectives:

(K1) the exchanged communication key is kept secret between the initiator and the responder;

(K2) the communication key accepted by the responder originates with the initiator; and

(K3) the responder will only accept a fresh key, i.e. one generated in the same protocol run.

Likewise, the authentication goals can also be broken down into different objectives:

(A1) when an initiator A sends a communication key k_c to a responder B who accepts the key, there is a corresponding protocol run with public key certificates from the authentication server that guarantee B that the key originates with A, and A that only B will obtain k_c; and

(A2) B is convinced that A has recently sent a message containing k_c to B.

Properties of the Exchanged Key

The Denning-Sacco protocol differs from the Otway-Rees or Yahalom protocols in that the authentication server is not responsible for key distribution itself, but only certifies the public keys needed by users to exchange keys securely over a possibly insecure network. For this reason, the objectives of the Denning-Sacco protocol regarding the exchanged key do not involve the authentication server at all. Otherwise, the goals for the key are similar to the secrecy goals of the other protocols. The unusual structure of the protocol, which omits a handshake message between the users, also requires that we choose a different label for the communication key than in the other protocols, in particular combining the secret key labels of the initiator and the responder. The secrecy properties we aim to prove regarding the objectives K1 and K2 of the Denning-Sacco protocol can be formulated similarly as the properties for the objectives S1 and S2 of the Otway-Rees or Yahalom protocols. However, we need to take into account the different role of the authentication server, and we need to adapt the proof slightly, considering that the key is labeled with the union of two separate labels. Specifically, this means that we again prove the secrecy goals K1 and K2 by showing that the key stored in the final state sessions of the initiator and the responder remains unknown to the attacker unless one of the principals involved in the protocol has been corrupted. The resulting lemmas in the Denning-Sacco model are similar to those defined in Section 4.3.6, which establish the secrecy of the key stored in the initiator and responder states in the improved Otway-Rees protocol. There, the validity of the secrecy properties was proved using the `secrecy_lemma`. Recall that the secrecy lemma ensures that a term labeled with a `readers` label is unknown to the attacker unless one of its readers is corrupted. For the Denning-Sacco protocol, we need the `secrecy_join_label_lemma` instead, which proves the same property, but for a term labeled with the union of two `readers` labels. In Section 4.5.3, we explained how we modeled the stateful parts of the protocol. This includes the `valid_session` predicate, which ensures the validity of protocol related state sessions and refines the terms stored in them. For the final state session of the initiator and responder, the predicate ensures that the key stored in the respective state session is labeled with `join (readers [P a]) (readers [P b])` for initiator `a` and responder `b`, or that state sessions of the authentication server or the initiator have been compromised. The secrecy lemma for `join` labels thus applies to the label of the communication key in the initiator and responder state sessions, and proves the properties we sought to ensure.

The objective K3 is a property of the key that depends on the timestamps in the Denning-Sacco protocol. Denning and Sacco proposed the usage of timestamps in key distribution protocols [16] mainly with the goal to prevent replays of old compromised session keys, which they see as a serious threat in such protocols. In this thesis, we presented a solution that allows us to model time-based properties in the DY* framework so that we can develop models of protocols that make use of timestamps to receive timeliness guarantees, such as the Denning-Sacco protocol, and prove their security, including time-dependent security properties, in DY*. We prove that the security objective K3 holds for the communication key in the Denning-Sacco protocol via a lemma `responder_comm_key_is_not_replay` in `DS.SecurityProps`.

```
    val responder_comm_key_is_not_replay: i:nat ->
    LCrypto unit (pki ds_preds)
    (requires (fun t0 -> i < trace_len t0))
    (ensures (fun t0 _ t1 -> forall a b pk_a pk_b ck t clock_cnt.
        did_event_occur_at i b (event_accept_key a b pk_a pk_b ck t clock_cnt)
        ==> clock_cnt <= recv_msg_3_delay /\
            (did_event_occur_at (t+1) auth_srv (event_certify a b pk_a pk_b t 0) \/ corrupt_id
 i (P auth_srv))))
```

The lemma receives an input argument `i`, which is required to be a valid trace index in the `requires` predicate. The property guaranteed is that when any responder `b` accepts a communication key `ck` from any initiator `a`, signaled by an event `event_accept_key a b pk_a pk_b ck t clock_cnt` at trace index `i` for some arbitrary timestamp `t` and clock counter value `clock_cnt`, then `clock_cnt` does not exceed the expected delay for receiving the third message and the authentication server has triggered `event_certify a b pk_a pk_b t 0` for the same timestamp `t` at trace index `t+1`, or is corrupt. For an honest server, this means that the timestamp `t` was obtained by the server and associated with a fresh clock during the public key certification step. When the responder accepts `ck`, the public key certificates from the server and the communication key certificate from the initiator contain this particular timestamp `t`, and the associated clock has ticked at most `recv_msg_3_delay` times. This and the correspondence of the initiator's verification key `pk_a` in both events lead to the conclusion that `ck` is a product of the current protocol run and originates with the initiator `a`. A dishonest `a` could still replay a compromised key, and a compromised server could issue a certificate with a false public key `pk_a`, but these are cases not targeted by Denning and Sacco with their proposal of timestamps for replay mitigation. By verifying the `responder_comm_key_is_not_replay` lemma in F*, we thus prove that objective K3 is satisfied in the Denning-Sacco protocol.

Authentication

The use of signatures combined with public key encryption instead of symmetric encryption and timestamps instead of nonces, and the omission of a handshake message from the responder to the initiator to confirm the nonces, leads to very different authentication properties from those of the other protocols analysed. Denning and Sacco argue that the handshake becomes obsolete with the timestamps, since both users get timeliness guarantees from the timestamps even without the handshake message. However, the timestamps do not provide locality guarantees like the nonces, and without the handshake, the initiator has no confirmation that the responder was online and received the communication key until the responder first sends a message encrypted with the key. Because of this, the Denning-Sacco protocol cannot provide the same level of strong mutual authentication as, say, the Yahalom protocol. Authentication objective A1 basically describes how users authenticate each other in the Denning-Sacco protocol. We express and prove this objective in our DY* model via a single lemma `mutual_authentication`.

```
    val mutual_authentication: i:nat -> j:nat ->
    LCrypto unit (pki ds_preds)
    (requires (fun t0 -> i < trace_len t0 /\ j < trace_len t0 /\ i < j))
    (ensures (fun t0 _ t1 -> forall a b pk_a pk_b ck t clock_cnt_a clock_cnt_b.
        did_event_occur_at i a (event_send_key a b pk_a pk_b ck t clock_cnt_a) /\
        did_event_occur_at j b (event_accept_key a b pk_a pk_b ck t clock_cnt_b)
        ==> (t+1) < i /\ did_event_occur_at (t+1) auth_srv (event_certify a b pk_a pk_b t 0)
\/ corrupt_id i (P auth_srv)))
```

We again have two inputs of type `nat`, valid trace indices `i` and `j`, so `i < j` holds. For two related events, `event_send_key a b pk_a pk_b ck t clock_cnt_a` at trace index `i` and `event_accept_key a b pk_a pk_b ck t clock_cnt_b` at trace index `j`, we have that `(t+1) < i` and that the authentication server triggered `event_certify a b pk_a pk_b t 0` at `t+1`, or was otherwise corrupted. This property holds for arbitrary values associated with the events, and ensures that if an honest server issues valid certificates for public keys `pk_a` and `pk_b` of users `a` and `b`, then any key `ck` exchanged using these public keys and corresponding secret keys is guaranteed to be actually shared between `a` and `b`, which means that any message received by `a` or `b` encrypted under `ck` is guaranteed to come from `b` or `a`, respectively, as long as neither of them leaks the key. So we have that `b` is convinced that `ck` originates from `a`, and likewise `a` is convinced that no one other than `b` has obtained `ck`. We verify the lemma in F* and thus prove that authentication objective A1 regarding mutual authentication is satisfied by the Denning-Sacco protocol. Though, as mentioned, mutual authentication here does not include `b` confirming to `a` that it was online and accepted the key, because it simply never does during the protocol. Receipt of the session key is only confirmed by `b` when it encrypts a message for `a` under `ck`, but this is not part of the protocol itself and is therefore not captured by the authentication property we proved.

