Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatik
Computer Science

Master Thesis

Efficient Interactive Construction of Machine-Checked Protocol Security Proofs in the Context of Dynamically Compromising Adversaries

Martin Schaub

April 2011

Supervisor: Simon Meier
Professor: Prof. Dr. Basin

Information Security
Swiss Federal Institute of Technology
CH-8092 Zürich

Acknowledgements

This thesis would not have been possible without the help of my supervisor Simon Meier. He not only supported me with his technical expertise in the topic of security protocol verification, but also by motivating me during difficult periods of this thesis. Moreover, his reviews of this thesis report and the resulting feedback helped me a lot to improve my English writing and especially my English technical writing skills.

I also want to thank Dr. Cas Cremers for his support during this thesis. Especially, I want to point out the interesting discussions we had on the subject of compromising adversaries. Without them I would have missed many interesting points about this topic and this thesis would be different.

This thesis represents not only the end of my master studies at ETH Zürich, but also to an almost 10 year education in computer science and information technology. On this path I had the honor to meet extraordinary people from whom I have learned a lot. Moreover, I also enjoyed support and inspiration from them for climbing the steps of the Swiss higher education system from an apprenticeship, towards my bachelor degree at the university of applied science up to the master degree at the ETH. I want to use this occasion to thank them. Notably the following people:

I want to thank Clemens Hofmann, Hubert Kohler, Oliver Höller, and Oliver Stadler for supporting me during my apprenticeship at IBM. Without their motivation I would have never started the university of applied sciences and thereby would have missed the opportunity to reveal my interest in computer sciences.

During my bachelor studies at FHNW I had the chance of attend lectures on the topic of software engineering and software design from Prof. Dr. Denzler. Those have been very inspiring and showed me the real beauty of software. I also want to thank Prof. Dr. Vogel for supervising a semester thesis and my bachelor thesis. These two professors inspired me to apply for a master study at ETH.

Without the support of my parents Thomas and Elisabeth, and my brother Dominik this education would not have been possible.

During my education path I also found good friends who joined me on some stages or even the whole way. Notably I want to thank Samuel Hänger and Michael Haspra for joining me on the long way. Moreover, I want to thank them and Noah Heuser, Max Urech, Malte Schwerhoff, and Manuel Kläy for making the ETH even more interesting with our conversations.

Last but not least I want to thank my girlfriend Estefani for her constant support.

Abstract

In this master thesis, we provide a method for the efficient interactive construction of machine-checkable protocol security proofs in the context of compromising adversaries. In our method, we first specify a protocol according to our security protocol model. That means we specify which data is sent and received. Moreover, we also specify at which points in the protocol execution, data is stored in an unprotected system state, when a session-key is established, and which data is a randomly generated number. In our security protocol model, the adversary can reveal data from all three categories. Moreover he can also reveal long-term secrets of protocol participants.

In the next step of our proof method, we specify the capabilities of the adversary. Based on these capabilities, we specify security properties. Before proving them, we establish a property per message meant to be secret. This property lists the reveals the adversary has to perform in order to learn the message. In the secrecy property proofs, we can use the property of the according message to verify that non of the reveals are allowed and therefore the message is secret. Moreover we use secrecy properties for shortening authentication property proofs.

We apply this proof method successfully to various protocols from the literature as well as artificial protocol creations. The security proof construction times depend on how much short-term data is available. In protocols without short-term data they are comparable to other methods for interactive construction of machine-checkable protocol security proofs.

For our proof method, we use a method that provides efficient construction of machine-checkable protocol security proofs for a Dolev-Yao style adversary as base. From the according security protocol model, we first removed the Dolev-Yao style adversary and replaced it with the compromising adversaries framework. Then we adopted the proof method to support the explicit capabilities of the adversary.

Contents

List of Figures

List of Tables

1. Introduction

There is a demand for certified security protocols because they are used in critical distributed systems. To ensure that security protocols are correct, certification processes requires formal proofs of the security properties that a security protocol achieves. However, due to the complexity of such *protocol security proofs* there have been mistakes in them. Therefore, it seems prudent to require *machine-checked proofs* to provide strong correctness guarantees for these proofs.

Meier, Cremers, and Basin propose in [19] an efficient construction method for machine-checked protocol security proofs. They provide support for interactive proof construction in the theorem prover Isabelle/HOL [20] and a tool that generates Isabelle/HOL proof scripts automatically. They report that even their interactive proof construction times are two orders of magnitudes faster than comparable interactive approaches. They call their reasoning method *decryption chain reasoning*. It uses a strong protocol independent invariant that provides the cases on how the adversary could have learned a message. As adversary model, they use a Dolev-Yao style adversary.

In the Dolev-Yao adversary model, participants are always either honest or they are compromised from the start. Moreover, the adversary cannot reveal short-term data like state, session-key, and random generated numbers. However, modern security protocols are designed to be resilient against these forms of short-terms reveal and dynamic compromises of a participant. Thereby they can achieve properties like perfect forward secrecy and resilience against session-key or state reveal. Hence the adversary model should reflect those capabilities such that we are able to prove that a protocol satisfies these properties.

In computational approaches stronger adversary models such as [5–8] by Bellare, Rogaway, Pointcheval, Canetti, and Krawczyk, with the ability to reveal long-term keys, session-keys, states, and intermediate computations are used. However, due to the lack of machine-checked proofs and the complexity of these adversary models, there are flaws reported in their protocol security proofs such as in [9, 15, 16].

Recently, Cremers and Basin propose in their symbolic model a framework they call *compromising adversaries* [3,4]. With this framework adversaries with the ability to reveal long-term keys, session-keys, states, and randomly generated numbers can be modeled. They implement compromising adversaries in the symbolic protocol verification tool Scyther [10,12]. However, Scyther has no support for generating or constructing machine-checked proofs.

We develop a method to construct machine-checked protocol security proofs for compromising adversaries. We use the work from [19] as base. From it we have an efficient method for constructing machine-checked protocol security proofs. We extend the security protocol model such that compromising adversaries are supported. In this adopted security protocol model we express security properties and construct protocol security proofs of case studies.

1.1. Contributions

First, we extend both the security protocol model as well as its Isabelle/HOL formalization from [19] with support for compromising adversaries

Second, we show how we express security properties in our adopted model. For this we formalized a method to distinguish intended communication partners from other participants and formalized the capabilities of the compromising adversaries for our security protocol model. Our method of defining intended communication partners can express all other such definitions known to us. Moreover, our definition of the capabilities of the adversary is expansible along the three dimensions of compromising adversaries, i.e. whose data is revealed, when the reveal occurs, and which kind of data is revealed.

Third, we provide an interactive proof method based on decryption chain reasoning for the adopted model. This method allows efficient protocol security proofs in the context of compromising adversaries.

Fourth, we demonstrate the effectiveness of our proof method on protocols from the literature as well as artificial protocol creations.

1.2. Organization

We present our security protocol model in Chapter 2. In Chapter 3, we present how we express security properties in this model. In Chapter 4, we introduce our proof method. Thereby, we present how we construct protocol security proofs. In Chapter 5, we demonstrate the effectiveness of our proof method on well-known protocols from the literature. Finally, we discuss related work in Chapter 6, and we draw conclusions and discuss future work in Chapter 7. We provide the Isabelle/HOL theories of all our formalizations at [1].

Our work heavily relies on concepts that Meier, Cremers, and Basin propose in [19]. In this report we explain only the concepts that we have adopted and adumbrate concepts that we have used without modifications. In Appendix B we provide a list of our concepts.

1.3. Notational Preliminaries

The notation used in this thesis is based on [19]. We recall it here for the convenience of our readers. We also use font and variable conventions, which we list in Appendix A.

For a binary relation R, we denote its reflexive transitive closure by R^*. We define the identity relation Id as $\{(x, x) \mid \mathit{true}\}$. Let f be a function. We write $\mathit{dom}(f)$ and $\mathit{ran}(f)$ to denote f's domain and range, respectively. We write $f[a \mapsto b]$ to denote f's update, defined as the function f' where $f'(x) = b$ when $x = a$ and $f'(x) = f(x)$ otherwise. We write $f : X \nrightarrow Y$ to denote a partial function mapping all elements in $\mathit{dom}(f) \subseteq X$ to elements from Y and all elements in $X \setminus \mathit{dom}(f)$ to the undefined value $\bot$, different from all other values.

For any set S, $\mathcal{P}(S)$ denotes the power set of S and S^* denotes the set of finite sequences of elements from S. We write $\langle s_1, \ldots, s_n \rangle$ to denote the sequence of elements s_1 through s_n. For a sequence s of length $|s|$ and $1 \leq i \leq |s|$, we write s_i to denote the i-th element of s. We write $s \mathbin{\hat{}} s'$ for the concatenation of sequences s and s'. Abusing set notation, we write $e \in s$ iff $\exists i.\ s_i = e$. We write $x <_s y$ to denote that x precedes y in the sequence s, i.e., $\exists a\, b.\ s = a \mathbin{\hat{}} b \wedge x \in a \wedge y \in b$. Note that $<_s$ is a strict total order on the elements in s if s is duplicate-free. We write $\mathit{last}(s)$ to denote the last element of the sequence s, defined as $\mathit{last}(s \mathbin{\hat{}} \langle e \rangle) = e$ and $\mathit{last}(\langle\rangle) = \bot$. We define $\mathit{set}(s)$ as $\{a \mid a \in s\}$. For a sequence s and a predicate p, we write $\mathit{filter}(p, s)$ to denote the sequence that we get by removing all elements x from s for which $\neg p(x)$ holds. We define it as

$$\mathit{filter}(p, s) \stackrel{\text{def}}{=} \begin{cases} \langle e \rangle \mathbin{\hat{}} \mathit{filter}(p, s') & \text{if } s = \langle e \rangle \mathbin{\hat{}} s' \text{ and } p(e) \\ \mathit{filter}(p, s') & \text{if } s = \langle e \rangle \mathbin{\hat{}} s' \text{ and } \neg p(e) \\ \langle \rangle & \text{if } s = \langle \rangle \ . \end{cases}$$

For a term t, the variables of t are denoted by $FV(t)$. We use standard notation for manipulating terms [2].

2. Security Protocol Model

In this chapter, we introduce our security protocol model, which is an extension of the security protocol model of [19]. Therefore, many definitions are the same. We repeat these definitions to make this thesis self-contained. However, we only explain new and changed definitions.

2.1. Protocol Specification

We assume given the pairwise-disjoint sets *Const*, *Fresh*, and *Var* denoting constants, messages to be freshly generated (nonces, coin flips, etc.), and variables. We further assume that the set of variables *Var* is partitioned into two sets *AVar* and *MVar*, denoting *agent variables* and *message variables*. We define the set *Pat* of *message patterns* as

$$\begin{aligned} Pat ::= {} & Const \mid Fresh \mid Var \mid \mathsf{h}(Pat) \mid (Pat, Pat) \mid \\ & \{\!|Pat|\!\}_{Pat} \mid Pat^{-1} \mid \mathsf{k}_{Pat,Pat} \mid \mathsf{pk}_{Pat} \mid \mathsf{sk}_{Pat} \ . \end{aligned}$$

We distinguish between long-term keys stored by an agent and short-term data that is available per session. Compromising adversaries can reveal both kind of data, i.e. they can perform reveals not only on long-term keys, but also on *randomly generated numbers*, *unprotected system states*, and *session-keys*. To specify which short-term data is currently available, we introduce the notion of a *note*. To define to which of the three types of short-term data the data belongs to, we introduce the notion of a *note type*. We define the set *NoteType* of note types as

$$NoteType ::= \mathsf{RandGen} \mid \mathsf{State} \mid \mathsf{SessKey} \ .$$

Let *Label* be a set of labels. We define the set *RoleStep* of *role steps* as

$$RoleStep ::= \mathsf{Send}_{Label}(Pat) \mid \mathsf{Recv}_{Label}(Pat) \mid \mathsf{Note}_{Label}(NoteType, Pat) \ .$$

A *note role step* $\mathsf{Note}_l(nt, pt)$ denotes that the message corresponding to the message pattern pt is of note type nt and currently available at this role step. The adversary can reveal the message by performing a reveal for short-term data of type nt.[1] Note that we use tupling to make multiple messages available for a reveal.

A *role* is a duplicate-free, finite sequence R of role steps such that

$$\begin{aligned} & \forall st \in R.\ \forall pt.\ (\ (\exists l\ nt.\ st = \mathsf{Note}_l(nt, pt)) \lor (\exists l.\ st = \mathsf{Send}_l(pt))\ ) \Rightarrow \\ & \qquad \forall v \in FV(pt) \cap MVar.\ \exists l', pt'.\mathsf{Recv}_{l'}(pt') <_R st \land v \in FV(pt')) \end{aligned}$$

holds. This states that every message variable of a role must be instantiated in a receive step before its contents can be used in a send or a note step. We denote the set of all roles by *Role*.

A *protocol* is a set of roles. We denote the set of all protocols by *Protocol*.

[1] We were inspired by the generate, sesskey, and state agent events from [4] to use notes to model short-term data.

Example 2.1: CRN Protocol

We illustrate protocol specifications with an extended version of the challenge-response protocol **CR** used as a running example in [19].

Let $s \in AVar$, $k \in Fresh$, and $v \in MVar$. We define $\mathbf{CRN} \overset{\text{def}}{=} \{C, S\}$, where

$$\begin{aligned} C &\overset{\text{def}}{=} \langle\ \mathsf{Send}_1(\{\!|k|\!\}_{\mathsf{pk}_s}),\quad \mathsf{Recv}_2(\mathsf{h}(k)),\quad \mathsf{Note}_3(\mathsf{SessKey}, k)\ \rangle \\ S &\overset{\text{def}}{=} \langle\ \mathsf{Recv}_1(\{\!|v|\!\}_{\mathsf{pk}_s}),\quad \mathsf{Note}_2(\mathsf{State}, v),\quad \mathsf{Send}_3(\mathsf{h}(v))\ \rangle\ . \end{aligned}$$

In this protocol, a client, modeled by the C role, chooses a fresh session-key k. It sends this session-key encrypted with the public key of the server with whom he wants to share k. The server, modeled by the role S, stores the received value v in its unprotected memory. This gives the adversary the opportunity to reveal v by a reveal of type State. Then the server confirms the receipt of v by sending the hash of v. Finally, the client receives the hash of k and has a session-key k, which the adversary can learn by a reveal of type $\mathsf{SessKey}$.

2.2. Protocol Execution

During the execution of a protocol **P**, agents may execute any number of instances of **P**'s roles in parallel. We call each role instance a *thread*.

2.2.1. Messages

We assume an infinite set *TID* of thread identifiers. We use the thread identifiers to distinguish between fresh messages generated by different threads. For a thread identifier i and a message $n \in Fresh$ to be freshly generated, we write $n\sharp i$ to denote the fresh message generated by the thread i for n. We overload notation and for a set N; we write $N\sharp TID$ to denote $\{n\sharp TID \mid n \in N, i \in TID\}$. We assume given a set *Agent* of agent names. We define the set *Msg* of *messages*

$$\begin{aligned} Msg ::= {} & Const \mid Fresh\sharp TID \mid Agent \mid \mathsf{h}(Msg) \mid (Msg, Msg) \mid \\ & \{\!|Msg|\!\}_{Msg} \mid \mathsf{k}_{Msg,Msg} \mid \mathsf{pk}_{Msg} \mid \mathsf{sk}_{Msg}\ . \end{aligned}$$

We assume the existence of an inverse function on messages, where k^{-1} denotes the inverse key of k. We have $\mathsf{pk}_x^{-1} = \mathsf{sk}_x$ and $\mathsf{sk}_x^{-1} = \mathsf{pk}_x$ for every message x, and $m^{-1} = m$ for all other messages m.

2.2.2. System State

For each thread, the system state stores a *program counter* as a sequence *done* of already processed role steps and a sequence *todo* of role steps still to be executed. The processing of a role step might have no outside effect. We say that a role step was skipped, if its processing has no outside effect, and keep track of all skipped role steps. Therefore, the system state also stores a set *skipped* of skipped role steps for each thread. We model this information as a partial function

$$th\colon TID \rightharpoonup (RoleStep^*, RoleStep^*, \mathcal{P}(RoleStep))\ .$$

Its domain $dom(th)$ denotes the identifiers of all threads in the system. For a thread $i \in dom(th)$ the first component of $th(i)$ is *done*, the second *todo*, and the third *skipped*. We call *th* a *thread pool* and define the set *ThreadPool* as the set of all thread pools. Furthermore, the system state contains a *variable store* $\sigma : Var \times TID \rightarrow Msg$ storing for each variable

v and thread identifier i the contents $\sigma(v,i)$ assigned to v by thread i. We define the set of all variable stores as $\mathit{Store} \overset{\text{def}}{=} \mathit{Var} \times \mathit{TID} \to \mathit{Msg}$. During the execution of a thread i in the context of a variable store σ, a message pattern pt is *instantiated* by the function inst: $(\mathit{Pat} \times \mathit{TID}) \to \mathit{Msg}$ to the message $\mathit{inst}_{\sigma,i}(pt)$. We define the function inst in Figure 2.1.

$$\mathit{inst}_{\sigma,i}(pt) \overset{\text{def}}{=} \begin{cases} pt & \text{if } pt \in \mathit{Const} \\ pt\sharp i & \text{if } pt \in \mathit{Fresh} \\ \sigma(pt,i) & \text{if } pt \in \mathit{Var} \\ \mathsf{h}(\mathit{inst}_{\sigma,i}(x)) & \text{if } pt = \mathsf{h}(x) \\ (\mathit{inst}_{\sigma,i}(x), \mathit{inst}_{\sigma,i}(y)) & \text{if } pt = (x,y) \\ \{\!|\mathit{inst}_{\sigma,i}(x)|\!\}_{(\mathit{inst}_{\sigma,i}(k))} & \text{if } pt = \{\!|x|\!\}_k \\ (\mathit{inst}_{\sigma,i}(x))^{-1} & \text{if } pt = x^{-1} \\ \mathsf{k}_{\mathit{inst}_{\sigma,i}(a)\,,\,\mathit{inst}_{\sigma,i}(b)} & \text{if } pt = \mathsf{k}_{a,b} \\ \mathsf{pk}_{\mathit{inst}_{\sigma,i}(a)} & \text{if } pt = \mathsf{pk}_a \\ \mathsf{sk}_{\mathit{inst}_{\sigma,i}(a)} & \text{if } pt = \mathsf{sk}_a\ . \end{cases}$$

Figure 2.1.: Definition of the function *inst*.

A *trace* is a sequence of *basic events*.

$$\mathit{BasicEvent} ::= \mathsf{St}(\mathit{TID}, \mathit{RoleStep}) \mid \mathsf{Ln}(\mathcal{P}(\mathit{Msg})) \mid \mathsf{LKR}(\mathit{Agent})$$

Additionally to the *basic step event* and the *basic learn event*, we introduce a *long-term key reveal event*. For an agent a, the long-term key reveal event $\mathsf{LKR}(a)$ models the reveal of the long-term keys of the agent a to the adversary.

A *system state* is a triple (tr, th, σ) from the set $\mathit{State} \overset{\text{def}}{=} \mathit{Trace} \times \mathit{ThreadPool} \times \mathit{Store}$.

2.2.3. Adversary Knowledge

Given a trace tr, we define the associated *adversary knowledge* $\mathit{knows}(tr)$ as the set of messages that the adversary learns from the basic learn events in tr.

$$\mathit{knows}(tr) \overset{\text{def}}{=} \bigcup_{\mathsf{Ln}(M) \in tr} M$$

Our security protocol model ensures that the adversary always learns the *initial adversary knowledge* as the first basic event in the trace. It consists of all public information, i.e. constants, agent names, and public keys.

$$\mathit{IK}_0 \overset{\text{def}}{=} \mathit{Const} \cup \mathit{Agent} \cup \{\mathsf{pk}_a \mid a \in \mathit{Agent}\}$$

We define the function $\mathit{split} : \mathit{Msg} \to \mathcal{P}(\mathit{Msg})$, such that $\mathit{split}(m)$ denotes the set of all messages that can be obtained from m by projecting pairs.

$$\mathit{split}(m) \overset{\text{def}}{=} \begin{cases} \{m\} \cup \mathit{split}(x) \cup \mathit{split}(y) & \text{if } m = (x,y) \\ \{m\} & \text{otherwise} \end{cases}$$

We use $\mathit{newMsgs}_{tr}(m)$ to denote the *new messages* learned by the adversary when seeing the message m in the context of the trace *tr*.

$$\mathit{newMsgs}_{tr}(m) \overset{\text{def}}{=} \mathit{split}(m) \setminus \mathit{knows}(tr)$$

To obtain all secret long-term keys of an agent, we define the function $\mathit{longTermKeys}\colon \mathit{Agent} \to \mathcal{P}(\mathit{Msg})$. $\mathit{longTermKeys}(a)$ denotes the set of secret long-term keys of an agent $a \in \mathit{Agent}$. Those keys are its long-term private key sk_a and all long-term symmetric keys $\mathsf{k}_{a,b}$ and $\mathsf{k}_{b,a}$ shared with any other agent b.

$$\mathit{longTermKeys}(a) \overset{\text{def}}{=} \{\mathsf{sk}_a\} \cup \bigcup_{b \in \mathit{Agent}} \{\mathsf{k}_{a,b}, \mathsf{k}_{b,a}\} \ .$$

2.2.4. Transition System

For a protocol $\mathbf{P}$, the *state transition relation* $\longrightarrow$ is defined by the transition rules in Figure 2.2.

A $\mathsf{Note}_l(nt, pt)$ role step of a thread i gives the adversary the opportunity to learn the message $\mathit{inst}_{\sigma,i}(pt)$ by performing a reveal of type *nt*. He can also decide to not take this opportunity. We reflect these two possibilities as non-deterministic choice of the transition rules Compr and SkipCompr whenever the next role step of a thread i is a note step.

A Compr transition models that the adversary reveals the message $\mathit{inst}_{\sigma,i}(pt)$ of type *nt* the thread i has currently available. We record this reveal by adding the basic events $\mathsf{St}(i, \mathsf{Note}_l(nt, pt))$ and $\mathsf{Ln}(\mathit{newMsgs}_{tr}(\mathit{inst}_{\sigma,i}(pt)))$ to the trace. They denote that the adversary performed a *nt* reveal on the thread i at the note step and thereby learns the new parts of the instantiated message pattern. Moreover, we remove the role step $\mathsf{Note}_l(nt, pt)$ from the head of the role steps *todo* and append it to the role steps *done* to express the advance of the program counter.

A SkipCompr transition models that the adversary does not reveal the information of type *nt* that the thread i currently has available. Hence, the trace remains unchanged and the processing of the role step has no outside effect. Thus we add the note step $\mathsf{Note}_l(nt, pt)$ to the set of skipped steps *skipped*. Moreover, to express the advance of the program counter we remove the role step $\mathsf{Note}_l(nt, pt)$ from the head of the role steps *todo* and append it to the role steps *done* to express the advance of the program counter.

Note that in our security protocol model, the adversary can only reveal short-term data at note steps. Hence if a note step was processed and not skipped, the adversary has performed a reveal. We call such reveals *short-term data reveals*. For a note step *nt* we also use the name *nt reveal*, i.e. a *session-key reveal*, *state reveal*, or *random number generator reveal*.

A LKR transition models a long-term key reveal of an agent by which the adversary learns all secret long-term keys of the agent. There is at most one long-term key reveal possible per agent. The reason is that the adversary does not have an advantage by performing such a reveal multiple times on the same agent, because he cannot learn anything new after the first long-term key reveal.

There is no transition rule for creating new threads. Instead we consider all possible sets of new threads in the set of *initial states* $Q_0(\mathbf{P})$ of our system.

$$\begin{aligned} Q_0(\mathbf{P}) \overset{\text{def}}{=} \{(\langle \mathsf{Ln}(\mathit{IK}_0)\rangle,\ th,\ \sigma) \mid\ & (\forall v \in \mathit{AVar},\ i \in \mathit{TID}.\ \sigma(v,i) \in \mathit{Agent}) \land \\ & (\forall i \in \mathit{dom}(th).\ \exists R \in P.\ th(i) = (\langle\rangle, R, \emptyset))\} \ . \end{aligned}$$

In each initial state $(tr, th, \sigma) \in Q_0(\mathbf{P})$, the variable store σ is defined such that every agent variable is instantiated with an agent name and each message variable is instantiated with

$$\frac{th(i) = (done, \langle \mathsf{Send}_l(pt)\rangle \,\hat{}\, todo, skipped)}{\begin{array}{l}(tr, th, \sigma) \longrightarrow (tr \,\hat{}\, \langle\, \mathsf{St}(i, \mathsf{Send}_l(pt)),\ \mathsf{Ln}(newMsgs_{tr}(inst_{\sigma,i}(pt)))\,\rangle,\\ \qquad th[i \mapsto (done \,\hat{}\, \langle \mathsf{Send}_l(pt)\rangle, todo, skipped)],\ \sigma)\end{array}}\ \textsc{Send}$$

$$\frac{th(i) = (done, \langle \mathsf{Recv}_l(pt)\rangle \,\hat{}\, todo, skipped) \qquad inst_{\sigma,i}(pt) \in knows(tr)}{\begin{array}{l}(tr, th, \sigma) \longrightarrow (tr \,\hat{}\, \langle\, \mathsf{St}(i, \mathsf{Recv}_l(pt))\,\rangle,\\ \qquad th[i \mapsto (done \,\hat{}\, \langle \mathsf{Recv}_l(pt)\rangle, todo, skipped)],\ \sigma)\end{array}}\ \textsc{Recv}$$

$$\frac{th(i) = (done, \langle \mathsf{Note}_l(ty, pt)\rangle \,\hat{}\, todo, skipped)}{\begin{array}{l}(tr, th, \sigma) \longrightarrow (tr \,\hat{}\, \langle\, \mathsf{St}(i, \mathsf{Note}_l(ty, pt)),\ \mathsf{Ln}(newMsgs_{tr}(inst_{\sigma,i}(pt)))\,\rangle,\\ \qquad th[i \mapsto (done \,\hat{}\, \langle \mathsf{Note}_l(ty, pt)\rangle, todo, skipped)],\ \sigma)\end{array}}\ \textsc{Compr}$$

$$\frac{th(i) = (done, \langle \mathsf{Note}_l(ty, pt)\rangle \,\hat{}\, todo, skipped)}{\begin{array}{l}(tr, th, \sigma) \longrightarrow (tr,\\ \qquad th[i \mapsto (done \,\hat{}\, \langle \mathsf{Note}_l(ty, pt)\rangle, todo, \{\mathsf{Note}_l(ty, pt)\} \cup skipped)],\ \sigma)\end{array}}\ \textsc{SkipCompr}$$

$$\frac{a \in Agent \qquad \mathsf{LKR}(a) \notin tr}{(tr, th, \sigma) \longrightarrow (tr \,\hat{}\, \langle\, \mathsf{LKR}(a), \mathsf{Ln}(longTermKeys(a) \setminus knows(tr))\,\rangle,\ th,\ \sigma)}\ \text{LKR}$$

$$\frac{x, y \in knows(tr) \qquad (x, y) \notin knows(tr)}{(tr, th, \sigma) \longrightarrow (tr \,\hat{}\, \langle\, \mathsf{Ln}(\{(x, y)\})\,\rangle, th, \sigma)}\ \textsc{Pair}$$

$$\frac{m \in knows(tr) \qquad \mathsf{h}(m) \notin knows(tr)}{(tr, th, \sigma) \longrightarrow (tr \,\hat{}\, \langle\, \mathsf{Ln}(\{\mathsf{h}(m)\})\,\rangle, th, \sigma)}\ \textsc{Hash}$$

$$\frac{m, k \in knows(tr) \qquad \{\!|m|\!\}_k \notin knows(tr)}{(tr, th, \sigma) \longrightarrow (tr \,\hat{}\, \langle\, \mathsf{Ln}(\{\{\!|m|\!\}_k\})\,\rangle, th, \sigma)}\ \textsc{Encr}$$

$$\frac{\{\!|m|\!\}_k \in knows(tr) \qquad k^{-1} \in knows(tr)}{(tr, th, \sigma) \longrightarrow (tr \,\hat{}\, \langle\, \mathsf{Ln}(newMsgs_{tr}(m))\,\rangle, th, \sigma)}\ \textsc{Decr}$$

Figure 2.2.: Definition of the transition relation $\longrightarrow$.

some arbitrary message. The thread pool th is defined such that every thread $i \in dom(th)$ instantiates a role of $\mathbf{P}$ and has neither executed nor skipped any role step yet.

For a protocol $\mathbf{P}$, we define the set of *reachable states* as

$$\mathit{reachable}(\mathbf{P}) \overset{\text{def}}{=} \{q \in Q \mid \exists q_0 \in Q_0(\mathbf{P}).\ q_0 \longrightarrow^* q\}.$$

2.3. Modeling Freshly Generated Asymmetric Keypairs

In our security protocol model, we have defined the pattern and message constructors sk and pk for denoting the public- and private-key of an agent, i.e. sk_a or pk_a for $a \in \mathit{Agent}$. However, the use of this constructors is not restricted to agent names. Hence we can use them in other contexts. One example is to model a freshly generated asymmetric keypair for a given input, i.e. the input for the key generation algorithm.

In our security protocol model, sk and pk represent private functions. Therefore, the adversary cannot apply them to messages. As a result, the adversary cannot generate fresh asymmetric key pairs by himself. Whenever we model a protocol that uses freshly generated asymmetric key pairs, we have to ensure that the adversary learns the freshly generated asymmetric key pair when he learns the input of the key generation algorithm. Otherwise the protocol might be secure in our security protocol model although it has flaws. We demonstrate the problem on the following protocol.

Let $v, k \in \mathit{Fresh}$ and $w, h \in \mathit{MVar}$. We define the protocol $\mathbf{Bad} \overset{\text{def}}{=} \{C^{\mathbf{Bad}}, S^{\mathbf{Bad}}\}$ where

$$C^{\mathbf{Bad}} \overset{\text{def}}{=} \langle\ \mathsf{Send}_1((\{|v|\}_{\mathsf{pk}_k}, k)), \mathsf{Note}_2(\mathit{State}, k)\ \rangle \text{ and}$$
$$S^{\mathbf{Bad}} \overset{\text{def}}{=} \langle\ \mathsf{Recv}_1((\{|w|\}_{\mathsf{pk}_h}, h))\ \rangle\ .$$

Although we can prove the secrecy of $v\sharp i$ for a thread i that runs the role $C^{\mathbf{Bad}}$ in this protocol, it should clearly not be the case. The reason is that the adversary learns $k\sharp i$ from the send step. Hence, he should be able to generate the private key $\mathsf{sk}_{k\sharp i}$. This allows him to decrypt the message $\{|v\sharp i|\}_{\mathsf{pk}_{k\sharp i}}$ and learn $v\sharp i$. However, the adversary cannot generate sk_m or pk_m for an already known message m by himself because these functions are private.