Since the initiator generates and sends the communication key to the responder, the responder gets stronger guarantees from the protocol than the initiator. These stronger guarantees are captured in the authentication goal A2, which is formulated in another lemma `initiator_authentication` in our DY* model.

```
    val initiator_authentication: i:nat ->
    LCrypto unit (pki ds_preds)
    (requires (fun t0 -> i < trace_len t0))
    (ensures (fun t0 _ t1 -> forall a b pk_a pk_b ck t clock_cnt.
        did_event_occur_at i b (event_accept_key a b pk_a pk_b ck t clock_cnt)
        ==> (exists clock_cnt'. clock_cnt' <= recv_msg_2_delay /\
            did_event_occur_before i a (event_send_key a b pk_a pk_b ck t clock_cnt') \/
            corrupt_id i (P a)) /\
            (t+1) < i /\ did_event_occur_at (t+1) auth_srv (event_certify a b pk_a pk_b t 0)
\/
            corrupt_id i (P auth_srv)))
```

The initiator authentication lemma states that each event `event_accept_key a b pk_a pk_b ck t clock_cnt` of the responder `b` at some valid trace index `i` is preceded by a recent corresponding event `event_send_key a b pk_a pk_b ck t clock_cnt'`, where `clock_cnt'` is at most `recv_msg_2_delay`, triggered by the initiator `a` if it is not corrupt, and an event `event_certify a b pk_a pk_b t 0` at t+1 with (t+1) < i triggered by the authentication server if it is not corrupt either. The matching principals and public keys in the events *send key* and *certify*, as well as the events *accept key* and *send key*, inform b that a has successfully validated its own key binding and also trusts the key binding of b, both of which have been certified by the server. This, the matching communication key ck in the *accept key* and *send key* events, and the fact that the clock counter does not exceed the expected maximum delay of the second message, confirms to b that it was indeed a who recently initiated communication with b and thus sent the third message containing ck destined for b. In that sense, the initiator authentication property in the Denning-Sacco protocol is about as strong as that in the Yahalom protocol without the explicit locality guarantees of the nonces. However, as mentioned above, mutual authentication in the Denning-Sacco protocol is more lax and only becomes strong when messages are actually exchanged encrypted under the key. Since communicating over a secure channel is the ultimate goal of any key exchange protocol, this is certainly not a flaw in the protocol. Similar to the lemmas proving the objectives K3 and A1, the initiator authentication lemma and therefore authentication objective A2 can be proved in F* without further proof work, because the event relations are ensured by the signature usage predicate in combination with the event predicate.

4.5.6 Discussion

In this thesis, we gave a detailed analysis of the Denning-Sacco protocol with public keys, resulting in the first symbolic security proof of the protocol in DY*. The proof bases on an extension of the DY* framework for modeling time-based properties of cryptographic protocols. Using this extension, we have developed a coherent model of the Denning-Sacco protocol in DY*, and formulated and proved security properties of the protocol derived from its description and from its properties observed during the analysis. To establish the soundness of our security proof, we have verified the properties defined in the model with F*.

In Section 3.3.3, an attack on the protocol documented by Abadi and Needham was presented. We have already explained that this attack bases on a different assumed message structure than the structure represented in our DY* model. Abadi and Needham assume that the initiator obtains the timestamp it includes in the communication key certificate from its own local clock, implying that it is different from the timestamp in the public key certificates, and that each timestamp is validated separately by comparison with the local clock of the respective receiver. Because the communication key certificate received from the initiator does not contain the identity of the intended key peer, the responder could then simply replay the certificate – and thus the key – while the timestamp in the certificate is valid. This attack would impose a serious flaw in the protocol, since it enables a malicious responder to impersonate the initiator to any other party it wishes. Our implementation of the Denning-Sacco protocol in DY* is based on the original message structure proposed by Denning and Sacco, and thereby prevents this attack because the initiator reuses the timestamp contained in the public key certificates in the communication key certificate, and the responder only accepts a key if the timestamps in all certificates match. We argued that it is valid to assume that timestamps from different certificates are compared if we presuppose that there is only a single timestamp used in all messages, obtained by the server and later reused by the initiator. Nonetheless, we highly recommend the inclusion of the identity of the communication key peer in the corresponding certificate – as demanded by the robustness principle of Boyd and Mao [7] – such that any possible receiver of the third message can verify this identity, and abort if the key appears to be intended for someone else. In addition, principals should still adhere to the timestamp validation procedure used in our model, which requires only a single comparison of the timestamp to the current local clock value to validate the timeliness of all certificates in a message at once. Since the attack assumes a different message structure, it is clearly not in contradiction to the security proof we provided in the previous chapter. Instead, the attack adds to our proof by highlighting a subtle property of the last message that would allow the communication key to be replayed if the certificates are not sufficiently checked by the recipient, and further that this is a crucial assumption we make in our proof.

Denning and Sacco's research was primarily concerned with the properties and benefits of timestamps in key exchange protocols, so we had to define other security objectives of the protocol, e.g. regarding secrecy of the communication key or authentication of the users, based on the properties and goals we observed while examining the protocol in detail in the course of our analysis. Also, with timestamps being used instead of nonces in the protocol, we had to come up with novel formulations for certain security properties in the Denning-Sacco model, e.g., for the authentication or the replay mitigation properties. We tried to emphasize the accuracy of our security proof including time-based properties by giving a detailed explanation of why we believe that our approach for modeling such properties in DY* is sound, and by demonstrating our implementation in a passive adversary setting, as well as in a setting with multiple malicious parties. The general applicability of our approach could be further illustrated by conducting DY* analyses of other protocols, which depend on timestamps, such as the Wide Mouthed Frog protocol [10].

5 Conclusion and Outlook

In this thesis, we analysed three authentication and key exchange protocols using the DY* symbolic prover: the Otway-Rees protocol, the Yahalom protocol and the Denning-Sacco protocol with public keys. In Section 3.3, we described the three protocols and summarized related research containing previous security analyses of these protocols and illustrating found attacks. The protocols were then analysed in detail in Chapter 4. For each of the protocols, we developed an accurate model in DY* based on the provided description, and expressed security properties for the model, also based on the protocol description and complemented with properties observed during the analysis. We tested the functional correctness of our models by executing them and inspecting the resulting trace, and further established their soundness and proved the security properties by verifying both together in F*. For security goals of protocols for which we could not prove respective security properties in the DY* model, we explored resulting attack vectors, extended the DY* model with possible attacks, and finally proposed an improved version of the respective protocol, for which we proved that remaining security goals are achieved by adapting the model accordingly and again verifying its soundness in F*. We then discussed the results of our analysis of each of the protocols in the context of the results of other analyses presented, on the one hand, to confirm our own results, and on the other hand, to explain possible differences in the outcomes with different methods used and different assumptions made for the analyses.

The analysis of the Otway-Rees protocol revealed what was already shown by multiple analyses before, that the protocol – if implemented strictly according to the protocol description – is at least vulnerable to an attack in which a malicious party can impersonate the intended target of the key exchange to the initiator of the protocol. Further, we also identified the well-known type-flaws in the protocol that possibly allow an attacker to replay parts of messages to the initiator as well as the responder, and thus substitute the key generated by the server with some publicly known term. We modeled three attacks in total based on the flaws of the protocol and proposed improvements that manage to prevent the attacks and make the protocol achieve the intended security goals, such as secrecy of the session key and timely mutual authentication, assuming that the parties completing the protocol together are all honest. Finally, we discussed improved versions of the protocol proposed by Boyd and Mao [7], which we showed to not actually prevent the impersonation attack, and by Chen [12], who suggested a strongly modified version achieving even stronger security goals than those intended by Otway and Rees. In contrast, our improved protocol uses minimal changes to achieve the goals that were not fulfilled with the original version.

Our analysis of the Yahalom protocol argues against research like [13] and [14] that labels the protocol as flawed, and shows that it achieves the security goals described by the BAN authors, who were the first to specify the protocol in a formal setting. The security properties of the Yahalom protocol include key secrecy and mutual authentication with additional timeliness guarantees, similar as in the (improved) Otway-Rees protocol, and again under the assumption that the involved principals are honest. Our security proof also accounts for the fact that the responder gets even stronger guarantees insofar that the initiator confirms with the last message that it knows and trusts the exchanged session key coming from the server. This guarantee is eventually returned to the initiator, when the responder sends the first message encrypted under the shared key. We claim that any attack breaking the security of the Yahalom protocol involves the compromise of at least one of the three principal roles involved in the protocol.