We fix this problem by replacing the input k for the key generation algorithm by the triple $(k, \mathsf{pk}_k, \mathsf{sk}_k)$ in all the cases where k is not used as key. Hence the adversary can learn the generated fresh keys while he learns the input of the key generation algorithm. We illustrate the adoptions in the protocol $\mathbf{Ok} \overset{\text{def}}{=} \{C^{\mathbf{Ok}}, S^{\mathbf{Ok}}\}$ where

$$C^{\mathbf{Ok}} \overset{\text{def}}{=} \langle\ \mathsf{Send}_1((\{|v|\}_{\mathsf{pk}_k}, (k, \mathsf{pk}_k, \mathsf{sk}_k))), \mathsf{Note}_2(\mathit{State}, (k, \mathsf{pk}_k, \mathsf{sk}_k))\ \rangle \text{ and}$$
$$S^{\mathbf{Ok}} \overset{\text{def}}{=} \langle\ \mathsf{Recv}_1((\{|w|\}_{\mathsf{pk}_h}, (h, \mathsf{pk}_h, \mathsf{sk}_h)))\ \rangle\ .$$

Another solution is the extension of the message algebra with constructors denoting the generation of fresh public and private keys for a given message. For this solution, the transition system must also get extended by transition rules that allow the adversary to learn the fresh asymmetric keypair for an input message known to him.

3. Security Properties

In this chapter, we explain how we formalize security properties in the context of compromising adversaries.

First, we introduce how we express secrecy properties in general. Thereby, we realize that we have to restrict the adversary, otherwise he is fundamentally too strong. Therefore, we include an adversary model into the secrecy properties. This adversary model explicitly states the capabilities of the adversary. Afterwards, we apply the learned lessons from secrecy properties to authentication properties. Finally, we map our adversary capabilities to the capabilities of the adversary introduced by Cremers and Basin in [4]. This is of interest because they used their capabilities to model several well known adversary models from the literature.

3.1. Secrecy Properties

Secrecy properties state that the adversary cannot learn a message. We express a secrecy property as *secrecy predicate* that must hold for all reachable states.

In a secrecy predicate, we universally quantify over all threads running a specified role. We call the role, from whose perspective the property is formulated, the *test role*. We call the thread that is represented by the quantified variable the *test thread* to distinguish him from other threads. To obtain the role a thread executes, we define the partial function $\mathit{role}_{\mathit{th}} : \mathit{TID} \nrightarrow \mathit{Role}$. $\mathit{role}_{\mathit{th}}(i) = R$ denotes that the thread i executes an instance of the role R.[1]

$$\mathit{role}_{\mathit{th}}(i) \overset{\text{def}}{=} \begin{cases} \mathit{done} \mathbin{\hat{}} \mathit{todo} & \text{if } \mathit{th}(i) = (\mathit{done}, \mathit{todo}, \mathit{skipped}) \\ \perp & \text{otherwise} \end{cases}$$

We have to be careful for secrecy properties about message variables. Intuitively the content of a message variable can only be secret, if the thread has received confirmation upon its content. Otherwise the message is instantiated arbitrarily and the adversary might know the content. We formalize this by requiring that the test thread has executed a *confirmation role step* after which the value must be secure. In [19] Meier, Cremers, and Basin introduce notion of an *event* to reason about properties of a trace. They define an event as either a *learn event* m denoting that the adversary learned the message m or a *step event* (i, st) denoting that thread i has executed the role step st. We extend this notion by a long-term key reveal event $\mathsf{LKR}(a)$, denoting that the adversary performed a long-term key reveal on agent a. We define the set

$$\mathit{Event} \overset{\text{def}}{=} \mathit{Msg} \cup (\mathit{TID} \times \mathit{RoleStep}) \cup \{\mathsf{LKR}(a) \mid a \in \mathit{Agent}\}$$

as the set of all events. The already defined function $\mathit{knows}(\mathit{tr})$ projects the trace tr to the set of all learn events. We denote the set of all step events occurring in tr by $\mathit{steps}(\mathit{tr})$.

$$\mathit{steps}(\mathit{tr}) \overset{\text{def}}{=} \{(i, s) \mid \mathsf{St}(i, s) \in \mathit{tr}\}$$

[1] Note that our security protocol model ensures that the role $\mathit{role}_{\mathit{th}}(i)$ of a thread i never changes. The reason is that every transition rule modifying the program counter of a thread removes a role step only from the head of todo and then appends this step to end of done. This ensures that $\mathit{done} \mathbin{\hat{}} \mathit{todo}$ remains the same.

We formalize a secrecy predicate for the message pattern v, the test role R, the confirmation step st, and the state (tr, th, σ) as

$$\forall i \in \mathit{TID}.\ \mathit{role}_{\mathit{th}}(i) = R \wedge (i, st) \in \mathit{steps}(tr) \Rightarrow \mathit{inst}_{\sigma,i}(v) \notin \mathit{knows}(tr)\ .$$

In this formalization, the adversary is not restricted, i.e. he can perform long-term key reveals on all agents and short-term data reveals on all threads. However, this makes him fundamentally too strong. There are three reasons for this. First, if the adversary performs a long-term key reveal on an intended partner agent, he can impersonate this partner agent. Thereby, he can learn a message whenever the impersonated agent receives the message meant to be secret. Second, if the message meant to be secret is noted as state or session-key, the adversary can learn the according message by performing such a short-term reveal on the test thread. Third, instead of a session-key or state reveal on the test thread, the adversary can also perform such a short-term data reveal on an intended partner thread that notes a message pattern that gets instantiated to the message to be secret.

We do not consider any of these attack legitimate. Therefore, we adopt our formalization of secrecy properties such that they do not consider states with these attacks.

In this section, we first define a general reveal type, so we do not have to distinguish between the type of the short-term data reveal and the type of the long-term key reveal anymore. Then we introduce the notion of an adversary capability, which allows the adversary to perform certain reveals. Afterwards, we define a function that allows us to check, if the adversary has obeyed his restrictions in a state. Subsequently we introduce long-term key reveal capabilities. We allow as many long-term keys as possible and still satisfy the principle form the first reason. Afterwards, we introduce a concept to distinguish intended and unintended partner threads. We use this concept to formalize our short-term data reveal adversary capabilities. They allow as many reveals as possible by enforcing the principle from the second and third reason. From this we observe that we can merge the second and third reason by introducing the concept of trust. Finally, we demonstrate how we create partnering functions.

3.1.1. Reveals

We introduce the type *reveal* to generalize the leakage of information by reveals to the adversary. A reveal is either a short-term data reveal $\mathsf{STDR}(nt, i)$ of a note type nt on a thread i or a long-term key reveal $\mathsf{LKR}(a)$ on an agent a. We define the set

$$\mathit{Reveal} \overset{\text{def}}{=} \{\mathsf{STDR}(nt, i) \mid nt \in \mathit{NoteType}, i \in \mathit{TID}\} \cup \{\mathsf{LKR}(a) \mid a \in \mathit{Agent}\}$$

as set of all reveals. The projection of a trace tr to the set of all reveals occurring in tr is defined as

$$\begin{aligned}\mathit{reveals}(tr) \overset{\text{def}}{=} & \{\mathsf{LKR}(a) \mid \mathsf{LKR}(a) \in tr\} \cup \\ & \{\mathsf{STDR}(nt, i) \mid \exists l\, pt.\ \mathsf{St}(i, \mathsf{Note}_l(nt, pt)) \in tr\}\ .\end{aligned}$$

3.1.2. Adversary Capabilities

An *adversary capability* allows the adversary to perform a specified set of reveals. Formally, we define an adversary capability as a function from a state to the set of allowed reveals.

$$\mathit{Capability} \overset{\text{def}}{=} \mathit{State} \rightarrow \mathcal{P}(\mathit{Reveal})$$

3.1.3. Adversary Compromise Models

To check whether the adversary has exceeded his power in a state, we use a set of adversary capabilities and verify that all reveals in the trace are justified by at least one capability. We call a set of adversary capabilities an *adversary compromise model.*

For an adversary compromise model *caps* we are interested in all states in which the adversary respects his restrictions. We formalize this by the *adversary compromise model function acm.*[2]

$$\begin{aligned}&acm\colon \mathcal{P}(\mathit{Capability}) \to \mathcal{P}(\mathit{State})\\ &acm(caps) \overset{\text{def}}{=} \{(tr, th, \sigma) \mid \forall r \in reveals(tr).\ \exists c \in caps.\ r \in c(tr, th, \sigma)\}\end{aligned}$$

We say that a state q is in the adversary compromise model *caps*, if $q \in acm(caps)$ holds.

We use the adversary compromise model function as a pre-condition for our secrecy predicates. Our secrecy predicate for the message pattern v, the test role R, the confirmation step st, the adversary compromise model *caps*, and the state (tr, th, σ) have the form

$$\begin{aligned}&\forall i \in TID.\ role_{th}(i) = R \wedge (i, st) \in steps(tr) \wedge (tr, th, \sigma) \in acm(caps)\\ &\qquad \Rightarrow inst_{\sigma,i}(v) \notin knows(tr)\ .\end{aligned}$$

Note that the more capabilities there are in the adversary compromise model *caps*, the stronger the adversary becomes. That means we have the weakest possible adversary by using the empty set as parameter and get a stronger or at least an equally strong adversary with every capability we add. Formally the set $acm(caps) \cap reachable(\mathbf{P})$ increases monotonically with the size of the adversary compromise model *caps.* Therefore, we have to consider more states in our proofs.

3.1.4. Long-Term Key Reveal Capabilities

To make the adversary as strong as possible, we want to allow as many long-term key reveals as possible without making the adversary fundamentally too strong. Hence, we only disallow long-term key reveals on a restricted set of agents, which we call the set of *protected agents.* We use a set of agent variables *vars* to represent the protected agents and the test thread i to access the content of these variables. We formalize this idea in the adversary capability $\mathsf{LKR}_{protected}\colon (TID \times \mathcal{P}(AVar)) \to Capability.$

$$\mathsf{LKR}_{protected}(i, vars) \overset{\text{def}}{=} \lambda(tr, th, \sigma).\ \{\mathsf{LKR}(a) \mid \forall v \in vars.\ inst_{\sigma,i}(v) \neq a\}$$

Properties such as perfect forward secrecy need the event order to distinguish whether a long-term key reveal is allowed or not. To reason about the order of events, Meier, Cremers, and Basin introduce the strict partial order $\prec_{tr} \subseteq Event \times Event$ in [19]. We extend this order by long-term key reveal events.

$$\begin{aligned}x \prec_{tr} y \overset{\text{def}}{=} \exists tr_1\ tr_2.\ tr = tr_1 \mathbin{\hat{}}\ tr_2 \wedge&\\ &x \in (knows(tr_1) \cup steps(tr_1) \cup \{\mathsf{LKR}(a) \mid \mathsf{LKR}(a) \in tr_1\}) \wedge\\ &y \in (knows(tr_2) \cup steps(tr_2) \cup \{\mathsf{LKR}(a) \mid \mathsf{LKR}(a) \in tr_2\})\end{aligned}$$

[2]Cremers and Basin introduce adversary compromise models in [4]. We explain the relation to their adversary compromise model in Section 3.3

To allow all long-term key reveals that happened after the thread i has executed the role step st, we introduce the adversary capability $\mathsf{LKR}_{afterStep}\colon (\mathit{TID} \times \mathit{RoleStep}) \to \mathit{Capability}$.[3]

$$\mathsf{LKR}_{afterStep}(i, st) \overset{\text{def}}{=} \lambda(tr, th, \sigma).\ \{\mathsf{LKR}(a) \mid (i, st) \prec_{tr} \mathsf{LKR}(a)\}$$

Note that the event order allows us to create other long-term key reveal capabilities, e.g. allow long-term key reveals before the test thread has executed its first step or between its first and last step.

Example 3.1: Adversary Compromise Model for the CRN Protocol
In this example, we present a simple adversary compromise model for the **CRN** protocol. For a test thread i, the adversary compromise model allows long-term key reveals on all agents except the one that is represented by the agent variable s, i.e. the agent that runs the server. We formalize this by the adversary compromise model

$$\{\mathsf{LKR}_{protected}(i, \{s\})\}\ .$$

3.1.5. Partnering

State and session-key reveals on intended communication partners are one of the reasons the adversary is fundamentally too strong. To allow these reveals on non-intended partners, we must distinguish between intended and unintended communication partners first. We use the concept of partnering for this purpose. There are many definitions of partnering in the literature. We present a *partnering function* as our solution, which is very flexible and can model all partnering definitions known to us.

A *partnering relation* is a relation over thread identifiers. A *partnering function* is a mapping from a state to a partnering relation.

$$\mathit{Partnering} \overset{\text{def}}{=} \mathit{State} \to \mathcal{P}(\mathit{TID} \times \mathit{TID})$$

For a partnering function p, $(i, j) \in p(tr, th, \sigma)$ denotes that the thread j is a partner of the thread i in the state (tr, th, σ).

3.1.6. Short-Term Data Reveal Capabilities

We want to allow as many state reveals as possible without making the adversary fundamentally too strong. We achieve this by allowing state reveals on all threads except the test thread i and partners of the test thread in a partnering function p. We formalize this in the adversary capability $\mathsf{StateR}\colon (\mathit{TID} \times \mathit{Partnering}) \to \mathit{Capability}$.

$$\mathsf{StateR}(i, p) \overset{\text{def}}{=} \lambda(tr, th, \sigma).\ \{\mathsf{STDR}(\mathsf{State}, j) \mid (i, j) \notin p(tr, th, \sigma) \wedge j \neq i\}$$

Session-key reveals should only be allowed on threads that are not the test thread i nor a partner of the test thread in the partnering function p. Thereby we allow as many session-key reveals as possible while making the adversary not fundamentally too strong. We formalize this in the adversary capability $\mathsf{SessKeyR}$.[4]

$$\mathsf{SessKeyR}(i, p) \overset{\text{def}}{=} \lambda(tr, th, \sigma).\ \{\mathsf{STDR}(\mathsf{SessKey}, j) \mid (i, j) \notin p(tr, th, \sigma) \wedge j \neq i\}$$

[3] Note that our security protocol model ensures that the long-term keys of every agent are revealed at most once. Thus, a long-term key reveal on an agent after the step st cannot justify an earlier one.

[4] We conclude that $(\mathsf{SessKey}, j) \in \mathsf{SessKeyR}(i, p)(tr, th, \sigma) \Leftrightarrow (\mathsf{State}, j) \in \mathsf{StateR}(i, p)(tr, th, \sigma)$ holds since both adversary capabilities have the same conditions.

3.1.7. Trust Functions

We observe that in the *SessKeyR* and *StateR* adversary capabilities, the adversary can reveal the corresponding short-term data of a thread whenever the thread is not the test thread or a partner of him. For reflexive partnering relations, we do not have to distinguish between the test thread and his partners anymore. However, partnering relations are usually not reflexive.