For the analysis of the Denning-Sacco protocol, we have developed an extension of the DY* framework that lets us model protocols with timestamps, and typical properties of such protocols. We have demonstrated our implementation on a DY* model of the Denning-Sacco protocol and shown that time-based properties can be captured while not restricting the DY* adversary in its ability to launch attacks that would not violate the time-based properties of the protocol in reality. Our work shows that expressing properties involving timestamps is not an easy task in DY*, because while the timeliness guarantees of timestamps in combination with real clocks are rather lax and can possibly even differ in multiple executions of the protocol, symbolic provers like DY* are exactly built to express and prove strict security properties using a symbolic measure of time that results in the same outcome regarding the protocol's security in all possible protocol runs. Therefore, we expressed the real-world properties of the protocol as idealized properties in the DY* model. For the Denning-Sacco model, we were able to prove that the exchanged key is secret and shared between the initiator and responder, unless one of them is compromised, and further that keys exchanged between honest parties cannot be replayed once compromised. Regarding authentication, we could show that if all parties remain honest, the exchanged key indeed provides a means for authenticated and confidential communication between initiator and responder, meaning that both users have a reliable view on the origin of messages encrypted under this very key. Similarly as in the Yahalom protocol, the responder gets the added guarantee at the end of the protocol that the initiator possesses the same communication key as itself and that the initiator indeed intended to share this key with the responder. Notably, this ultimately strong unidirectional authentication is achieved without the need for the responder to be involved in requesting the public key certificates and thus without a handshake between the users. The results of our analysis stand against research like [1] and [2], describing an attack on the protocol based on a different structure of the last message assumed by the authors, where the timestamp in the communication key certificate created by the initiator is independent of the timestamp in the public key certificates. In Section 3.3.3, we have clarified what the actual structure of the last message looks like. Precisely, one of the authors of the protocol (Giovanni Maria Sacco) confirmed that the initiator reuses the timestamp from the other certificates provided by the server. Our analysis bases on the originally intended structure of the last message, and that the timestamps are validated in the most efficient and secure way, resulting in the first symbolic security proof of the Denning-Sacco protocol with public keys.

Outlook

While developing the DY* models for the Otway-Rees and Yahalom protocols, we had to write our own API to install the symmetric long-term keys that are pre-shared between initiator and authentication server, and responder and server in both protocols. Also, the keys were stored within the protocol specific application state defined in modules `OYRS.Sessions` for the Otway-Rees protocol and `YLM.Sessions` for the Yahalom protocol, even though the long-term keys could potentially be used in any authentication protocol based on symmetric long-term keys. Storing keys in the protocol state also led us to first model the Otway-Rees protocol so that each principal could only participate in a single protocol run, which is a restriction we usually do not want to make when modeling a protocol. Hence, we think it is important that DY* offers a generic API for generating and installing symmetric long-term keys, in addition to the API for public keys, as part of the `LabeledPKI` module in the labeled layer, so that this is no longer a problem faced by developers of cryptographic protocol code written in DY*.

Besides this, there are also a couple of lemmas we defined in helper modules on the level of our protocol code that are generally useful when modeling protocols involving an authentication server or any other kind of third party. To give just one example, we had to define lemmas similar to `readers_is_injective`, which shows the injectivity between a label `readers [P p]` and the principal `p` in the state session identifier `P p` that can read the label, for `readers` labels with two or even three state session identifiers, since we used symmetric keys shared between two or even three parties to encrypt messages in the Otway-Rees and Yahalom protocols. We recommend extending DY* with lemmas that may be useful in a wider range of protocols, thus avoiding the hassle of having to define them anew for each protocol model.

As mentioned, we have further presented an extension of the DY* framework in this thesis that allowed us to model the Denning-Sacco protocol and capture its properties depending on timestamps. To verify the general applicability of our proposed approach, we encourage other protocols based on timestamps to be modeled with our DY* extension, for example the Wide Mouthed Frog protocol [10], which uses symmetric long-term keys like Otway-Rees and Yahalom, but requires only two messages to exchange a key, and see if we can adequately capture the security properties of these protocols. Moreover, it makes sense to integrate the extension (currently consisting of the two modules `DS.Clock` dedicated with timestamp validation, and `DS.SendRecv`, which contains wrappers for the send and receive functions that increase the value of a clock) directly into the DY* framework, such that they are no longer part of the protocol model itself.

We also hope that with the first elaborate symbolic security proof of the Denning-Sacco protocol with public keys using DY* in this thesis, we can guide future research more towards investigating protocols with time-based properties. As we find and as our analysis shows, such protocols have very interesting properties: they typically require very few messages to perform a secure key exchange, and they protect very effectively against the replay of compromised keys if none of the parties sharing the key acts with malicious intent at the time of the key exchange.

Bibliography

[1] M. Abadi, R. Needham. "Prudent engineering practice for cryptographic protocols". In: *IEEE transactions on Software Engineering* 22.1 (1996), pp. 6–15 (cit. on pp. 40, 41, 130).

[2] R. Anderson, R. Needham. "Programming Satan's computer". In: *Computer Science Today: Recent Trends and Developments* (2005), pp. 426–440 (cit. on pp. 40, 41, 130).

[3] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Vigano, L. Vigneron. "The AVISS security protocol analysis tool". In: *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*. Springer. 2002, pp. 349–354 (cit. on pp. 32, 42).

[4] M. Backes. "A cryptographically sound Dolev-Yao style security proof of the Otway-Rees protocol". In: *Computer Security–ESORICS 2004: 9th European Symposium on Research in Computer Security, Sophia Antipolis, France, September 13-15, 2004. Proceedings 9*. Springer. 2004, pp. 89–108 (cit. on pp. 18, 33, 36, 80).

[5] M. Backes, B. Pfitzmann, M. Waidner. "A universally composable cryptographic library". In: *Cryptology ePrint Archive* (2003) (cit. on pp. 33, 36).

[6] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, T. Würtele. "DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021, pp. 523–542 (cit. on pp. 18, 19, 21, 31).

[7] C. Boyd, W. Mao. "Limitations of logical analysis of cryptographic protocols". In: Citeseer. 1993 (cit. on pp. 19, 31, 34, 35, 41, 63, 69, 78, 127, 129).

[8] C. Boyd, A. Mathuria, D. Stebila. *Protocols for authentication and key establishment*. Vol. 1. Springer, 2003 (cit. on pp. 30, 31, 36, 40, 41).

[9] S. H. Brackin. "Evaluating and improving protocol analysis by automatic proof". In: *Proceedings. 11th IEEE Computer Security Foundations Workshop (Cat. No. 98TB100238)*. IEEE. 1998, pp. 138–152 (cit. on p. 32).

[10] M. Burrows. "Wide mouthed frog". In: *Security Protocols Open Repository, http://www. lsv. ens-cachan. fr/Software/spore/wideMouthedFrog. html* (1989) (cit. on pp. 128, 131).

[11] M. Burrows, M. Abadi, R. Needham. *A Logic of Authentication*. Digital Equipment Corporation, 1989 (cit. on pp. 17, 19, 30, 31, 34–39, 43, 78, 81, 96).

[12] L. Chen. "Improved Otway Rees protocol and its formal verification". In: *2008 International Conference on Communications, Circuits and Systems*. IEEE. 2008, pp. 498–501 (cit. on pp. 32, 35, 38, 79, 129).

[13] L. Chen, M. Shi. "Security analysis and improvement of Yahalom protocol". In: *2008 3rd IEEE Conference on Industrial Electronics and Applications*. IEEE. 2008, pp. 1137–1140 (cit. on pp. 32, 37, 39, 97, 129).

[14] J. A. Clark, J. L. Jacob. “A survey of authentication protocol literature: Version 1.0”. In: (1997) (cit. on pp. 31, 36, 39, 97, 129).

[15] V. Cortier, S. Kremer, B. Warinschi. “A survey of symbolic methods in computational analysis of cryptographic systems”. In: *Journal of Automated Reasoning* 46 (2011), pp. 225–259 (cit. on p. 33).

[16] D. E. Denning, G. M. Sacco. “Timestamps in key distribution protocols”. In: *Communications of the ACM* 24.8 (1981), pp. 533–536 (cit. on pp. 19, 30, 39, 41, 43, 99, 125).

[17] D. Dolev, A. Yao. “On the security of public key protocols”. In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208 (cit. on pp. 17, 25, 31).

[18] *F*: A Proof-Oriented Programming Language*. Accessed: 2023-03-23. URL: https://www.fstar-lang.org/#introduction (cit. on p. 21).

[19] L. Gong, R. M. Needham, R. Yahalom. “Reasoning about Belief in Cryptographic Protocols.” In: *IEEE Symposium on Security and Privacy*. Vol. 1990. Citeseer. 1990, pp. 234–248 (cit. on p. 32).

[20] J. D. Guttman, F. J. Thayer. “Authentication tests”. In: *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*. IEEE. 2000, pp. 96–109 (cit. on p. 35).

[21] S. Holderbach. $MAThesis_{D}YStar_{P}rotocols$. Accessed: 2023-06-21. 2023. URL: https://github.com/elsamuray7/MAThesis_DYStar_Protocols (cit. on p. 44).

[22] X. Li, Q. Wang. “An improvement of authentication test for security protocol analysis”. In: *2007 International Conference on Computational Intelligence and Security Workshops (CISW 2007)*. IEEE. 2007, pp. 745–748 (cit. on p. 40).

[23] G. Lowe. “Breaking and fixing the Needham-Schroeder public-key protocol using FDR”. In: *Tools and Algorithms for the Construction and Analysis of Systems: Second International Workshop, TACAS’96 Passau, Germany, March 27–29, 1996 Proceedings 2*. Springer. 1996, pp. 147–166 (cit. on p. 18).

[24] D. McGrew. *An interface and algorithms for authenticated encryption*. Tech. rep. 2008 (cit. on p. 25).

[25] S. Meier, B. Schmidt, C. Cremers, D. Basin. “The TAMARIN prover for the symbolic analysis of security protocols”. In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer. 2013, pp. 696–701 (cit. on pp. 18, 32).

[26] R. M. Needham, M. D. Schroeder. “Using encryption for authentication in large networks of computers”. In: *Communications of the ACM* 21.12 (1978), pp. 993–999 (cit. on pp. 17, 18).

[27] D. Otway, O. Rees. “Efficient and timely mutual authentication”. In: *ACM SIGOPS Operating Systems Review* 21.1 (1987), pp. 8–10 (cit. on pp. 18, 19, 30, 33, 35, 43, 45, 59).

[28] L. C. Paulson. “Relations between secrets: Two formal analyses of the Yahalom protocol”. In: *Journal of computer security* 9.3 (2001), pp. 197–216 (cit. on pp. 38, 39, 98).

[29] T. Perrin, M. Marlinspike. “The double ratchet algorithm”. In: *GitHub wiki* (2016), p. 10 (cit. on p. 18).

[30] E. Rescorla. *The transport layer security (TLS) protocol version 1.3*. Tech. rep. 2018 (cit. on p. 18).

[31] P. Ryan, S. A. Schneider, M. Goldsmith, G. Lowe, B. Roscoe. *The modelling and analysis of security protocols: the CSP approach*. Addison-Wesley Professional, 2001 (cit. on pp. 17, 32, 38, 39, 98).

[32] E. Snekkenes. “Exploring the BAN approach to protocol analysis”. In: *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society. 1991, pp. 171–171 (cit. on p. 32).

[33] P. Syverson. “A taxonomy of replay attacks [cryptographic protocols]”. In: *Proceedings The Computer Security Foundations Workshop VII*. IEEE. 1994, pp. 187–191 (cit. on pp. 31, 37).

[34] P. F. Syverson, P. C. Van Oorschot. “On unifying some cryptographic protocol logics”. In: *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE. 1994, pp. 14–28 (cit. on pp. 32, 35, 37).

[35] L. Vigano. “Automated security protocol analysis with the AVISPA tool”. In: *Electronic Notes in Theoretical Computer Science* 155 (2006), pp. 61–86 (cit. on pp. 18, 32).

[36] M. Yang, J. Luo. “Analysis of security protocols based on authentication test”. In: *Journal of Software* 17.1 (2006), pp. 148–156 (cit. on p. 35).

[37] L. Yu, Y.-Y. Guo, Z.-P. Zhuo, S.-M. Wei. “Analysis and Improvement of Otway-Rees Based on Enhanced Authentication Tests”. In: *International Journal of Network Security* 23.3 (2021), pp. 426–435 (cit. on pp. 35, 79).

All links were last followed on June 15, 2023.

A Output of the Otway-Rees Model

A.1 Benign Attacker Trace

```
======================
Otway-Rees
======================
Starting Benign Attacker:
start
 0. Generated AE sk_i_srv(0)
    Label: [initiator;server]
    Usage: AE sk_i_srv
 1. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
 2. Generated AE sk_r_srv(2)
    Label: [responder;server]
    Usage: AE sk_r_srv
 3. SetState responder:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
 4. SetState server:
    Session 0(v0): ((APP | (srv_sess | (initiator | (AE sk_i_srv(0) | sk_i_srv)))))
 5. SetState server:
    Session 0(v0): ((APP | (srv_sess | (initiator | (AE sk_i_srv(0) | sk_i_srv)))))
    Session 1(v0): ((APP | (srv_sess | (responder | (AE sk_r_srv(2) | sk_r_srv)))))
 6. Generated NONCE conv_id(6)
    Label: Public
    Usage: NONCE conv_id
 7. Generated NONCE nonce_i(7)
    Label: [initiator;server]
    Usage: NONCE nonce_i
 8. Event initiator: initiate(NONCE conv_id(6),initiator,responder,NONCE nonce_i(7))
 9. Message initiator->responder: (msg1 | (NONCE conv_id(6) | (initiator | (responder |
aead_enc(AE sk_i_srv(0),iv,(NONCE nonce_i(7) | (NONCE conv_id(6) | (initiator | responder))),
ev_i)))))
10. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
    Session 1(v0): ((APP | (i_sent_m1 | (server | (AE sk_i_srv(0) | (responder | (NONCE
conv_id(6) | NONCE nonce_i(7))))))))
11. Generated NONCE nonce_r(11)
    Label: [responder;server]
    Usage: NONCE nonce_r
12. Event responder: req_key(NONCE conv_id(6),initiator,responder,NONCE nonce_r(11))
```

```
13. Message responder->server: (msg2 | (NONCE conv_id(6) | (initiator | (responder | (aead_enc
(AE sk_i_srv(0),iv,(NONCE nonce_i(7) | (NONCE conv_id(6) | (initiator | responder))),ev_i) |
aead_enc(AE sk_r_srv(2),iv,(NONCE nonce_r(11) | (NONCE conv_id(6) | (initiator | responder))),
ev_r))))))
14. SetState responder:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
    Session 1(v0): ((APP | (r_sent_m2 | (server | (AE sk_r_srv(2) | (initiator | (NONCE
conv_id(6) | NONCE nonce_r(11))))))))
15. Generated AE sk_i_r(15)
    Label: [server;initiator;responder]
    Usage: AE sk_i_r
16. Event server: send_key(NONCE conv_id(6),initiator,responder,NONCE nonce_i(7),NONCE nonce_r
(11),AE sk_i_r(15))
17. Message server->responder: (msg3 | (NONCE conv_id(6) | (aead_enc(AE sk_i_srv(0),iv,(NONCE
nonce_i(7) | AE sk_i_r(15)),ev_i) | aead_enc(AE sk_r_srv(2),iv,(NONCE nonce_r(11) | AE sk_i_r
(15)),ev_r))))
18. SetState server:
    Session 0(v0): ((APP | (srv_sess | (initiator | (AE sk_i_srv(0) | sk_i_srv)))))
    Session 1(v0): ((APP | (srv_sess | (responder | (AE sk_r_srv(2) | sk_r_srv)))))
    Session 2(v0): ((APP | (srv_sent_m3 | (initiator | (responder | (NONCE conv_id(6) | (NONCE
 nonce_i(7) | (NONCE nonce_r(11) | AE sk_i_r(15)))))))))
19. Event responder: fwd_key(NONCE conv_id(6),initiator,responder,AE sk_i_r(15))
20. Message responder->initiator: (msg4 | (NONCE conv_id(6) | aead_enc(AE sk_i_srv(0),iv,(
NONCE nonce_i(7) | AE sk_i_r(15)),ev_i)))
21. SetState responder:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
    Session 1(v0): ((APP | (r_sent_m4 | (server | (initiator | AE sk_i_r(15))))))
22. Event initiator: recv_key(NONCE conv_id(6),initiator,responder,AE sk_i_r(15))
23. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
    Session 1(v0): ((APP | (i_rcvd_m4 | (server | (responder | AE sk_i_r(15))))))
PROTOCOL RUN: Successful execution of Otway-Rees protocol.
Finished Benign Attacker:
```

In the trace entries 0 to 5, the principal's states are set up according to their initial knowledge, including generation and installation of long-term keys shared between the initiator and server, and responder and server, respectively. The event `initiate(NONCE conv_id(6),initiator,responder ,NONCE nonce_i(7))` followed by the first message from the initiator to the responder indicates the start of the protocol. The responder proceeds by augmenting the first message and requesting a session key from the server. In turn, the key `AE sk_i_r(15)` is generated and distributed by the server in two certificates – one for each user. The certificates are first received by the responder, who then forwards the initiator's certificate. Both store the key in their respective final protocol state sessions.