We can say that the test thread trusts these threads to not get shot-term data revealed by the adversary because then the adversary obeys the restrictions of the adversary compromise model. Therefore, we introduce the concept of a *trust relation*. Let p be a partnering function. We define the according *trust function*

$$t(tr, th, \sigma) \overset{\text{def}}{=} p(tr, th, \sigma) \cup Id$$

such that a thread i trusts a thread j if $(i, j) \in t(tr, th, \sigma)$, i.e. the thread j is either a partner of the thread i or $i = j$. This gives us the advantage of specifying the partnering function independently of the trust function and later extend the partnering function to a trust function. Thus, our next concern is how partnering functions are specified.

3.1.8. Specification of Partnering Functions

Intuitively a thread j is an intended communication partner of a thread i, if they communicate to each other during the protocol run. Thereby they agree on sent and received messages and following from this on the content of variables as soon as they have exchanged them. We call the role step after which all variables of a message are exchanged the *guard step*. In our security protocol model, threads cannot receive values unknown to them, because we pre-instantiate all variables in the initial state. Therefore, a thread already knows what a partner thread must send or receive such that they still agree on these variables and are still partners. Hence guard steps are always steps of the partner j.

The roles of a protocol are specified such that they can communicate to each other. Therefore, we need to ensure that two threads are only partners, if they instantiate roles intended to communicate with each other.

We formalize these two intuitions by the *partnering function generator* function

$$mkPartnering\colon Role \times Role \times \mathcal{P}(Pat \times Pat \times RoleStep) \to Partnering\ .$$

It takes two roles and a set of conditions as parameters and returns a partnering function. Every condition is a triple consisting of two message patterns and a guard role step. A condition holds, if a thread j has not executed the guard role step or the instantiation of the first message pattern for a thread i is equal to the instantiation of the second message pattern for the thread j. The thread j is a partner of the thread i, if all conditions hold. Moreover, the thread i must instantiate the role from the first parameter and the thread j must instantiate the role from the second parameter.

$$\begin{aligned} &mkPartnering(R, R', conds) = \lambda(tr, th, \sigma). \\ &\quad \{(i, j) \mid role_{th}(i) = R \wedge role_{th}(j) = R' \wedge \\ &\qquad \forall (pt, pt', st) \in conds.\ (j, st) \in steps(tr) \Rightarrow inst_{\sigma,i}(pt) = inst_{\sigma,j}(pt')\} \end{aligned}$$

A partnering function should capture all directions, i.e. not only from a role R to a role R' but also vice-versa. Our partnering functions generated from *mkPartnering* are unidirectional because they only capture the partners of one role. For including the other direction, we create also a partnering function for this direction using *mkPartnering*. Then we combine the resulting partnering relations for a state by the set union operator.

An important point about partnering functions is that the stronger they are, i.e. the smaller the resulting partnering relations is, the stronger the adversary gets. This is a consequence from allowing the adversary only short-term data reveals on threads that are not not trusted. Another important point is to consider the agent names in the conditions. Agreeing agent names ensures that two partners are at least executed by the corresponding agent. We can include both points easily into our partnering-function generator by using additional conditions.

In general, we prefer to have only variables and fresh values as message patterns in the conditions. On the one hand this ensures that messages that only consist of these elements are equal. On the other hand the conditions are clearer that with long expressions.

Example 3.2: CRN-Trust Function

In this example, we illustrate how we define a partnering function and the according trust function for the **CRN** protocol using *mkPartnering*. Intuitively, in the **CRN** protocol the roles C and S meant to be communicating with each other. Therefore, a thread j is a partner of a thread i, if either i runs the role C and j runs the role S or i runs the role S and j runs the role C. Moreover, they need to agree on the server agent name and the session-key after thread j has executed its first step.

We formalize this intuition by creating two partnering functions with *mkPartnering*. One is in the direction from the role C to the role S and the other is in the direction from the role S to the role C. Then we combine them with the union operator. Afterwards, we extend this combination by the identity relation *Id* to define a trust function.

$$\begin{aligned} CRN_{trusted} \overset{\text{def}}{=} \lambda(tr, th, \sigma).\ & Id \cup \\ & mkPartnering(C, S, \{(k, v, S_1), (s, s, S_1)\})(tr, th, \sigma) \cup \\ & mkPartnering(S, C, \{(v, k, C_1), (s, s, C_1)\})(tr, th, \sigma) \end{aligned}$$

Note that although the parameters of the two *mkPartnering* function seem almost symmetric, there are protocols that can have a stronger partnering function in one direction than in another.

Example 3.3: Adversary Compromise Model for the CRN-Protocol

In Example 3.1 we give a simple adversary compromise model that allows long-term key reveals on all agents except the one the agent variable s is instantiated to. In this example we extend this adversary compromise model by also allowing session-key reveals on threads not trusted by the test thread in the trust function $CRN_{trusted}$. Formally we provide a function BKE: $TID \to \mathcal{P}(State)$ that given the test thread i returns all states that satisfy our adversary compromise model.

$$BKE \overset{\text{def}}{=} \lambda i.\ acm(\{\mathsf{LKR}_{protected}(i, \{s\}), \mathsf{SessKeyR}(i, CRN_{trusted})\})$$

Example 3.4: CRN Secrecy Property

We formulate a secrecy predicate for the fresh value k of the role C from the protocol **CRN**. Intuitively the adversary must not know the fresh message $k\sharp i$ for any thread i instantiating the role C in a state in the adversary compromise model states $BKE(i)$. We formalize this in the predicate $\phi_{sec}^{\mathbf{CRN}}$.

$$\begin{aligned} \phi_{sec}^{\mathbf{CRN}}(tr, th, \sigma) \overset{\text{def}}{=} & \forall i \in TID.\ role_{th}(i) = C \wedge (tr, th, \sigma) \in BKE(i) \\ & \Rightarrow k\sharp i \notin knows(tr) \end{aligned}$$

Note that the secrecy of the fresh message $k\sharp i$ is not dependent on the execution of some confirmation role step. This is a consequence from the fact that this value is fixed and therefore needs no confirmation.

Discussion of *mkPartnering*

The partnering functions generated by *mkPartnering* have some interesting properties we point out in this subsection. Therefore, let $p = mkPartnering(R, R', conds)$ be a partnering function.

The first property is that all conditions *conds* are irrelevant in the initial state $(tr, th, \sigma) \in Q_0(\mathbf{P})$, because no thread has executed any step and therefore no guard step was executed. Hence a thread j is a partner of a thread i, if they run the specified role, i.e. $role_{th}(i) = R$ and $role_{th}(j) = R'$.

The second property is that as soon as two threads are not partners any more, they cannot be partners again. This is a consequence from the immutability of our variables. Hence, if two messages do not match in a state, they cannot match at a later state. As a consequence the cardinality of the resulting partnering relation decreases monotonically with the progress of the transition system. Moreover, a thread might have partners that have executed a different number of role steps from their role.

The third property is that we can guarantee that at most one partner has executed a send guard step in which he sends a fresh value, if this fresh value is in the condition of the partnering function. This is a consequence of the uniqueness of fresh values. Hence, only one thread can send this fresh value.

The forth property concerns with role-symmetric protocols. For them we have a reflexive partnering relation, if we use the same role for both parameters.

3.2. Authentication Properties

Authentication properties express that whenever a thread reaches a certain point in executing its role, there is another thread j he is communicating with. There are various authentication properties in the literature such as in [18] by Lowe and [13] by Cremers, which require different insurances of the thread j. These range from the execution of a certain role, to the agreement on certain variables to a specific event order. We call the insurances a property requires the *authentication claim.*

We formalize authentication properties as *authentication predicates* that hold for all reachable states. This allows us to use the same principles as we use for secrecy properties. Therefore, we have the following form for a state (tr, th, σ), a adversary compromise model *caps*, a role R, a role step st, and an authentication claim ϕ.

$$\forall i \in TID.\ role_{th}(i) = R \wedge (i, st) \in steps(tr) \wedge (tr, th, \sigma) \in acm(caps) \Rightarrow \phi$$

Example 3.5: CRN Authentication Property

With this authentication property, we express that in a state of the adversary compromise model states $BKE(i)$ for a client thread i, non-injective synchronization holds. That means whenever a client thread i executes C_2, then there exists a server thread j. They agree on the content of the agent variable s and on the session-key. Moreover, the thread j must have executed its first step after the thread i has executed its first and the thread j must have executed its third step before the thread i executes its second. We formalize this as authentication predicate $\phi^{\mathbf{CRN}}_{auth}$.

$$\begin{aligned}\phi^{\mathbf{CRN}}_{auth}(tr, th, \sigma) \overset{\text{def}}{=} {} & \forall i \in \mathit{TID}.\ \mathit{role}_{th}(i) = R \wedge (i, C_2) \in \mathit{steps}(tr) \wedge (tr, th, \sigma) \in \mathit{BKE}(i) \\ & \Rightarrow (\ \exists j \in \mathit{TID}.\ \mathit{role}_{th}(j) = S \wedge \\ & \qquad \sigma(s, i) = \sigma(s, j) \wedge k\sharp i = \sigma(v, j) \wedge \\ & \qquad (i, C_1) \prec_{tr} (j, S_1) \wedge (j, S_3) \prec_{tr} (i, C_2)\ )\end{aligned}$$

3.3. Relation to Adversary Compromise Rules

Cremers and Basin introduce adversary compromise models in [3, 4]. Besides theoretic insights they also show how they model adversary models from the literature using adversary compromise models. We are relating our adversary capabilities to theirs and add missing ones. This enables us to use their insights and knowledge of adversary modeling for our work.

They represent adversary capabilities as adversary compromise rules of their transition system. These transition rules have premises to ensure that the adversary is not fundamentally too strong. Their adversary compromise model is a subset of adversary compromise rules. Thereby they have a parametric transition system that depends on the selected adversary compromise model.

In the following we relate our adversary capabilities to their adversary compromise rules and introduce missing ones. To emphasis this mapping, we name our capabilities according to the represented adversary compromise rules.

3.3.1. Long-Term Key Reveal Capabilities

In this subsection, we introduce four adversary capabilities to represent the four long-term key reveal adversary compromise rules from [3,4]. We use our adversary capabilities $\mathit{LKR}_{protected}$ and $\mathit{LKR}_{afterStep}$ from the Subsection 3.1.4 to formalize them.

$\boldsymbol{LKR}_{others}$

Typically security properties require protection of the agent that executes the test thread and all agents that the test thread communicates with. These agents correspond to the content of all agent variables of the test role. To compute them we first extract all agent variables from a role step and then extend this function to a role.

$FV(st)$ for a role step st denotes the set of variables in the message pattern of the role step st and is defined as

$$FV(st) \overset{\text{def}}{=} \begin{cases} FV(pt) & \text{if } st = \mathsf{Send}_l(pt) \\ FV(pt) & \text{if } st = \mathsf{Recv}_l(pt) \\ FV(pt) & \text{if } st = \mathsf{Note}_l(nt, pt)\ . \end{cases}$$

We define the function $aVars(st) \overset{\text{def}}{=} FV(st) \cap AVar$ that given a role step st denotes the set of all agent variables in the pattern of role step st. We overload notation of $aVars(R)$ for a role R; we write $aVars(R)$ to denote the set of all agent variables in all role steps of the role R.

$$aVars(R) \overset{\text{def}}{=} \bigcup_{st \in R} aVars(st)$$

We define the adversary capability $\mathsf{LKR}_{others}\colon \mathit{TID} \to \mathit{Capability}$ that protects all agents that are represented by an agent variables of test role.[5]

$$\mathsf{LKR}_{others}(i) \overset{\text{def}}{=} \lambda(tr, th, \sigma).\ \bigcup_{role_{th}(i)=R} \mathsf{LKR}_{protected}(i, aVars(R))(tr, th, \sigma)$$

LKR_{actor}

Some adversary models for authentication properties allow the adversary to reveal the long-term keys of the agent that runs the test thread. We call the agent that runs the test thread the *actor*. The only side-condition for this reveal is that the actor is not allowed to instantiate any other role different from the test role in the protocol run of the test thread.

For this capability, we protect the agents that communicate with the test thread. Those are represented by all agent variables of the test role without the agent variable representing the actor. For a test thread i instantiating the role R and the actor agent variable $actor$ we formalize this by

$$\mathsf{LKR}_{protected}(i, aVars(R) \setminus \{actor\})\ .$$

If the actor needs to be protected, i.e. is not in this set, then he runs a different role than the test role in the protocol run of the test thread. We formalize this in the adversary capability $\mathsf{LKR}_{actor}\colon (\mathit{TID} \times \mathit{AVar}) \to \mathit{Capability}$ for the test thread i and the agent variable $actor$ that represents the actor.

$$\begin{aligned}&\mathsf{LKR}_{actor}(i, actor) \overset{\text{def}}{=} \lambda(tr, th, \sigma).\\ &\quad \{\mathsf{LKR}(inst_{\sigma,i}(actor))\} \cap \bigcup_{role_{th}(i)=R} \mathsf{LKR}_{protected}(i, aVars(R) \setminus \{actor\})(tr, th, \sigma)\end{aligned}$$

LKR_{after}

For perfect forward secrecy, the test thread must be finished with its protocol run before long-term key reveals are allowed. We express this by allowing all long-term key reveals that happened after the last send or receive role step of the test role. We use the last communication role step because note role steps can be skipped on the one hand. On the other hand this view corresponds to the security notion for key-exchange protocols. There reveals are modeled as queries and no marker steps are needed in the roles. Therefore, their last communication step is their last step.

We define the partial-function $lastComStep\colon Role \rightharpoonup RoleStep$ that given a role R denotes the last send or receive step of R.

$$lastComStep(R) \overset{\text{def}}{=} last(\ filter(\ (\lambda st.\ \neg(\exists l\ nt\ pt.\ st = \mathsf{Note}_l(nt, pt))), R\ )\ )$$

We define the adversary capability $\mathsf{LKR}_{after}\colon \mathit{TID} \to \mathit{Capability}$ to express perfect forward secrecy, i.e. allow long-term key reveals on all agents after the thread i has finished communicating.

$$\mathsf{LKR}_{after}(i) \overset{\text{def}}{=} \lambda(tr, th, \sigma).\ \bigcup_{\substack{role_{th}(i)=R,\\ lastComStep(R)=st}} \mathsf{LKR}_{afterStep}(i, st)(tr, th, \sigma)$$

[5]Note that we use the union operator such that the domain of the variable introduced in its condition cannot be $\bot$. This ensures R is not $\bot$.

$\mathit{LKR}_{aftercorrect}$

In certain adversary models of key-exchange protocols, protocols with only two communication steps cannot achieve perfect forward secrecy [5]. However, such two communication step protocols can still satisfy the notion of weak perfect forward secrecy in these models. In weak perfect forward secrecy, long-term key reveals are allowed after the protocol run of the test thread and the protocol runs of all other threads that participate in the protocol run of the test thread, i.e. the protocol session, are finished.

Cremers and Basin [3] formalize weak perfect forward secrecy by requesting that besides the test thread i, also a partner thread of the test thread in the partnering p must be finished.[6] We implement this in the adversary capability $\mathit{LKR}_{aftercorrect} \colon (\mathit{TID} \times \mathit{Partnering}) \to \mathit{Capability}$.

$$\begin{aligned} &\mathit{LKR}_{aftercorrect}(i,p) \overset{\text{def}}{=} \lambda(tr,th,\sigma). \\ &\quad \{\mathsf{LKR}(a) \mid \mathsf{LKR}(a) \in \mathit{LKR}_{after}(i)(tr,th,\sigma) \land \\ &\qquad\qquad \exists (i,j) \in p(tr,th,\sigma).\ \mathsf{LKR}(a) \in \mathit{LKR}_{after}(j)(tr,th,\sigma)\} \end{aligned}$$

Note that only a strong partnering function guarantees us that the protocol session of the test thread is finished. The reason is that the partnering function must ensure that the thread i and the thread j agree on all messages they have exchanged. If we use a weaker partnering function, we cannot guarantee that the protocol session is really finished. However, we allow more long-term key reveals with a weak partnering function. Hence the strength of the partnering function is a trade off between the strength of the adversary and the correct definition of weak perfect forward secrecy.