A.2 Impersonation Attacker Trace

```
Starting Impersonate Responder to Initiator Attacker:
start
 0. Generated AE sk_i_srv(0)
    Label: [alice;server]
    Usage: AE sk_i_srv
 1. SetState alice:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | bob)))))
 2. Generated AE sk_r_srv(2)
    Label: [bob;server]
    Usage: AE sk_r_srv
 3. SetState bob:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
 4. Generated AE sk_r_srv(4)
    Label: [eve;server]
    Usage: AE sk_r_srv
 5. SetState eve:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(4)))))
 6. SetState server:
    Session 0(v0): ((APP | (srv_sess | (alice | (AE sk_i_srv(0) | sk_i_srv)))))
 7. SetState server:
    Session 0(v0): ((APP | (srv_sess | (alice | (AE sk_i_srv(0) | sk_i_srv)))))
    Session 1(v0): ((APP | (srv_sess | (bob | (AE sk_r_srv(2) | sk_r_srv)))))
 8. SetState server:
    Session 0(v0): ((APP | (srv_sess | (alice | (AE sk_i_srv(0) | sk_i_srv)))))
    Session 1(v0): ((APP | (srv_sess | (bob | (AE sk_r_srv(2) | sk_r_srv)))))
    Session 2(v0): ((APP | (srv_sess | (eve | (AE sk_r_srv(4) | sk_r_srv)))))
 9. Compromised eve(0)(0)
10. Generated NONCE conv_id(10)
    Label: Public
    Usage: NONCE conv_id
11. Generated NONCE nonce_i(11)
    Label: [alice;server]
    Usage: NONCE nonce_i
12. Event alice: initiate(NONCE conv_id(10),alice,bob,NONCE nonce_i(11))
13. Message alice->bob: (msg1 | (NONCE conv_id(10) | (alice | (bob | aead_enc(AE sk_i_srv(0),
iv,(NONCE nonce_i(11) | (NONCE conv_id(10) | (alice | bob))),ev_i)))))
14. SetState alice:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | bob)))))
    Session 1(v0): ((APP | (i_sent_m1 | (server | (AE sk_i_srv(0) | (bob | (NONCE conv_id(10)
| NONCE nonce_i(11))))))))
15. Generated NONCE nonce_attacker(15)
    Label: Public
    Usage: NONCE nonce_attacker
16. Message *->*: NONCE nonce_attacker(15)
17. Message eve->server: (msg2 | (NONCE conv_id(10) | (alice | (eve | (aead_enc(AE sk_i_srv(0)
,iv,(NONCE nonce_i(11) | (NONCE conv_id(10) | (alice | bob))),ev_i) | aead_enc(AE sk_r_srv(4),
iv,(NONCE nonce_attacker(15) | (NONCE conv_id(10) | (alice | bob))),ev_r))))))
18. Generated AE sk_i_r(18)
    Label: [server;alice;eve]
    Usage: AE sk_i_r
19. Event server: send_key(NONCE conv_id(10),alice,eve,NONCE nonce_i(11),NONCE nonce_attacker
(15),AE sk_i_r(18))
20. Message server->eve: (msg3 | (NONCE conv_id(10) | (aead_enc(AE sk_i_srv(0),iv,(NONCE
nonce_i(11) | AE sk_i_r(18)),ev_i) | aead_enc(AE sk_r_srv(4),iv,(NONCE nonce_attacker(15) | AE
 sk_i_r(18)),ev_r))))
21. SetState server:
    Session 0(v0): ((APP | (srv_sess | (alice | (AE sk_i_srv(0) | sk_i_srv)))))
    Session 1(v0): ((APP | (srv_sess | (bob | (AE sk_r_srv(2) | sk_r_srv)))))
    Session 2(v0): ((APP | (srv_sess | (eve | (AE sk_r_srv(4) | sk_r_srv)))))
    Session 3(v0): ((APP | (srv_sent_m3 | (alice | (eve | (NONCE conv_id(10) | (NONCE nonce_i
(11) | (NONCE nonce_attacker(15) | AE sk_i_r(18)))))))))
22. Message bob->alice: (msg4 | (NONCE conv_id(10) | aead_enc(AE sk_i_srv(0),iv,(NONCE nonce_i
(11) | AE sk_i_r(18)),ev_i)))
23. Event alice: recv_key(NONCE conv_id(10),alice,bob,AE sk_i_r(18))
24. SetState alice:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | bob)))))
    Session 1(v0): ((APP | (i_rcvd_m4 | (server | (bob | AE sk_i_r(18))))))
PROTOCOL RUN: Successful execution of Otway-Rees protocol.
Finished Impersonate Responder to Initiator Attacker:
```

We set up one additional responder `eve` in this attack, so that the first 8 trace entries correspond to the setup phase. `eve` represents the malicious principal controlled by the attacker, who thus compromises the long-term key of `eve` and the server. The initiator `alice` initiates a protocol run with the intention of establishing a shared key with `bob`, as indicated by the event `initiate(NONCE conv_id (10),alice,bob,NONCE nonce_i(11))` and by the first message. The key of the attack is to be found in the trace entries 15 to 17, where the attacker, on behalf of `eve`, generates a nonce `nonce_attacker(15)`, and emits a message that makes it look to the server like `eve` was the responder to which `alice` wanted to talk to by replacing `bob`'s name with `eve`'s in the plaintext. The server accepts the message and distributes a key destined for `alice` and `eve`. However, the final protocol state session of `alice` shows that she still believes to share this key with `bob`.

A.3 Traces of Intercept and Replay Attacks

```
Starting Intercept Msg1 Attacker:
start
 0. Generated AE sk_i_srv(0)
    Label: [initiator;server]
    Usage: AE sk_i_srv
 1. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
 2. Generated AE sk_r_srv(2)
    Label: [responder;server]
    Usage: AE sk_r_srv
 3. SetState responder:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
 4. SetState server:
    Session 0(v0): ((APP | (srv_sess | (initiator | (AE sk_i_srv(0) | sk_i_srv)))))
 5. SetState server:
    Session 0(v0): ((APP | (srv_sess | (initiator | (AE sk_i_srv(0) | sk_i_srv)))))
    Session 1(v0): ((APP | (srv_sess | (responder | (AE sk_r_srv(2) | sk_r_srv)))))
 6. Generated NONCE conv_id(6)
    Label: Public
    Usage: NONCE conv_id
 7. Generated NONCE nonce_i(7)
    Label: [initiator;server]
    Usage: NONCE nonce_i
 8. Event initiator: initiate(NONCE conv_id(6),initiator,responder,NONCE nonce_i(7))
 9. Message initiator->responder: (msg1 | (NONCE conv_id(6) | (initiator | (responder |
aead_enc(AE sk_i_srv(0),iv,(NONCE nonce_i(7) | (NONCE conv_id(6) | (initiator | responder))),
ev_i)))))
10. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
    Session 1(v0): ((APP | (i_sent_m1 | (server | (AE sk_i_srv(0) | (responder | (NONCE
conv_id(6) | NONCE nonce_i(7))))))))
11. Message responder->initiator: (msg4 | (NONCE conv_id(6) | aead_enc(AE sk_i_srv(0),iv,(
NONCE nonce_i(7) | (NONCE conv_id(6) | (initiator | responder))),ev_i)))
12. Event initiator: recv_key(NONCE conv_id(6),initiator,responder,(NONCE conv_id(6) | (
initiator | responder)))
13. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
    Session 1(v0): ((APP | (i_rcvd_m4 | (server | (responder | (NONCE conv_id(6) | (initiator
| responder)))))))
PROTOCOL RUN: Successful execution of Otway-Rees protocol.
Finished Intercept Msg1 Attacker:
```

This attack runs the same as the benign scenario to the point where the responder would complete the session key request to the server. The adversary intercepts the first message from the initiator, and replays its part of the request as session key certificate to the initiator, who accepts `(NONCE conv_id(6) | (initiator | responder))` consisting only of terms derivable by the attacker as supposed session key for secret communication with the responder.

```
Starting Intercept Msg2 Attacker:
start
 0. Generated AE sk_i_srv(0)
    Label: [initiator;server]
    Usage: AE sk_i_srv
 1. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
 2. Generated AE sk_r_srv(2)
    Label: [responder;server]
    Usage: AE sk_r_srv
 3. SetState responder:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
 4. SetState server:
    Session 0(v0): ((APP | (srv_sess | (initiator | (AE sk_i_srv(0) | sk_i_srv)))))
 5. SetState server:
    Session 0(v0): ((APP | (srv_sess | (initiator | (AE sk_i_srv(0) | sk_i_srv)))))
    Session 1(v0): ((APP | (srv_sess | (responder | (AE sk_r_srv(2) | sk_r_srv)))))
 6. Generated NONCE conv_id(6)
    Label: Public
    Usage: NONCE conv_id
 7. Generated NONCE nonce_i(7)
    Label: [initiator;server]
    Usage: NONCE nonce_i
 8. Event initiator: initiate(NONCE conv_id(6),initiator,responder,NONCE nonce_i(7))
 9. Message initiator->responder: (msg1 | (NONCE conv_id(6) | (initiator | (responder |
aead_enc(AE sk_i_srv(0),iv,(NONCE nonce_i(7) | (NONCE conv_id(6) | (initiator | responder))),
ev_i)))))
10. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
    Session 1(v0): ((APP | (i_sent_m1 | (server | (AE sk_i_srv(0) | (responder | (NONCE
conv_id(6) | NONCE nonce_i(7))))))))
11. Generated NONCE nonce_r(11)
    Label: [responder;server]
    Usage: NONCE nonce_r
12. Event responder: req_key(NONCE conv_id(6),initiator,responder,NONCE nonce_r(11))
13. Message responder->server: (msg2 | (NONCE conv_id(6) | (initiator | (responder | (aead_enc
(AE sk_i_srv(0),iv,(NONCE nonce_i(7) | (NONCE conv_id(6) | (initiator | responder))),ev_i) |
aead_enc(AE sk_r_srv(2),iv,(NONCE nonce_r(11) | (NONCE conv_id(6) | (initiator | responder))),
ev_r))))))
14. SetState responder:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
    Session 1(v0): ((APP | (r_sent_m2 | (server | (AE sk_r_srv(2) | (initiator | (NONCE
conv_id(6) | NONCE nonce_r(11))))))))
15. Message server->responder: (msg3 | (NONCE conv_id(6) | (aead_enc(AE sk_i_srv(0),iv,(NONCE
nonce_i(7) | (NONCE conv_id(6) | (initiator | responder))),ev_i) | aead_enc(AE sk_r_srv(2),iv
,(NONCE nonce_r(11) | (NONCE conv_id(6) | (initiator | responder))),ev_r))))
16. Event responder: fwd_key(NONCE conv_id(6),initiator,responder,(NONCE conv_id(6) | (
initiator | responder)))
17. Message responder->initiator: (msg4 | (NONCE conv_id(6) | aead_enc(AE sk_i_srv(0),iv,(
NONCE nonce_i(7) | (NONCE conv_id(6) | (initiator | responder))),ev_i)))
18. SetState responder:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
    Session 1(v0): ((APP | (r_sent_m4 | (server | (initiator | (NONCE conv_id(6) | (initiator
| responder)))))))
19. Event initiator: recv_key(NONCE conv_id(6),initiator,responder,(NONCE conv_id(6) | (
initiator | responder)))
20. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
    Session 1(v0): ((APP | (i_rcvd_m4 | (server | (responder | (NONCE conv_id(6) | (initiator
| responder)))))))
PROTOCOL RUN: Successful execution of Otway-Rees protocol.
Finished Intercept Msg2 Attacker:
```

In this attack, the adversary waits even longer than in the previous attack and lets the responder complete the session key request. The adversary intercepts the request to the server and replays both parts of the request, respectively, to the iniator and responder. As a result, they both accept `(NONCE conv_id(6) | (initiator | responder))` as session key, allowing the attacker to impersonate either user to the other.

A.4 Output and Trace of the Improved Model

```
======================
Otway-Rees
======================
Starting Benign Attacker:
start
 0. Generated AE sk_i_srv(0)
    Label: [initiator;server]
    Usage: AE sk_i_srv
 1. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
 2. Generated AE sk_r_srv(2)
    Label: [responder;server]
    Usage: AE sk_r_srv
 3. SetState responder:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
 4. SetState server:
    Session 0(v0): ((APP | (srv_sess | (initiator | (AE sk_i_srv(0) | sk_i_srv)))))
 5. SetState server:
    Session 0(v0): ((APP | (srv_sess | (initiator | (AE sk_i_srv(0) | sk_i_srv)))))
    Session 1(v0): ((APP | (srv_sess | (responder | (AE sk_r_srv(2) | sk_r_srv)))))
 6. Generated NONCE conv_id(6)
    Label: Public
    Usage: NONCE conv_id
 7. Generated NONCE nonce_i(7)
    Label: [initiator;server]
    Usage: NONCE nonce_i
 8. Event initiator: initiate(NONCE conv_id(6),initiator,responder,server,NONCE nonce_i(7))
 9. Message initiator->responder: (msg1 | (NONCE conv_id(6) | (initiator | (responder |
aead_enc(AE sk_i_srv(0),iv,(NONCE conv_id(6) | (initiator | (responder | NONCE nonce_i(7)))),
ev1)))))
10. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
    Session 1(v0): ((APP | (i_sent_m1 | (server | (AE sk_i_srv(0) | (responder | (NONCE
conv_id(6) | NONCE nonce_i(7))))))))
11. Generated NONCE nonce_r(11)
    Label: [responder;server]
    Usage: NONCE nonce_r
12. Event responder: req_key(NONCE conv_id(6),initiator,responder,server,NONCE nonce_r(11))
13. Message responder->server: (msg2 | (NONCE conv_id(6) | (initiator | (responder | (aead_enc
(AE sk_i_srv(0),iv,(NONCE conv_id(6) | (initiator | (responder | NONCE nonce_i(7)))),ev1) |
aead_enc(AE sk_r_srv(2),iv,(NONCE conv_id(6) | (initiator | (responder | NONCE nonce_r(11)))),
ev2))))))
14. SetState responder:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
    Session 1(v0): ((APP | (r_sent_m2 | (server | (AE sk_r_srv(2) | (initiator | (NONCE
conv_id(6) | NONCE nonce_r(11))))))))
15. Generated AE sk_i_r(15)
    Label: [server;initiator;responder]
    Usage: AE sk_i_r
16. Event server: send_key(NONCE conv_id(6),initiator,responder,server,NONCE nonce_i(7),NONCE
nonce_r(11),AE sk_i_r(15))
17. Message server->responder: (msg3 | (NONCE conv_id(6) | (aead_enc(AE sk_i_srv(0),iv,(NONCE
nonce_i(7) | (responder | AE sk_i_r(15))),ev3_i) | aead_enc(AE sk_r_srv(2),iv,(NONCE nonce_r
(11) | (initiator | AE sk_i_r(15))),ev3_r))))
18. SetState server:
    Session 0(v0): ((APP | (srv_sess | (initiator | (AE sk_i_srv(0) | sk_i_srv)))))
    Session 1(v0): ((APP | (srv_sess | (responder | (AE sk_r_srv(2) | sk_r_srv)))))
    Session 2(v0): ((APP | (srv_sent_m3 | (initiator | (responder | (NONCE conv_id(6) | (NONCE
 nonce_i(7) | (NONCE nonce_r(11) | AE sk_i_r(15)))))))))
19. Event responder: fwd_key(NONCE conv_id(6),initiator,responder,server,NONCE nonce_r(11),AE
sk_i_r(15))
20. Message responder->initiator: (msg4 | (NONCE conv_id(6) | aead_enc(AE sk_i_srv(0),iv,(
NONCE nonce_i(7) | (responder | AE sk_i_r(15))),ev3_i)))
```

```
21. SetState responder:
    Session 0(v0): ((APP | (r_init | (server | AE sk_r_srv(2)))))
    Session 1(v0): ((APP | (r_sent_m4 | (server | (initiator | AE sk_i_r(15))))))
22. Event initiator: recv_key(NONCE conv_id(6),initiator,responder,server,NONCE nonce_i(7),AE
sk_i_r(15))
23. SetState initiator:
    Session 0(v0): ((APP | (i_init | (server | (AE sk_i_srv(0) | responder)))))
    Session 1(v0): ((APP | (i_rcvd_m4 | (server | (responder | AE sk_i_r(15))))))
PROTOCOL RUN: Successful execution of Otway-Rees protocol.
Finished Benign Attacker:
Starting Intercept Msg1 Attacker:
start
ERROR: i_recv_m4: decryption of part intended for initiator failed: aead_dec: key or ad
mismatch
Finished Intercept Msg1 Attacker:
Starting Intercept Msg2 Attacker:
start
ERROR: r_send_m4: decryption of part intended for responder failed: aead_dec: key or ad
mismatch
Finished Intercept Msg2 Attacker:
Starting Impersonate Responder to Initiator Attacker:
start
ERROR: srv_send_m3: principal names in encrypted parts do not match with principal names in
unencrypted part
Finished Impersonate Responder to Initiator Attacker:
```

The trace entries are similar as those in the trace of the benign attacker in the original protocol depicted in Appendix A.1 except for the differences in the message structure and in the data associated with events. The output resulting from the execution of the attacks on the original protocol in the improved model shows that the modifications to the protocol prevent them. The *intercept and replay* attacks fail in the improved model due to a mismatch in the associated data, which happens because the attacker replays ciphertexts in a context different to the intended one. The impersonation attack fails because the server performs checks on the conversation identifier and principal identities in the encrypted and unencrypted parts of the third message and detects inconsistencies.