Intuitively, in multi-party protocols a protocol run is finished, if the intended communication partners of all roles are finished. However, in our definition we request that only one partner has finished. Therefore, this definition must be adopted for the multi-party case.[7]

3.3.2. Short-Term Data Reveal Capabilities

We introduce the capabilities $\mathit{SessKeyR}$ and StateR in Subsection 3.1.6 for allowing the adversary to perform session-key and state reveals. These two rules are inspired by the corresponding adversary compromise rules and match them. Hence, the only missing adversary capability deals with randomly generated numbers.

The capability $\mathit{RandGenR} \colon \mathit{Capability}$ allows the reveal of all randomly generated numbers without any restrictions.

$$\mathit{RandGenR} \overset{\text{def}}{=} \{\mathsf{STDR}(\mathsf{RandGen}, j) \mid j \in \mathit{TID}\}$$

[6]Note that we cannot use a trust function or a reflexive partnering function here. Otherwise the $\mathit{LKR}_{aftercorrect}$ would be equal to LKR_{after}

[7]Note that replacing the existential quantifier by an universal quantifier would not work. The reason is that threads that have not processed any role step and instantiate a role that meant to be communicating with the test role, are also partners of the test thread. If the adversary must wait until they have finished their communication or until they are not partners anymore, reveals would be allowed later. Thereby the adversary is weaker.

4. Constructing Protocol Security Proofs

In this chapter we explain how we create protocol security proofs. We have successfully applied this method to our case studies.

We start by explaining our inference system we use to perform the proofs. Then we explain the actual proof procedure. Afterwards we show how proofs that use our inference system are formalized in Isabelle/HOL. Finally, we present insights we have gained from applying our proof procedure.

4.1. Inference Rules

We extend the inference rules for decryption chain reasoning from Meier, Cremers, and Basin [19] with new inference rules to reason about reveals, the adversary compromise model function *acm*, and the partnering function generator *mkPartnering*. Moreover, we also modify some inference rules of the decryption chain reasoning to reflect our security protocol model extensions.

4.1.1. Core Inference Rules for Decryption Chain Reasoning

In [19] Meier, Cremers, and Basin present the Chain rule together with a set of core inference rules for decryption chain reasoning. The Chain rule lists all possibilities on how the adversary can learn a message. It uses the observation that whenever the adversary can learn a message by decryption, it is sufficient to check chains of decryptions starting from a message sent by participants in the protocol. They use the *chain*-predicate for formalizing these decryption chains. The *chain*-predicate is defined in Figure 4.1.

$$\begin{aligned}
\mathit{chain}_{tr}(E,\, m',\, m) \overset{\text{def}}{=}\; & \\
& (m' = m \wedge \forall e \in E.\; e \prec_{tr} m\;) \vee \\
& (\exists x\, k.\; m' = \{\!|x|\!\}_k \wedge (\forall e \in E.\; e \prec_{tr} \{\!|x|\!\}_k) \wedge \mathit{chain}_{tr}(\{\{\!|x|\!\}_k, k^{-1}\},\, x,\, m)) \vee \\
& (\exists x\, y.\; m' = (x, y) \wedge (\mathit{chain}_{tr}(E,\, x,\, m) \vee \mathit{chain}_{tr}(E,\, y,\, m)))
\end{aligned}$$

Figure 4.1.: Definition of the *chain* predicate using recursion over the message m'.

We adopt these inference rules for our model and describe the new and changed rules. Our inference rules are given in Figure 4.2. They are formally derived from our transition system under $(tr, th, \sigma) \in \mathit{reachable}(\mathbf{P})$. In the following we explain our newly introduced inference rules LKRevealed, NoteRevealed, RoleSend and RoleRecv rules. Moreover, we also explain the changes to the Chain rule.

The rule LKRevealed states that, if a long-term key reveal $\mathsf{LKR}(a)$ on an agent a happened before some other event e, then the long-term keys of the agent a were revealed. The rule

$$\frac{(m_1, m_2) \in \mathit{knows}(\mathit{tr})}{m_1 \in \mathit{knows}(\mathit{tr})}\ \textsc{Kn}_1 \qquad \frac{(m_1, m_2) \in \mathit{knows}(\mathit{tr})}{m_2 \in \mathit{knows}(\mathit{tr})}\ \textsc{Kn}_2$$

$$\frac{(m_1, m_2) \prec_{tr} e}{m_1 \prec_{tr} e}\ \textsc{Ord}_1 \qquad \frac{(m_1, m_2) \prec_{tr} e}{m_2 \prec_{tr} e}\ \textsc{Ord}_2$$

$$\frac{m \prec_{tr} e}{m \in \mathit{knows}(\mathit{tr})}\ \textsc{Known} \qquad \frac{(i, s) \prec_{tr} e}{(i, s) \in \mathit{steps}(\mathit{tr})}\ \textsc{Exec}$$

$$\frac{(i, \mathsf{Note}_l(\mathit{nt}, \mathit{pt})) \prec_{tr} e}{\mathsf{STDR}(\mathit{nt}, i) \in \mathit{reveals}(\mathit{tr})}\ \textsc{NoteRevealed} \qquad \frac{\mathsf{LKR}(a) \prec_{tr} e}{\mathsf{LKR}(a) \in \mathit{reveals}(\mathit{tr})}\ \textsc{LKRevealed}$$

$$\frac{e \prec_{tr} e}{\mathit{false}}\ \textsc{Irr} \qquad \frac{e_1 \prec_{tr} e_2 \qquad e_2 \prec_{tr} e_3}{e_1 \prec_{tr} e_3}\ \textsc{Trans}$$

$$\frac{\mathit{role}_{th}(i) = R \qquad \mathsf{Send}_l(\mathit{pt}) <_R \mathit{st} \qquad (i, \mathit{st}) \in \mathit{steps}(\mathit{tr})}{(i, \mathsf{Send}_l(\mathit{pt})) \prec_{tr} (i, \mathit{st})}\ \textsc{RoleSend}$$

$$\frac{\mathit{role}_{th}(i) = R \qquad \mathsf{Recv}_l(\mathit{pt}) <_R \mathit{st} \qquad (i, \mathit{st}) \in \mathit{steps}(\mathit{tr})}{(i, \mathsf{Recv}_l(\mathit{pt})) \prec_{tr} (i, \mathit{st})}\ \textsc{RoleRecv}$$

$$\frac{(i, \mathsf{Recv}_l(\mathit{pt})) \in \mathit{steps}(\mathit{tr})}{\mathit{inst}_{\sigma,i}(\mathit{pt}) \prec_{tr} (i, \mathsf{Recv}_l(\mathit{pt}))}\ \textsc{Input}$$

$$\frac{m \in \mathit{knows}(\mathit{tr})}{\begin{array}{l} (m \in \mathit{IK}_0) \lor \\ (\exists x.\ m = \mathsf{h}(x) \land x \prec_{tr} \mathsf{h}(x)) \lor \\ (\exists x\ k.\ m = \{\!|x|\!\}_k \land x \prec_{tr} \{\!|x|\!\}_k \land k \prec_{tr} \{\!|x|\!\}_k) \lor \\ (\exists x\ y.\ m = (x, y) \land x \prec_{tr} (x, y) \land y \prec_{tr} (x, y)) \lor \\ (\exists R \in \mathbf{P}.\ \exists\, \mathsf{Send}_l(\mathit{pt}) \in R.\ \exists i.\ \mathit{role}_{th}(i) = R \land \\ \qquad \mathit{chain}_{tr}(\{(i, \mathsf{Send}_l(\mathit{pt}))\},\ \mathit{inst}_{\sigma,i}(\mathit{pt}),\ m)\ ) \lor \\ (\exists R \in \mathbf{P}.\ \exists\, \mathsf{Note}_l(\mathit{ty}, \mathit{pt}) \in R.\ \exists i.\ \mathit{role}_{th}(i) = R \land \\ \qquad \mathit{chain}_{tr}(\{(i, \mathsf{Note}_l(\mathit{nt}, \mathit{pt}))\},\ \mathit{inst}_{\sigma,i}(\mathit{pt}),\ m)\ ) \lor \\ (\exists \mathsf{LKR}(a) \in \mathit{tr}.\ m \in \mathit{longTermKeys}(a) \land \mathsf{LKR}(a) \prec_{tr} m) \end{array}}\ \textsc{Chain}$$

Figure 4.2.: Inference rules derived from the transition system under the assumption that $(\mathit{tr}, \mathit{th}, \sigma) \in \mathit{reachable}(\mathbf{P})$.

NoteRevealed states that whenever a note step $\mathsf{Note}_l(nt, pt)$ of a thread i happened before some other event e, then the adversary has performed a nt-reveal on the thread i. These rules follow from the definition of the event order $\prec_{tr}$.

The rules RoleSend and RoleRecv state that, if a thread i that is an instance of the role R and has executed the role step st, then all send and receive steps st' such that $st' <_R st$, have been executed before st. Both rules are sound, because the Send and Recv transition rules are the only transition rules applicable for send and receive role steps and they preserve the role order.

We extend the chain rule Chain with two new cases on how the adversary can learn messages. The first case reflects our Compr transition rule. It expresses that the adversary can learn a message m by revealing a message $inst_{\sigma,i}(pt)$ currently available at a note step $\mathsf{Note}_l(nt, pt)$ of a thread i and learning the message m by zero or more decryptions and projections of $inst_{\sigma,i}(pt)$. That means we also consider decryption chains starting from a note step. The second case reflects our LKR transition rule. It expresses that the adversary can learn a long-term key of an agent a by performing a long-term key reveal $\mathsf{LKR}(a)$ on this agent.

4.1.2. Inference Rules for Reasoning about Adversary Compromise Models and Partnering Functions

To reason about our partnering function generator *mkPartnering* and the adversary compromise model function *acm*, we introduce the two inference rules MkPartI and AllowedReveals. Both are given in Figure 4.3.

The rule MkPartI states that a thread j is a partner of a thread i whenever the thread i instantiates the role R and the thread j instantiates the role R' and they both agree on each condition from the set *conds*. This rule follows from the definition of the partnering function generator *mkPartnering*.

The rule AllowedReveals states that every reveal in a state $(tr, th, \sigma) \in acm(caps)$ of an adversary compromise model *caps* is justified by an adversary capability $cap \in caps$. This inference rule follows from the definition of the function *acm*.

$$\frac{\begin{array}{c} role_{th}(i) = R \qquad\qquad role_{th}(j) = R' \\ \forall (pt, pt', st) \in conds.\ (j, st) \in steps(tr) \Rightarrow inst_{\sigma,i}(pt) = inst_{\sigma,j}(pt') \end{array}}{(i, j) \in mkPartnering(R, R', conds)(tr, th, \sigma)}\ \textsc{MkPartI}$$

$$\frac{(tr, th, \sigma) \in acm(caps) \qquad r \in reveals(tr)}{\exists cap \in caps.\ r \in cap}\ \textsc{AllowedReveals}$$

Figure 4.3.: Inference rules to reason about partnerings and adversary compromise models.

4.2. Protocol Security Proof Procedure

Our security proof procedure has three parts. First, we specify the roles, the protocol, the partnering function, the adversary compromise model, and the security properties as we explain in Chapter 2 and Chapter 3. Second, we perform preparations for the upcoming protocol security proofs. In these preparations, we customize inference rules and prove additional properties to reason more effectively. These activities achieve this by factoring out commonly shared parts of the proofs. Third, we perform the protocol security proofs, i.e. we prove that the security properties hold.

4.2.1. Inference Rule Customizations

To reason more effectively about a protocol, an adversary compromise model, and a trust function we pre-instantiate the CHAIN, ALLOWEDREVEALS, and MKPARTI inference rules for the concrete object. This instantiation is completely mechanical and can be performed automatically by the theorem prover. In the following we explain the instantiation process for each of the rules.

We instantiate the CHAIN rule by instantiating its premise for different outermost message constructors, enumerate all note and send steps in the conclusion, and unfold all occurrences of the *chain*-predicate. Thereby we have a first clue on the message derivation capabilities of the adversary.

Example 4.1: CRN CHAIN-Rule Instantiation

In this example, we illustrate the instantiation of the CHAIN rule for the **CRN** protocol. Those instantiated instances are presented in Figure 4.4.

The rules KCHAIN$_{\mathbf{CRN}}$ and SKCHAIN$_{\mathbf{CRN}}$ show that the adversary can learn long-term keys only after performing a long-term key reveal. The rule ECHAIN$_{\mathbf{CRN}}$ shows that the adversary can learn an encryption by either creating it himself or from the send step of the client C_1. The rule HCHAIN$_{\mathbf{CRN}}$ shows that the adversary can learn hashes only by either creating them himself or from the send step of a server S_3.

The rule NCHAIN$_{\mathbf{CRN}}$ shows that the adversary has three possibilities to learn a fresh message. The first is to decrypt the message that the client sends in C_1 with the long-term private key of the server. The second is to perform a session-key reveal of the session-key available at C_3. The third is that the adversary reveals the unprotected system state of a server thread in S_2.

Note that we need an additional thought for the third possibility because $\sigma(v, i)$ could be an arbitrary message in our untyped model. Thus we cannot unfold the chain predicate further, if we have $chain_{tr}(E,\ \sigma(v,i),\ m)$ for a set of events E and a message m. However, Meier, Cremers, and Basin noticed in [19] that in certain protocols all message variables can store only either a fresh message or a message that the adversary already knows before the client receives a message for the message variable the first time. They call a protocol that satisfies this property *weak-atomic*. The **CRN** protocol is weak-atomic. Therefore we know that $\sigma(v, i)$ is either a fresh message or a message that the adversary knows before (j, S_1). This allows the following proof idea. If $\sigma(v, i)$ stores a fresh value, then we get NCHAIN$_{\mathbf{CRN}}$ as stated. In case the adversary already knows $\sigma(v, i)$ beforehand, we get a contradiction because the adversary would learn $\sigma(v, i)$ before and after the role step S_2.

We pre-instantiate the inference rule ALLOWEDREVEALS for an adversary compromise model to reason about reveals allowed in this adversary compromise model. Thereby we replace the existential quantifier in its conclusion with a conjunction over all capabilities. Then we unfold all the definitions of the adversary capability in the expression and simplify the result.