B Output of the Yahalom Model

```
======================
Yahalom
======================
Starting Benign Attacker:
start
new long term key session of alice with server
new long term key session of bob with server
installed long term key of alice at server
installed long term key of bob at server
 0. Generated AE YLM.lt_key(0)
    Label: [alice;server]
    Usage: AE YLM.lt_key
 1. SetState alice:
    Session 0(v0): ((APP | (p_key_sess | (server | AE YLM.lt_key(0)))))
 2. Generated AE YLM.lt_key(2)
    Label: [bob;server]
    Usage: AE YLM.lt_key
 3. SetState bob:
    Session 0(v0): ((APP | (p_key_sess | (server | AE YLM.lt_key(2)))))
 4. SetState server:
    Session 0(v0): ((APP | (srv_sess | (alice | AE YLM.lt_key(0)))))
 5. SetState server:
    Session 0(v0): ((APP | (srv_sess | (alice | AE YLM.lt_key(0)))))
    Session 1(v0): ((APP | (srv_sess | (bob | AE YLM.lt_key(2)))))
 6. Generated NONCE YLM.nonce_a(6)
    Label: Public
    Usage: NONCE YLM.nonce_a
 7. Event alice: initiate(alice,bob,server,NONCE YLM.nonce_a(6))
 8. Message alice->bob: (msg1 | (alice | NONCE YLM.nonce_a(6)))
 9. SetState alice:
    Session 0(v0): ((APP | (p_key_sess | (server | AE YLM.lt_key(0)))))
    Session 1(v0): ((APP | (i_sent_m1 | (bob | NONCE YLM.nonce_a(6)))))
10. Generated NONCE YLM.nonce_b(10)
    Label: [bob;alice;server]
    Usage: NONCE YLM.nonce_b
11. Event bob: req_key(alice,bob,server,NONCE YLM.nonce_a(6),NONCE YLM.nonce_b(10))
12. Message bob->server: (msg2 | (bob | aead_enc(AE YLM.lt_key(2),iv,(ev2 | (alice | (NONCE
YLM.nonce_a(6) | NONCE YLM.nonce_b(10)))),ev2)))
13. SetState bob:
    Session 0(v0): ((APP | (p_key_sess | (server | AE YLM.lt_key(2)))))
    Session 1(v0): ((APP | (r_sent_m2 | (alice | (server | NONCE YLM.nonce_b(10))))))
14. Generated AE YLM.comm_key(14)
    Label: [server;alice;bob]
    Usage: AE YLM.comm_key
```

```
15. Event server: send_key(alice,bob,server,NONCE YLM.nonce_a(6),NONCE YLM.nonce_b(10),AE YLM.
comm_key(14))
16. Message server->alice: (msg3 | (aead_enc(AE YLM.lt_key(0),iv,(ev3_i | (bob | (AE YLM.
comm_key(14) | (NONCE YLM.nonce_a(6) | NONCE YLM.nonce_b(10))))),ev3_i) | aead_enc(AE YLM.
lt_key(2),iv,(ev3_r | (alice | AE YLM.comm_key(14))),ev3_r)))
17. SetState server:
    Session 0(v0): ((APP | (srv_sess | (alice | AE YLM.lt_key(0)))))
    Session 1(v0): ((APP | (srv_sess | (bob | AE YLM.lt_key(2)))))
    Session 2(v0): ((APP | (srv_sent_m3 | (alice | (bob | AE YLM.comm_key(14))))))
18. Event alice: fwd_key(alice,bob,server,NONCE YLM.nonce_a(6),NONCE YLM.nonce_b(10),AE YLM.
comm_key(14))
19. Message alice->bob: (msg4 | (aead_enc(AE YLM.lt_key(2),iv,(ev3_r | (alice | AE YLM.
comm_key(14))),ev3_r) | aead_enc(AE YLM.comm_key(14),iv,(ev4 | NONCE YLM.nonce_b(10)),ev4)))
20. SetState alice:
    Session 0(v0): ((APP | (p_key_sess | (server | AE YLM.lt_key(0)))))
    Session 1(v0): ((APP | (i_sent_m4 | (bob | (server | AE YLM.comm_key(14))))))
21. Event bob: recv_key(alice,bob,server,NONCE YLM.nonce_b(10),AE YLM.comm_key(14))
22. SetState bob:
    Session 0(v0): ((APP | (p_key_sess | (server | AE YLM.lt_key(2)))))
    Session 1(v0): ((APP | (r_rcvd_m4 | (alice | (server | AE YLM.comm_key(14))))))
PROTOCOL RUN: Successful execution of Yahalom protocol.
Finished Benign Attacker:
```

Since the Yahalom protocol requires the same pre-shared keys as the Otway-Rees protocol, the long-term keys are again installed in the first five trace entries. The initiator `alice` then begins a protocol run with the responder `bob`, by triggering the event `initiate(alice,bob,server,NONCE YLM .nonce_a(6))` and sending a message to `bob`. Again, `bob` reacts by merging `alice`'s message into a request for the server to generate and distribute a session key. In response, the server generates a key `AE YLM.comm_key(14)` and sends key certificates for both users to `alice`, who then completes the protocol run by forwarding the key to `bob`. The session key is again stored in the final state sessions of the users.

C Output of the Denning-Sacco Model

C.1 Output and Trace of Benign Attacker

```
======================
Denning-Sacco
======================
Starting Benign Attacker:
start
generating private key for alice
generating private key for bob
generating private key for server
installing public key for alice at server
installing public key for bob at server
installing public key for server at alice
installing public key for server at bob
 0. Generated SIG DS.sig_key(0)
    Label: [alice]
    Usage: SIG DS.sig_key
 1. SetState alice:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(0))))
 2. Generated PKE DS.pke_key(2)
    Label: [bob]
    Usage: PKE DS.pke_key
 3. SetState bob:
    Session 0(v0): ((DecryptionKey | (DS.pke_key | PKE DS.pke_key(2))))
 4. Generated SIG DS.sig_key(4)
    Label: [server]
    Usage: SIG DS.sig_key
 5. SetState server:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(4))))
 6. SetState server:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(4))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (alice | vk(SIG DS.sig_key(0))))))
 7. SetState server:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(4))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (alice | vk(SIG DS.sig_key(0))))))
    Session 2(v0): ((EncryptionKey | (DS.pke_key | (bob | pk(PKE DS.pke_key(2))))))
 8. SetState alice:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(0))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (server | vk(SIG DS.sig_key(4))))))
 9. SetState bob:
    Session 0(v0): ((DecryptionKey | (DS.pke_key | PKE DS.pke_key(2))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (server | vk(SIG DS.sig_key(4))))))
10. Event alice: initiate(alice,bob)
```

```
11. Message alice->server: (msg1 | (alice | bob))
12. SetState alice:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(0))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (server | vk(SIG DS.sig_key(4))))))
    Session 2(v0): ((APP | (i_sent_m1 | bob)))
13. Message server->server: 13
14. Event server: certify(alice,bob,vk(SIG DS.sig_key(0)),pk(PKE DS.pke_key(2)),13,0)
15. Generated NONCE SIG_NONCE(15)
    Label: [server]
    Usage: NONCE SIG_NONCE
16. Message server->alice: (msg2 | (((cert_a | (alice | (vk(SIG DS.sig_key(0)) | 13))) | sign(
SIG DS.sig_key(4),NONCE SIG_NONCE(15),(cert_a | (alice | (vk(SIG DS.sig_key(0)) | 13))))) | ((
cert_b | (bob | (pk(PKE DS.pke_key(2)) | 13))) | sign(SIG DS.sig_key(4),NONCE SIG_NONCE(15),(
cert_b | (bob | (pk(PKE DS.pke_key(2)) | 13)))))))
17. SetState server:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(4))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (alice | vk(SIG DS.sig_key(0))))))
    Session 2(v0): ((EncryptionKey | (DS.pke_key | (bob | pk(PKE DS.pke_key(2))))))
    Session 3(v0): ((APP | (srv_sent_m2 | (alice | bob))))
18. Generated AE DS.comm_key(18)
    Label: Join [bob] [alice]
    Usage: AE DS.comm_key
19. Event alice: send_key(alice,bob,vk(SIG DS.sig_key(0)),pk(PKE DS.pke_key(2)),AE DS.comm_key
(18),13,2)
20. Generated NONCE SIG_NONCE(20)
    Label: [alice]
    Usage: NONCE SIG_NONCE
21. Generated NONCE PKE_NONCE(21)
    Label: [alice]
    Usage: NONCE PKE_NONCE
22. Message alice->bob: (msg3 | (((cert_a | (alice | (vk(SIG DS.sig_key(0)) | 13))) | sign(SIG
 DS.sig_key(4),NONCE SIG_NONCE(15),(cert_a | (alice | (vk(SIG DS.sig_key(0)) | 13))))) | (((
cert_b | (bob | (pk(PKE DS.pke_key(2)) | 13))) | sign(SIG DS.sig_key(4),NONCE SIG_NONCE(15),(
cert_b | (bob | (pk(PKE DS.pke_key(2)) | 13))))) | pke_enc(pk(PKE DS.pke_key(2)),NONCE
PKE_NONCE(21),((comm_key | (AE DS.comm_key(18) | 13)) | sign(SIG DS.sig_key(0),NONCE SIG_NONCE
(20),(comm_key | (AE DS.comm_key(18) | 13))))))))
23. SetState alice:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(0))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (server | vk(SIG DS.sig_key(4))))))
    Session 2(v0): ((APP | (i_sent_m3 | (bob | AE DS.comm_key(18)))))
24. Event bob: accept_key(alice,bob,vk(SIG DS.sig_key(0)),pk(PKE DS.pke_key(2)),AE DS.comm_key
(18),13,4)
25. SetState bob:
    Session 0(v0): ((DecryptionKey | (DS.pke_key | PKE DS.pke_key(2))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (server | vk(SIG DS.sig_key(4))))))
    Session 2(v0): ((APP | (r_rcvd_m3 | (alice | AE DS.comm_key(18)))))
PROTOCOL RUN: Successful execution of Denning-Sacco protocol.
Finished Benign Attacker:
```

This trace shows a successful execution of the Denning-Sacco protocol in our DY* model. In the trace entries 0 to 9, the private and public keys used to run the protocol are set up in the states of the respective principals. The protocol then begins with the `initiate(alice,bob)` event triggered by the initiator `alice`, who wants to exchange a key with `bob`. Ultimately, the protocol leads to the exchange of a communication key `AE DS.comm_key (18)`, which is stored in the final protocol state sessions of `alice` and `bob`. The time-based properties of the protocol are captured in terms of the last two fields of the events *certify*, *send key*, and *accept key*, where the first field is the timestamp contained in the certificates and the second field is the clock counter of the corresponding clock at the time of the event. As the trace shows, the clock counter is always in the expected validity window of the timestamp, which shows that our model succeeds to model the protocol's time-dependent properties.

C.2 Output and Trace of Fake Certificate Attacker

```
Starting Fake Certificate Attacker:
start
generating private key for alice
generating private key for bob
generating private key for eve
generating private key for server
installing public key for alice at server
installing public key for bob at server
installing public key for eve at server
installing public key for server at alice
installing public key for server at bob
compromised eve's secret key
compromised server's sign key
queried eve's public key in server state
queried alice's public key in server state
 0. Generated SIG DS.sig_key(0)
    Label: [alice]
    Usage: SIG DS.sig_key
 1. SetState alice:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(0))))
 2. Generated PKE DS.pke_key(2)
    Label: [bob]
    Usage: PKE DS.pke_key
 3. SetState bob:
    Session 0(v0): ((DecryptionKey | (DS.pke_key | PKE DS.pke_key(2))))
 4. Generated PKE DS.pke_key(4)
    Label: [eve]
    Usage: PKE DS.pke_key
 5. SetState eve:
    Session 0(v0): ((DecryptionKey | (DS.pke_key | PKE DS.pke_key(4))))
 6. Generated SIG DS.sig_key(6)
    Label: [server]
    Usage: SIG DS.sig_key
 7. SetState server:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(6))))
 8. SetState server:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(6))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (alice | vk(SIG DS.sig_key(0))))))
 9. SetState server:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(6))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (alice | vk(SIG DS.sig_key(0))))))
    Session 2(v0): ((EncryptionKey | (DS.pke_key | (bob | pk(PKE DS.pke_key(2))))))
10. SetState server:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(6))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (alice | vk(SIG DS.sig_key(0))))))
    Session 2(v0): ((EncryptionKey | (DS.pke_key | (bob | pk(PKE DS.pke_key(2))))))
    Session 3(v0): ((EncryptionKey | (DS.pke_key | (eve | pk(PKE DS.pke_key(4))))))
11. SetState alice:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(0))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (server | vk(SIG DS.sig_key(6))))))
12. SetState bob:
    Session 0(v0): ((DecryptionKey | (DS.pke_key | PKE DS.pke_key(2))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (server | vk(SIG DS.sig_key(6))))))
13. Compromised eve(0)(0)
14. Compromised server(0)(0)
15. Compromised server(3)(0)
16. Compromised server(1)(0)
17. Event alice: initiate(alice,bob)
18. Message alice->server: (msg1 | (alice | bob))
19. SetState alice:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(0))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (server | vk(SIG DS.sig_key(6))))))
    Session 2(v0): ((APP | (i_sent_m1 | bob)))
20. Message *->*: 20
21. Generated NONCE SIG_NONCE(21)
    Label: Public
    Usage: NONCE SIG_NONCE
22. Message *->*: NONCE SIG_NONCE(21)
23. Message server->alice: (msg2 | (((cert_a | (alice | (vk(SIG DS.sig_key(0)) | 20))) | sign(
SIG DS.sig_key(6),NONCE SIG_NONCE(21),(cert_a | (alice | (vk(SIG DS.sig_key(0)) | 20))))) | ((
cert_b | (bob | (pk(PKE DS.pke_key(4)) | 20))) | sign(SIG DS.sig_key(6),NONCE SIG_NONCE(21),(
cert_b | (bob | (pk(PKE DS.pke_key(4)) | 20)))))))
24. Generated AE DS.comm_key(24)
    Label: Join [eve] [alice]
    Usage: AE DS.comm_key
25. Event alice: send_key(alice,bob,vk(SIG DS.sig_key(0)),pk(PKE DS.pke_key(4)),AE DS.comm_key
(24),20,1)
26. Generated NONCE SIG_NONCE(26)
    Label: [alice]
    Usage: NONCE SIG_NONCE
27. Generated NONCE PKE_NONCE(27)
    Label: [alice]
    Usage: NONCE PKE_NONCE
28. Message alice->bob: (msg3 | (((cert_a | (alice | (vk(SIG DS.sig_key(0)) | 20))) | sign(SIG
 DS.sig_key(6),NONCE SIG_NONCE(21),(cert_a | (alice | (vk(SIG DS.sig_key(0)) | 20))))) | (((
cert_b | (bob | (pk(PKE DS.pke_key(4)) | 20))) | sign(SIG DS.sig_key(6),NONCE SIG_NONCE(21),(
cert_b | (bob | (pk(PKE DS.pke_key(4)) | 20))))) | pke_enc(pk(PKE DS.pke_key(4)),NONCE
PKE_NONCE(27),((comm_key | (AE DS.comm_key(24) | 20)) | sign(SIG DS.sig_key(0),NONCE SIG_NONCE
(26),(comm_key | (AE DS.comm_key(24) | 20))))))))
29. SetState alice:
    Session 0(v0): ((SigningKey | (DS.sig_key | SIG DS.sig_key(0))))
    Session 1(v0): ((VerificationKey | (DS.sig_key | (server | vk(SIG DS.sig_key(6))))))
    Session 2(v0): ((APP | (i_sent_m3 | (bob | AE DS.comm_key(24)))))
```

```
PROTOCOL RUN: Successful execution of Denning-Sacco protocol.
Finished Fake Certificate Attacker:
```

This trace displays the execution of a *fake certificate* attack by a compromised server. The setup phase of the attack is covered by the trace entries 0 to 16 and includes, in addition to the setup phase in the benign scenario depicted in Appendix C.1, the compromise of certain state sessions of the server and another principal `eve` by the attacker. The initiator is honest and therefore begins the protocol with the *initiate* event. The root of the attack can be found in trace entry 23, where the server sends a false certificate to `alice` signed with the servers sign key `SIG DS.sig_key(6)`, claiming that the public key `pk(PKE DS.pke_key(4))` of `eve` belongs to `bob`. In trace entry 25, `alice` triggers *send key*, which means that it has accepted this false certificate. It sends the communication key to `bob`, but encrypted under the public key of `eve` from the public key certificate, so that the adversary can intercept the message and decrypt the certificate containing the communication key with `eve`'s compromised secret key. The value of the clock counter is 1 at the time the *send key* event is triggered, and 2 after `alice` sends the third message to `bob` in trace entry 28. When the adversary obtains the key, the timestamp is thus still valid, giving the adversary the ability to replay the key to another user without them noticing. For our model, this attack shows that timestamps cannot protect against replays in scenarios where the attacker controls one or more principals actively involved in the protocol.

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