Example 4.2: Instantiation of the inference rule ALLOWEDREVEALS for *BKE*

We illustrate the instantiation of the inference rule ALLOWEDREVEALS for the adversary compromise model from Example 3.3. The resulting inference rule ALLOWEDREVEALS$_{BKE}$ states that, if a reveal happened in a state of the adversary compromise model states $BKE(i)$ for a thread i, then either it is justified by the LKR_{others} or $\mathit{SessKeyR}$ capability. By unfolding the definition of the LKR_{others} capability, we have that the reveal must be a

$$\frac{\mathsf{sk}_a \in \mathit{knows}(tr)}{\mathsf{LKR}(a) \prec_{tr} \mathsf{sk}_a}\ \text{SKC}\textsc{hain}_{\mathbf{CRN}}$$

$$\frac{\mathsf{k}_{a,b} \in \mathit{knows}(tr)}{\mathsf{LKR}(a) \prec_{tr} \mathsf{k}_{a,b} \vee \mathsf{LKR}(b) \prec_{tr} \mathsf{k}_{b,a}}\ \text{KC}\textsc{hain}_{\mathbf{CRN}}$$

$$\frac{\{\!|m|\!\}_x \in \mathit{knows}(tr)}{\begin{array}{l}(m \prec_{tr} \{\!|m|\!\}_x \wedge x \prec_{tr} \{\!|m|\!\}_x) \vee \\ (\exists j.\ \mathit{role}_{th}(j) = C \wedge (j, C_1) \prec_{tr} \{\!|k\sharp j|\!\}_{\mathsf{pk}_{\sigma(s,j)}} = \{\!|m|\!\}_x)\end{array}}\ \text{EC}\textsc{hain}_{\mathbf{CRN}}$$

$$\frac{\mathsf{h}(x) \in \mathit{knows}(tr)}{\begin{array}{l}(x \prec_{tr} \mathsf{h}(x)) \vee \\ (\exists j.\ \mathit{role}_{th}(j) = S \wedge (j, S_3) \prec_{tr} \mathsf{h}(\sigma(v,j)) = \mathsf{h}(x))\end{array}}\ \text{HC}\textsc{hain}_{\mathbf{CRN}}$$

$$\frac{n\sharp i \in \mathit{knows}(tr)}{\begin{array}{r}(\ \mathit{role}_{th}(i) = C \wedge (\ i, C_1) \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,i)}} \prec_{tr} k\sharp i \\ \wedge\, \mathsf{sk}_{\sigma(s,i)} \prec_{tr} k\sharp i \wedge k\sharp i = n\sharp i\ ) \vee \\ (\ \mathit{role}_{th}(i) = C \wedge (i, C_3) \prec_{tr} k\sharp i \wedge k\sharp i = n\sharp i\ ) \vee \\ (\ \exists j.\ \mathit{role}_{th}(j) = S \wedge (j, S_2) \prec_{tr} k\sharp i \wedge \sigma(v, i) = n\sharp i\ )\end{array}}\ \text{NC}\textsc{hain}_{\mathbf{CRN}}$$

Figure 4.4.: Instantiated instances of the Chain rule for the **CRN** protocol.

long-term key reveal on an agent a and the content of the agent variable s of the thread i must not be equal to the agent a. From unfolding the capability *SessKeyR*, we have that the reveal must be a session-key reveal on a thread j that is not trusted by the thread i with respect to $\mathit{CRN}_{trusted}$.

$$\frac{(tr, th, \sigma) \in \mathit{BKE}(i) \qquad r \in \mathit{reveals}(tr)}{\begin{array}{l}(\ \exists a.\ r = \mathsf{LKR}(a) \wedge \sigma(s, i) \neq a\ ) \vee \\ (\ \exists j.\ r = \mathsf{STDR}(\mathsf{SessKey}, j) \wedge (i, j) \notin \mathit{CRN}_{trusted}(tr, th, \sigma) \wedge j \neq i\ )\end{array}}\ \text{A}\textsc{llowed}\text{R}\textsc{eveals}_{\mathit{BKE}}$$

Figure 4.5.: Instantiated inference rule AllowedReveals for *BKE*

Our trust functions are a combination of partnering functions and the identify relation. Our partnering functions are combinations of partnering functions generated from our partnering function generator *mkPartnering*. To reason about such structures we cannot just instantiate the MkPartI inference rules, but have to create a simple inference rule first and then apply MkPartI to all *mkPartnering* instances in the resulting expression.

We use $(i, j) \in t(tr, th, \sigma)$ as premise and conclusion for the inference rule. Then we unfold the definition of the trust function in the premise and simply the resulting expression such that the set union operators are replaced by a logical disjunction. Then we replace all occurrences of *mkPartnering* with the premise of the inference rule MkPartI.

Example 4.3: $CRN_{trusted}$ Specific Inference Rule
In this example, we explain the inference rule $\textsc{Trusted}_{\mathbf{CRN}}$ for the partnering definition $CRN_{trusted}$. The inference rule $\textsc{Trusted}_{\mathbf{CRN}}$ states that whenever a thread j is a partner of a thread i, then it is either because of the partnering function from the role C to the role S, the partnering function from the role S to the role C, or $i = j$ holds. In the first direction, i instantiate the role C, j instantiate the role S, and if j has executed role step S_1, then they agree on the session-key and the agent name of the server. For the other direction, i instantiates the role S, j instantiates the role C and they have the same session-key and server agent name after i executes C_1. We present the resulting inference rule in Figure 4.6.

$$\frac{\begin{array}{l} (\ i = j\ ) \vee \\ (\ role_{th}(i) = C \wedge role_{th}(j) = S \wedge \\ \quad (j, S_1) \in steps(tr) \Rightarrow k\sharp i = \sigma(v, j) \wedge \\ \quad (j, S_1) \in steps(tr) \Rightarrow \sigma(s, i) = \sigma(s, j)\ ) \vee \\ (\ role_{th}(i) = S \wedge role_{th}(j) = C \wedge \\ \quad (j, C_1) \in steps(tr) \Rightarrow \sigma(v, i) = k\sharp j \wedge \\ \quad (j, C_1) \in steps(tr) \Rightarrow \sigma(s, i) = \sigma(s, j)\ ) \end{array}}{(i, j) \in CRN_{trusted}(tr, th, \sigma)} \ \textsc{Trusted}_{\mathbf{CRN}}$$

Figure 4.6.: Inference rule to reason about the trust function $CRN_{trusted}$.

4.2.2. Origin Properties

In a correct protocol, the adversary can learn messages meant to be secret only by exceeding the capabilities that the adversary compromise model imposes on him. Hence, we can split the task of proving the secrecy of such a message into two tasks. First, we determine which reveals the adversary has to perform for learning the message without the restrictions of an adversary compromise model. Second, we verify that these reveals are not allowed in the adversary compromise model. This method provides the advantage that we can use the results from the first task for different second tasks, i.e. use it for different adversary compromise models. Moreover, we can see which guarantees a protocol really provides without the interference of the adversary compromise model.

For the first task, we introduce the notion of an *origin property* that lists the reveals that the adversary must have performed to learn a message. We call these reveals the *reveal-conditions*. We formalize origin properties as origin predicates that must hold for all reachable states. Formally, an origin predicate has the form

$$\begin{aligned} \forall i \in TID.\ & role_{th}(i) = R \wedge (i, st) \in steps(tr) \wedge inst_{\sigma,i}(v) \in knows(tr) \\ & \Rightarrow cond1 \vee \ldots \vee condN \end{aligned}$$

for reveal-conditions $cond1, \ldots, condN$, the state (tr, th, σ), the role R, the message pattern v, and the confirmation role step st. It denotes that whenever a thread i instantiates the role R and has executed the role step st, the adversary must have learned v by one of the reveal-conditions $cond1, \ldots, condsN$.

We use a standard form on how we write reveal-conditions. This standard form corresponds to the definitions of the adversary capabilities and therefore makes the second task easier.

Given the test thread i, an agent a, a thread j, a note type nt, a partnering function p, and a trust function t we have listed the reveal-condition forms in Figure 4.7.

$$\begin{array}{rr}
\mathsf{LKR}(a) \in \mathit{reveals}(\mathit{tr}) & \text{(LTKR.Agent)} \\
\mathsf{LKR}(a) \prec_{tr} (i, \mathit{st}) & \text{(LTKR.Before)} \\
\mathsf{STDR}(\mathit{nt}, i) \in \mathit{reveals}(\mathit{tr}) & \text{(STDR.Test)} \\
\exists j \in \mathit{TID}.\ (i,j) \in p(\mathit{tr}, \mathit{th}, \sigma) \wedge \mathsf{STDR}(\mathit{nt}, i) \in \mathit{reveals}(\mathit{tr}) & \text{(STDR.Part)} \\
\exists j \in \mathit{TID}.\ (i,j) \in t(\mathit{tr}, \mathit{th}, \sigma) \wedge \mathsf{STDR}(\mathit{nt}, i) \in \mathit{reveals}(\mathit{tr}) & \text{(STDR.Trust)} \\
\exists j \in \mathit{TID}.\ \mathsf{STDR}(\mathit{nt}, i) \in \mathit{reveals}(\mathit{tr}) & \text{(STDR.UnSpec)}
\end{array}$$

Figure 4.7.: Standard form of reveal-conditions.

(LTKR.Agent) denotes that the adversary has performed a long-term key reveal on an agent a. With (LTKR.Before) we denote that the adversary has performed a long-term key reveal on an agent a before the thread i has executed the role step st. A short-term data reveal of type nt on the test thread is denoted by (STDR.Test). (STDR.Part) denotes a short-term data reveal of type nt on a partner of the test thread i in the partnering function p. (STDR.Trust) denotes the short-term data reveal of type nt on a thread that is trusted by the test thread i in the trust function t. Note that due to the use of a trust function instead of a partnering function, we can merge the two reveal-conditions of the form (STDR.Test) and (STDR.Part) into one of the form (STDR.Trust). A short-term data reveal of type nt on an unspecified thread j is denoted by (STDR.UnSpec).

We create an origin property for each message meant to be secret in the protocol, i.e. one origin property for each secrecy property. We do not create origin properties for secret long-term keys because we already have an origin property for them by the for a protocol instantiated inference rule Chain. For further use we are interested in the strongest possible origin property for a message, i.e. the one with the least number and strongest reveal-conditions.

Example 4.4: Origin Property for the Session-Key k in the CRN Protocol

In this example we illustrate by which means the adversary can learn the session-key k of a client thread i, if he does not obey any restrictions. Intuitively the adversary can learn k either by decrypting the message sent by the client in C_1, a session-key reveal in C_3, or a state-reveal in S_2. For decrypting the message sent in C_1 the adversary has to perform a long-term key reveal on the agent $\sigma(s, i)$. We formalize this intuition in the origin predicate $\phi^{\mathbf{CRN}}_{origin}$.

$$\begin{aligned}
&\phi^{\mathbf{CRN}}_{origin}(\mathit{tr}, \mathit{th}, \sigma) \overset{\text{def}}{=} \\
&\quad \forall i \in \mathit{TID}.\ \mathit{role}_{\mathit{th}}(i) = C \wedge k\sharp i \in \mathit{knows}(\mathit{tr}) \Rightarrow \\
&\qquad\qquad \mathsf{LKR}(\sigma(s, i)) \in \mathit{reveals}(\mathit{tr}) \vee \\
&\qquad\qquad (\ \exists j \in \mathit{TID}.\ \mathsf{STDR}(\mathsf{State}, j) \in \mathit{reveals}(\mathit{tr})\ ) \vee \\
&\qquad\qquad (\ \exists j \in \mathit{TID}.\ (i,j) \in \mathit{CRN}_{trusted}(\mathit{tr}, \mathit{th}, \sigma) \wedge \mathsf{STDR}(\mathsf{SessKey}, j) \in \mathit{reveals}(\mathit{tr})\ )
\end{aligned}$$

Example 4.5: CRN Origin Proof

In this example, we show that the origin property $\phi^{\mathbf{CRN}}_{origin}$ holds for all reachable states.

$$\forall (\mathit{tr}, \mathit{th}, \sigma) \in \mathit{reachable}(\mathbf{CRN}).\ \phi^{\mathbf{CRN}}_{origin}(\mathit{tr}, \mathit{th}, \sigma)$$

Proof. We must show that for every reachable state $(tr, th, \sigma) \in \mathit{reachable}(\mathbf{CRN})$ and all threads i instantiating the role C such that $k\sharp i \in \mathit{knows}(tr)$ holds, implies

$$\begin{aligned}&\mathsf{LKR}(\sigma(s,i)) \in \mathit{reveals}(tr) \lor \\ &(\ \exists j \in \mathit{TID}.\ \mathsf{STDR}(\mathsf{State}, j) \in \mathit{reveals}(tr)\ ) \lor \\ &(\ \exists j \in \mathit{TID}.\ (i,j) \in \mathit{CRN}_{trusted}(tr, th, \sigma) \land \mathsf{STDR}(\mathsf{SessKey}, j) \in \mathit{reveals}(tr)\ )\ .\end{aligned}$$

We apply the rule NChain$_{\mathbf{CRN}}$ to $k\sharp i \in \mathit{knows}(tr)$, which yields the following conclusion, whose disjuncts we have named.

$$(\ \mathit{role}_{th}(i) = C \land (\ i, C_1) \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,i)}} \prec_{tr} k\sharp i \land \mathsf{sk}_{\sigma(s,i)} \prec_{tr} k\sharp i \land k\sharp i = k\sharp i\ ) \lor \quad \text{(CRk1)}$$
$$(\ \mathit{role}_{th}(i) = C \land (i, C_3) \prec_{tr} k\sharp i \land k\sharp i = k\sharp i\ ) \lor \quad \text{(CRk2)}$$
$$(\ \exists j.\ \mathit{role}_{th}(j) = S \land (j, S_2) \prec_{tr} k\sharp i \land \sigma(v, i) = k\sharp i\ ) \quad \text{(CRk3)}$$

Case (CRk1) is where the adversary learns $k\sharp i$ by decrypting the first message of i. Therefore, he must know $\mathsf{sk}_{\sigma(s,i)}$. He can only learn $\mathsf{sk}_{\sigma(s,i)}$ from a long-term key reveal on the agent $\sigma(s,i)$. We prove this by applying rule Known to $\mathsf{sk}_{\sigma(s,i)} \prec_{tr} k\sharp i$ which yields $\mathsf{sk}_{\sigma(s,i)} \in \mathit{knows}(tr)$. Using rule SKChain$_{\mathbf{CRN}}$, we conclude $\mathsf{LKR}(\sigma(s,i)) \prec_{tr} \mathsf{sk}_{\sigma(s,i)}$. Thus we have $\mathsf{LKR}(\sigma(s,i)) \in \mathit{reveals}(tr)$ by rule LKRevealed.

Case (CRk2) is where the adversary learns $k\sharp i$ from a session-key reveal on the test thread i. We get $\mathsf{STDR}(\mathsf{SessKey}, i) \in \mathit{reveals}(tr)$ from $(C_3, i) \prec_{tr} k\sharp i$ using NoteRevealed. From $i = i$ we have $(i,i) \in \mathit{CRN}_{trusted}(tr, th, \sigma)$ using Trusted$_{\mathbf{CRN}}$.

Case (CRk3) implies that there is a server thread j, $\mathit{role}_{th}(j) = S$, and the adversary learns $\sigma(v,j) = k\sharp i$ from revealing the state at S_2. Let j be an arbitrary thread such that $\mathit{role}_{th}(j) = S \land (j, S_2) \prec_{tr} k\sharp i \land \sigma(v,i) = k\sharp i$ holds. We have $\mathsf{STDR}(\mathsf{State}, j) \in \mathit{reveals}(tr)$ from applying the rule NoteRevealed to $(j, S_2) \prec_{tr} k\sharp i$.

This concludes our proof.

■

4.2.3. Secrecy Proofs

With the help of origin properties, we can reduce secrecy property proofs to proofs that no reveal-condition is allowed in the adversary compromise model. Therefore, we create a case for every reveal-condition. Then we verify that the reveal of the reveal-condition is not allowed by applying the inference rule AllowedReveals, which we have instantiated for the adversary compromise model of the security property. If the origin property is formulated strong enough, this leads to contradictions in all cases and therefore the secrecy property holds.

An origin property is strong enough for an adversary compromise model of a secrecy property, if all reveal conditions of the origin property are disallowed in the adversary compromise model. A reveal-condition is disallowed in an adversary compromise model, if the reveal of the reveal-condition is not allowed in every adversary capability of the adversary compromise model. We can also take the perspective from the origin property. Then we have that, there is an adversary compromise model for each origin property of a message, such that the origin property is strong enough for the secrecy proof of the message. To determine this adversary compromise model we compute an adversary capability that does not contain this reveal, i.e. this adversary capability is the union of all adversary capabilities of the adversary compromise model, and create a singleton set with this adversary capability. We use the union of our adversary capabilities to get a set of reveals for each reveal-condition that does not include the reveal. Then we use the intersection of these sets to determine this adversary capability. If the union of the adversary capabilities of the adversary compromise model we use for a secrecy property proof is a subset of this computed adversary capability, then we can prove the secrecy property using the origin property. We use this technique informally to verify that we specify origin properties strong enough for the later secrecy proofs.

To reason efficiently whether the reveal of a reveal-condition is allowed in an adversary capability, we introduce in Subsection 4.2.2 a standard form for reveal-conditions. This form allows a mapping of reveal conditions to adversary capabilities in which the reveal is not allowed. In Table 4.1, we list this mapping. In this table, a reveal condition that has the form stated in the first column is disallowed by the adversary capability in the second column under the condition of the third column. Note that we restrict the table to adversary capabilities that have the same constructor, i.e. LKR or STDR, and the same note type. The reason is that a capability that allows only long-term key reveals cannot allow a short-term data reveal and vice versa. Moreover, a reveal condition of a certain note type is not allowed by all short-term data reveal capabilities of other types.

Reveal Condition	Capability	Additional Conditions
(LTKR.Agent)	$\mathit{LKR}_{others}(i)$	$\exists R.\ role_{th}(i) = R$ and $\exists v \in aVars(R).\ inst_{\sigma,i}(v) = a$
	$\mathit{LKR}_{protected}(i, vars)$	$\exists v \in vars.\ \sigma(i, v) = a$
(LTKR.Before)	$\mathit{LKR}_{after}(i)$	-
	$\mathit{LKR}_{aftercorrect}(i, p(tr, th, \sigma))$	-
	$\mathit{LKR}_{afterStep}(i, st)$	
	$\mathit{LKR}_{others}(i)$	$\exists R.\ role_{th}(i) = R$ and $\exists v \in aVars(R).\ inst_{\sigma,i}(v) = a$
	$\mathit{LKR}_{protected}(i, vars)$	$\exists v \in vars.\ \sigma(i, v) = a$
(STDR.Test)	$\mathit{SessKeyR}(i, p(tr, th, \sigma))$	$nt = \mathsf{SessKey}$
	$\mathit{StateR}(i, p(tr, th, \sigma))$	$nt = \mathsf{State}$
(STDR.Part)	$\mathit{SessKeyR}(i, p(tr, th, \sigma))$	$nt = \mathsf{SessKey}$
	$\mathit{StateR}(i, p(tr, th, \sigma))$	$nt = \mathsf{State}$
(STDR.Trust)	$\mathit{SessKeyR}(i, p(tr, th, \sigma))$	$nt = \mathsf{SessKey}$
	$\mathit{StateR}(i, p(tr, th, \sigma))$	$nt = \mathsf{State}$
(STDR.UnSpec)	-	

Table 4.1.: Reveal conditions that are not allowed in an adversary capability.

A long-term key reveal on an agent a in the form (LTKR.Agent) for the test thread i is not allowed by the reveal condition LKR_{others}, if an agent variable of the test-role represents the agent a. Moreover the $\mathit{LKR}_{protected}$ capability does also not allow it, if the agent a is protected by one of the agent variables $vars$, i.e. the contents of one of the variable in $vars$ is equal to a.

Reveal conditions of the form (LTKR.Before) for the thread i, an agent a, and a role step st are not allowed by the LKR_{after}, $\mathit{LKR}_{aftercorrect}$, and the $\mathit{LKR}_{afterStep}$ adversary capabilities. Moreover also the LKR_{others} and $\mathit{LKR}_{protected}$ adversary capabilities do not allow this reveal, if the agent a is represented by an agent variable in the test role or is protected by an agent variable in $vars$.

The reveal-conditions forms (STDR.Test), (STDR.Part), and (STDR.Trust) for the session-key note type are not allowed by the $\mathit{SessKeyR}$ adversary capability. Furthermore, these reveal condition forms for a state note type are not allowed by StateR adversary capability. Note that the short-term key reveal-condition of the form (STDR.UnSpec) might allow short-term data reveal of the according note type. Therefore, this form should be only used, if we do not allow these types of reveal later anyway.

We have not yet mentioned the adversary capabilities $\mathit{RandGenR}$ and LKR_{actor}. The reason why we have not mentioned the $\mathit{RandGenR}$ capability is that it allows any random generator reveals. Hence we can only use an origin property for secrecy property proofs with an adversary compromise models that contains the $\mathit{RandGenR}$ capability, if there is no reveal-condition for a random generator reveal in the origin properties. The LKR_{actor} capability can be used only in authentication proofs and not for secrecy proofs. The reason is that whenever the adversary knows the secret long-term keys of the agent running the test thread, he can impersonate him.

Example 4.6: CRN Adversary Compromise Model for $\phi^{\mathbf{CRN}}_{origin}$

In this example, we illustrate how we determine the adversary model for which the origin property $\phi^{\mathbf{CRN}}_{origin}$ is strong enough such that it can be used to prove a secrecy property for this adversary compromise model. In the $\phi^{\mathbf{CRN}}_{origin}$ we have the reveal conditions

$$\begin{aligned}
&\mathsf{LKR}(\sigma(s,i)) \in \mathit{reveals}(tr)\ ,\\
&(\ \exists j \in \mathit{TID}.\ \mathsf{STDR}(\mathsf{State}, j) \in \mathit{reveals}(tr)\ )\ ,\ \text{and}\\
&(\ \exists j \in \mathit{TID}.\ (i,j) \in \mathit{CRN}_{trusted}(tr, th, \sigma) \wedge \mathsf{STDR}(\mathsf{SessKey}, j) \in \mathit{reveals}(tr)\ )
\end{aligned}$$

for the test thread i. The long-term key reveal condition is not allowed by the $\mathit{LKR}_{protected}$ adversary capability, if the agent variable s is protected. Moreover the LKR_{others} adversary capability also does not allow this reveal because the agent variable s is in the role C. Furthermore all short-term data reveal capabilities do not allow this reveal. We define the following set of reveals for a state that does not contain this long-term key reveal.

$$\begin{aligned}
&\mathit{LKR}_{protected}(i, \{s\})(tr, th, \sigma) \cup \mathit{LKR}_{others}(i)(tr, th, \sigma) \cup\\
&\{\mathsf{STDR}(nt, j) \mid nt \in \mathit{NoteType}, j \in \mathit{TID}\}
\end{aligned}$$

The state reveal of the second reveal condition has the form (STDR.UnSpec). Therefore the StateR adversary capability might allow this reveal. However, all other short-term data reveal adversary capabilities and long-term key adversary capabilities do not allow it. We define the following set of reveals for a state that does not contain this state reveal.

$$\begin{aligned}
&\{\mathsf{LKR}(a) \mid a \in \mathit{Agent}\} \cup\\
&\{\mathsf{STDR}(nt, j) \mid nt \in \mathit{NoteType} \setminus \{\mathsf{State}\}, j \in \mathit{TID}\}
\end{aligned}$$

The session-key reveal of the third reveal condition is not allowed by the SessKey reveal adversary capability with respect to the trust function $\mathit{CRN}_{trusted}$. Moreover other short-term data reveal adversary capabilities and long-term key reveal capabilities do not allow it either. We define the following set of reveals for a state that does not contain this session-key reveal.

$$\begin{aligned}
&\{\mathsf{LKR}(a) \mid a \in \mathit{Agent}\} \cup \mathit{SessKeyR}(i, \mathit{CRN}_{trusted})(tr, th, \sigma) \cup\\
&\{\mathsf{STDR}(nt, j) \mid nt \in \mathit{NoteType} \setminus \{\mathsf{SessKey}\}, j \in \mathit{TID}\}
\end{aligned}$$

The intersection of the sets for a given state is the following set.

$$\begin{aligned}
&\mathit{LKR}_{protected}(i, \{s\})(tr, th, \sigma) \cup \mathit{LKR}_{others}(i)(tr, th, \sigma) \cup\\
&\mathit{RandGenR} \cup (tr, th, \sigma) \cup \mathit{SessKeyR}(i, \mathit{CRN}_{trusted})(tr, th, \sigma)
\end{aligned}$$

Note that the union of the adversary capabilities that are in the adversary compromise model of $BKE(i)$ for this state is a subset of this set. Therefore this origin property is strong enough to prove $\phi_{sec}^{\mathbf{CRN}}$, which uses the adversary compromise model of $BKE(i)$.

Example 4.7: Proof that $\phi_{sec}^{\mathbf{CRN}}$ holds for all reachable states in the CRN protocol We show that $\forall (tr, th, \sigma) \in reachable(\mathbf{CRN}).\ \phi_{sec}^{\mathbf{CRN}}(tr, th, \sigma)$ for the secrecy property $\phi_{sec}^{\mathbf{CRN}}$ of Example 3.4 holds.

$$\begin{aligned} \forall (tr, th, \sigma) \in reachable(\mathbf{CRN}).\ \forall i \in TID.\ role_{th}(i) = C \wedge (tr, th, \sigma) \in BKE(i) \\ \Rightarrow k\sharp i \notin knows(tr) \end{aligned}$$

Proof. We prove this by contradiction. Suppose that the secrecy predicate $\phi_{sec}^{\mathbf{CRN}}$ does not hold for a state $(tr, th, \sigma) \in reachable(\mathbf{CRN})$. Hence, there is a thread i such that $role_{th}(i) = C$, $(tr, th, \sigma) \in BKE(i)$ and $k\sharp i \in knows(tr)$. From applying the origin property $\phi_{origin}^{\mathbf{CRN}}$ to $k\sharp i \in knows(tr)$ and $role_{th}(i) = C$ we conclude the following disjunction, whose disjuncts we have named.

$$\begin{aligned} &\mathsf{LKR}(\sigma(s,i)) \in reveals(tr) \vee && \text{(CRorigin1)} \\ &(\ \exists j \in TID.\ \mathsf{STDR}(\mathsf{State}, j) \in reveals(tr)\ ) \vee && \text{(CRorigin2)} \\ &(\ \exists j \in TID.\ (i,j) \in CRN_{trusted}(tr, th, \sigma) \wedge \mathsf{STDR}(\mathsf{SessKey}, j) \in reveals(tr)\ ) && \text{(CRorigin3)} \end{aligned}$$

In the states $BKE(i)$ of the adversary compromise model, only session-key reveals on threads not trusted by the test thread i with respect to the trust function $CRN_{trusted}$ and long-term key reveals on agents different from $\sigma(s,i)$ are allowed. We verify that the reveals of the reveal conditions are allowed by applying the ALLOWEDREVEALS$_{BKE}$ inference rule to all reveals. By removing all cases that are trivially false due to syntactic inequalities, we the following disjunction.

$$\begin{aligned} &(\ \exists a.\ \mathsf{LKR}(\sigma(s,i)) = \mathsf{LKR}(a) \wedge \sigma(s,i) \neq a\ ) \vee && \text{(CRorigin1')} \\ &(\ \exists j'.\ \mathsf{STDR}(\mathsf{SessKey}, j) = \mathsf{STDR}(\mathsf{SessKey}, j') \wedge (i, j') \notin CRN_{trusted}(tr, th, \sigma) \wedge j' \neq i\ ) \\ & && \text{(CRorigin3')} \end{aligned}$$

Case (CRorigin1') is false due to $\sigma(s,i) = a$ and thereby $\sigma(s,i) \neq \sigma(s,i)$.
Case (CRorigin3') is false because the thread j is trusted. Formally we have $j = j'$ from $\mathsf{STDR}(\mathsf{SessKey}, j) = \mathsf{STDR}(\mathsf{SessKey}, j')$. Hence we have $(i,j) \notin CRN_{trusted}(tr, th, \sigma)$ and $(i,j) \in CRN_{trusted}(tr, th, \sigma)$ from which we conclude false.
This concludes our proof.

■

4.2.4. Authentication Proofs

Due to the use of origin properties, the proofs of secrecy properties have been simplified to case distinctions. We could also make use of origin properties in authentication proofs to get contradictions whenever the adversary knows a message for which we have an origin property. However, we still would have to do all the case distinctions to determine that all reveal-conditions are not allowed by the adversary compromise model, which we have already done in the secrecy properties. Therefore, we favor to use secrecy properties to get contradictions whenever a message meant to be secret is in the knowledge of the adversary.

Whenever we use the LKR_{actor} adversary capability, we still have to rely on origin properties. The reason is that we use the adversary capability LKR_{actor} to verify that authentication properties hold despite of key compromise impersonation attacks. Moreover, we cannot use this capability in adversary compromise models of secrecy properties because it would allow the impersonation of the test thread. Hence we have a different adversary compromise model in the secrecy properties than in the authentication properties and cannot use secrecy properties for authentication proofs. Note that an origin property is only strong enough for an adversary compromise model with LKR_{actor}, if it does not contain long-term key reveals on the actor.

Example 4.8: CRN Authentication Proof
In this example, we show that the non-injective synchronization property $\phi_{auth}^{\mathbf{CRN}}$ from Example 3.5 holds in every reachable state of the **CRN** protocol.

$$\forall(tr, th, \sigma) \in reachable(\mathbf{CRN}).\ \phi_{auth}^{\mathbf{CRN}}(tr, th, \sigma)$$

Proof. We must show that in every reachable state $(tr, th, \sigma) \in reachable(\mathbf{CRN})$ that is in the adversary compromise model $(tr, th, \sigma) \in BKE(i)$ the following formula is implied for every thread that instantiates the role C and has executed (i, C_2).

$$\begin{aligned}\exists j \in TID.\ & role_{th}(j) = S \wedge \\ & \sigma(s,i) = \sigma(s,j) \wedge k\sharp i = \sigma(v,j) \wedge \\ & (i, C_1) \prec_{tr} (j, S_1) \wedge (j, S_3) \prec_{tr} (i, C_2)\ .\end{aligned}$$

Intuitively, the adversary must know the message $\mathsf{h}(k\sharp i)$ before the client i receives it in its last step. Formally, we have $\mathsf{h}(k\sharp i) \prec_{tr} (i, S_2)$ from $(i, S_2) \in steps(tr)$ by rule Input. Hence using Known, we deduce $\mathsf{h}(k\sharp i) \in knows(tr)$. Now we enumerate the cases of how the adversary can learn $\mathsf{h}(k\sharp i)$ by applying the HChain$_{\mathbf{CRN}}$ rule.

$$(\ k\sharp i \prec_{tr} \mathsf{h}(k\sharp i)\ ) \vee \quad \text{(NiSync1)}$$
$$(\ \exists j.\ role_{th}(j) = S \wedge (j, S_3) \prec_{tr} \mathsf{h}(\sigma(v,j)) = \mathsf{h}(k\sharp i)\ ) \quad \text{(NiSync2)}$$

Case (NiSync1) contradicts the secrecy of $k\sharp i$, which we have derived from $k\sharp i \prec_{tr} \mathsf{h}(k\sharp i)$ by rule Known.

Case (NiSync2) is where the adversary has learned the message from the send step S_3 from a sever thread j. From the equality $\mathsf{h}(\sigma(v,j)) = \mathsf{h}(k\sharp i)$ and the injectivity of $\mathsf{h}(\cdot)$ we infer $\sigma(v,j) = k\sharp i$. By using transitivity on $(j, S_3) \prec_{tr} \mathsf{h}(k\sharp i)$ and $\mathsf{h}(k\sharp i) \prec_{tr} (i, C_3)$ we have $(j, S_3) \prec_{tr} (i, C_2)$.

The server thread must have executed also S_1, if he executes S_3. Hence, we are interested in the source of the message received in S_1. Formally, we have $(j, S_3) \in steps(tr)$ by the Exec rule from $(j, S_3) \prec_{tr} (i, C_3)$ and $(j, S_1) \prec_{tr} (j, S_3)$ by the RoleRecv rule. Hence by applying Exec to this, we have $(j, S_1) \in steps(tr)$. We have $\{\!|\sigma(v,j)|\!\}_{\mathsf{pk}_{\sigma(s,j)}} \prec_{tr} (j, S_1)$ from this using Input. This is equal to $\{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}} \prec_{tr} (j, S_1)$ using $\sigma(v,j) = k\sharp i$. Then we use Known for deducing $\{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}} \in knows(tr)$. We enumerate the cases of how the adversary can learn $\{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}}$ by applying the EChain$_{\mathbf{CRN}}$ rule.

$$(\ k\sharp i \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}} \wedge \mathsf{pk}_{\sigma(s,j)} \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}}\ ) \vee \quad \text{(NiSync2.1)}$$
$$(\ \exists i'.\ role_{th}(i') = C \wedge (i', C_1) \prec_{tr} \{\!|k\sharp i'|\!\}_{\mathsf{pk}_{\sigma(s,i')}} = \{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}}\ ) \quad \text{(NiSync2.2)}$$

Case (NiSync2.1) is where the adversary creates the encryption by himself. We have $k\sharp i \in knows(tr)$ from applying rule Known to $k\sharp i \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}}$, which contradicts the secrecy of $k\sharp i$.

Case (NiSync2.2) states that the adversary has learned $\{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}}$ from the send step C_1 of a thread i'. Due to injective encryption, we conclude $k\sharp i' = k\sharp i$ from $\{\!|k\sharp i'|\!\}_{\mathsf{pk}_{\sigma(s,i')}} = \{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}}$. This implies $i' = i$ and we further conclude $\sigma(s,i) = \sigma(s,j)$. Moreover, we get $(i, C_1) \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}}$ by $(i', C_1) \prec_{tr} \{\!|k\sharp i'|\!\}_{\mathsf{pk}_{\sigma(s,i')}}$. Then we use transitivity to connect this to $\{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}} \prec_{tr} (j, S_1)$ and have $(i, C_1) \prec_{tr} (j, S_1)$.

This concludes our proof.

■

Note that this proof is very similar to the proof of the non-injective synchronization property in [19] by Meier, Cremers, and Basin for the **CR** protocol. The reason is that they also use secrecy properties for authentication proofs. Moreover, we use the secrecy property $\phi_{sec}^{\mathbf{CRN}}(tr, th, \sigma)$ whenever the adversary knows $k\sharp i$. Thereby, we do not need any adversary compromise model specific reasoning in the authentication property. Nevertheless there are protocols for which we still do adversary compromise model specific reasoning in the authentication properties.

4.3. Interactive Proof Construction

In [19], Meier, Cremers, and Basin extended Isabelle/HOL with a new proof method for supporting decryption chain reasoning. They call this method *sources.* A call "sources m" applies the Chain rule to m. Then it converts the resulting conclusion into disjunctive normal form and creates a case for each disjunct. Cases that have syntactic inequalities are automatically discharged.[1] Note that a sources call corresponds to applying the for the protocol instantiated chain rule that matches the outermost message constructor of m.

4.4. Best Practices

In this section we explain best practices, we have learned from using our proof procedure. Moreover, we show also interesting observations we have made.

We start by explaining a heuristic for selecting facts and an inference rule that is applied to these facts for the next proof step. In this heuristic we tried to capture our intuition of protocol security proofs. Then we explain how we can make our authentication properties stronger by allowing all reveals after the test thread has finished. Finally we present a method to use origin properties for the proof of other origin properties even if they are formulated for a different role and contain short-term data reveal.

4.4.1. Proof Heuristic

During proofs of real world protocols, we have usually a big set of facts. This makes the situation difficult to select the facts and inference rule that is applied to these facts for the next proof step. Besides the general idea about how a protocol works and why it is secure, we use a heuristic to select the next inference rule and facts to apply the rule to.

In the following subsections, we explain the criteria of our heuristic. We select the action provided by the first criterion that matches. A case can be solved directly, if we can solve it using the automatic tools of Isabelle/HOL.

[1] Note that we also use the sources nomenclature in this report to denote the same process a sources call in Isabelle/HOL does.

Short-Term Data Reveals

Our adversary capabilities for session-key and state reveals do not allow reveals on the test thread, its partners, and threads that the test thread trusts. We define a criterion stating that *whenever the adversary has performed a session-key or state reveal on the test thread, on a partner of the thread, or on a thread that is trusted by the test thread, then we can solve the case directly.*

Long-Term Key Reveals

Long-term key reveals on agents participating in the protocol run of the test thread are usually not allowed. Therefore, we formulate the criterion stating that *whenever there is a long-term key reveal in the facts we solve the case directly.*

Secret Keys Stay Secret

In security protocols no secret long-term keys are sent. Therefore, the adversary has to learn them by performing long-term key reveals. We use this observation as criterion for our heuristic: *We source a secret long-term key in the adversary knowledge whenever we have not already sourced it.*

Cannot Fake

Intuitively, the reason why many security protocols achieve secrecy and authentication is because of signed and encrypted messages the adversary cannot fake. We use this intuition to formalize a criterion for the heuristic. *Whenever we have a signed message, i.e. an encryption with a long-term asymmetric private key, or a message encrypted with a long-term symmetric key in the facts that we have not sourced before, we source it.*

If there are multiple messages available that match this criterion, we use the following observation. At the beginning we know only that the test thread exists. By sourcing a message that the test thread has received, we learn about a second thread that agrees on some variables. The more received messages of the test thread we source, the more variables we know they agree on. Therefore, *we source the message received by the test thread that contains the most variables on which they do not already agree.*

For most protocols, we have two cases from sourcing such a message. The first is that the adversary fakes the message. Therefore, he must know the long-term private key or a long-term symmetric key. The second case is where the message was sent from another thread. Thereby we know that the test thread and the other thread agree on the content of more variables.

Note that when we source a message that is received by the other thread, a new thread identifier variable might be introduced. The reason is that this message might contain message variables for which it is not clear, if they agree on some message of the test thread. This additional thread identifier usually does not help in our proofs. Therefore the best strategy is to show that this thread identifier variable contains the thread identifier of the test thread. We achieve this by showing that the thread that is stored in the new thread identifier variable agrees on a fresh message with the test thread. Since fresh messages are unique, the new thread identifier variable can only store the thread identifier of the test thread.

Make Partners

Whenever there is a note step in the trace that was executed by a thread different from the test thread, then it is likely that the thread is a partner of the test thread, i.e. trusted by the test thread. Therefore, we have to ensure they agree on all messages the partnering function generator uses to determine partners. We formulate this the following criterion

Whenever a thread j has executed a note step and j is not trusted by the test thread, source the message that contains the most variables on which they do not already agree.

Source Secret Nonce of other Role

Our protocol security proofs get smaller and thereby more understandable by using origin properties for secrecy property proofs and secrecy properties for authentication property proofs. We want to make use of this positive effect also for origin proofs. Therefore, we use origin properties in proofs of other origin properties with equally strong reveal-conditions. This should be possible whenever the message that the already shown origin property talks about, is in adversary knowledge. This works fine for origin properties that are formulated from the view of the same role or do not contain short-term data reveal-conditions on the test thread, partners of him, or threads that are trusted by him. There the problem is that we do not know the relation of the test thread of one origin property to the test thread of the other property. Thereby, we also do not know how to relate the corresponding partners and trusted threads. Nevertheless, we present a solution for certain cases in Subsection 4.4.3.

Solve a case directly, if there is an origin property for a message in the adversary knowledge and the properties are from the perspective of the same role. When their role differ, source the message.

Our experience shows that the proofs from sourcing such a message are smaller than the original proofs. The reason is that the two threads might already agree on certain variables. To support this criterion optimally, we first define and prove origin properties for fresh values. Thereby we get two advantages. The first is that the secrecy proofs of message variables naturally depend on the secrecy of fresh messages they store. Therefore, our intention automatically uses the origin properties already shown for the fresh message. The second is that proofs of origin properties for message variables are usually longer than those for fresh messages. Therefore, we profit the most by making these proofs smaller and thereby more understandable.

4.4.2. Authentication Properties are Post-Test-Invariant

Cremers and Basin noticed in [4] that some properties hold also, if the adversary is unrestricted after the test thread has finished its execution. They call such properties *post-test invariants*. They also state that authentication properties fall into this category.

In our security protocol model, our authentication properties are also post-test invariants. Our authentication properties are implications. On the left hand side is the conjunction of a role constraint of the test thread and the insurance that the test thread has executed a specified role step. On the right hand side is the authentication claim.

Authentication claims are conjunctions containing existential quantifiers for introducing other threads, claims that a threads executes a specified role, equality claims on messages, and constraints of the execution order of role steps. As soon as a authentication claim of this form hold, it can never be broken. The reason is that variables are never updated, new basic events are only appended to the trace and thereby cannot disrupt the order of previous ones, and the role of a thread is never changed. Moreover, the conjunction of two operands for which this property holds also has this property.

Therefore, if the authentication claims holds when the left hand side of the implication of the authentication property holds, then the adversary cannot break it anymore. That means our authentication properties would also hold, if the adversary can use any means in our model after the test thread has finished.

4.4.3. Reusing Origin Properties from a Different Role

In Subsection 4.4.1, we discuss the benefits of using origin properties for the proof of other origin properties. There we also explain that using origin properties that contain short-term data reveal-conditions and that are formulated from the perspective of another role for other origin properties does not work in general. In this subsection, we present a method that works in certain cases. Note that although we explain the method for a partnering function, this method can be adopted such that it works also for trust functions.

The reveal-conditions described above as problematic are the following two for a thread j running the role R' from origin property ϕ'.

$$\mathsf{STDR}(nt, j) \in reveals(tr) \tag{RC1}$$

$$\exists i' \in TID.\ (j, i') \in p(tr, th, \sigma) \wedge \mathsf{STDR}(nt, i') \in reveals(tr) \tag{RC2}$$

When we use an origin property ϕ' for the proof of an origin property ϕ, then we need to show that for each reveal-condition of ϕ' there is a reveal-condition in ϕ. Therefore, we have to find a reveal-condition for the test thread i instantiating the role R for the two reveal conditions (RC1) and (RC2).

We could use the reveal-condition from (STDR.UnSpec), i.e. we do not specify the thread that performs the short-term data reveal. However, this weakens the origin property. Therefore, we need a solution using reveal-conditions in the form of (STDR.Test) or (STDR.Part), i.e. of the test thread i or a partner of the test thread.

Intuitively, when a thread j runs a role R' and stores a message that is equal to one that is secret for the test thread i, then the thread j should communicate to the test thread i. Therefore, they should be partners. Hence, the reveal-condition (RC1) is included in a reveal-condition of type (STDR.Part) in ϕ, i.e.

$$\exists j \in TID.\ (i, j) \in p(tr, th, \sigma) \wedge \mathsf{STDR}(nt, j) \in reveals(tr)\ .$$

We explain in Subsection 3.1.5 that whenever there is a condition (pt, pt', st) for the partnering function generator *mkPartnering* such that st is a send step sending a message containing a fresh value that is also in the message pattern pt', then at most one partner can execute st. That means whenever a thread j has two partners i, i' which have executed st then $i = i'$ holds. We formalize this intuition by properties in the following form, which we call *injective-partnering properties.*

$$\begin{aligned}\forall (tr, th, \sigma) \in reachable(\mathbf{P}).\ \forall i\ j\ i' \in TID.\ & \\ & role_{th}(i) = R \wedge role_{th}(j) = R' \wedge \\ & (j, i) \in p(tr, th, \sigma) \wedge (j, i') \in p(tr, th, \sigma) \wedge \\ & (i, st) \in steps(tr) \wedge (i', st) \in steps(tr) \\ & \Rightarrow i = i'\end{aligned}$$

Assume we have the reveal-condition (RC2). Assume further that all note steps that allow the adversary to reveal the message the origin property ϕ' argues about are after st in the role R, i.e. $st <_R st'$ for all such note steps st'. That means the message could have been revealed only from one partner thread of the thread j. We formalize this by applying the injective-partnering property. Then, we show that $(j, i) \in p(tr, th, \sigma)$ holds. From this we have that the test thread i has performed the reveal. Hence the reveal-condition (RC2) is included in a reveal condition of the form (STDR.Test) in ϕ, i.e.[2]

$$\mathsf{STDR}(nt, i) \in \mathit{reveals}(tr) \ .$$

[2]We have exploited this technique in our proof script of the TLS protocol available at [1].

5. Case Studies

We have performed several case studies to assess the effectiveness of our method. These case studies comprise both well-known protocols from the literature as well as artificial protocols. In all case studies our method worked well. We present interesting results of these case studies in this chapter. Case studies in which our method worked as expected and revealed no interesting results are not listed.

We start by explaining findings in the TLS protocol. Then we continue by showing results for the Needham-Schroeder-Lowe protocol. Afterwards, we introduce a protocol which we have created to test several capabilities and present insights we have gained from proving security properties of this protocol. Finally we present information that is valid for all protocols. The formalization of these case studies is available at [1].

5.1. Transport Layer Security (TLS)

In this section we show our results from analyzing a model of the TLS protocol. It is based on the model from Meier, Cremers, and Basin [19] of TLS, which is an adoption of the TLS model from Paulson [22]. We extend this model by adding state reveals between any two communication role steps to represent that all data except the long-term keys is stored in the unprotected system memory. Moreover, we also have randomly generated numbers before the first send step of the roles. However, the pre-master secret is generated by a secure random number generator such that the adversary cannot reveal it by a random number reveal. Finally we have two session-keys in each of the roles. Given the fresh values $nc, ns, sid, pc, ps, pms \in \mathit{Fresh}$, the message variables $nc', ns', sid', pc', ps', pms' \in \mathit{MVar}$, the agent variables $c, s \in \mathit{AVar}$, and the pairwise distinct constants $1, 2, 3, 4, \mathit{PRF}, \mathit{CKEY}, \mathit{SKEY} \in \mathit{Const}$, we define the roles S and C in Figure 5.1 with the abbreviations

$$\begin{aligned}
\mathit{prf}_C &\overset{\text{def}}{=} \mathsf{h}(4, \mathit{PRF}, \mathit{pms}, nc, ns') \\
\mathit{cKey}_C &\overset{\text{def}}{=} \mathsf{h}(\mathit{CKEY}, nc, ns', \mathit{prf}_C) \\
\mathit{sKey}_C &\overset{\text{def}}{=} \mathsf{h}(\mathit{SKEY}, nc, ns', \mathit{prf}_C) \\
\mathit{prf}_S &\overset{\text{def}}{=} \mathsf{h}(4, \mathit{PRF}, \mathit{pms}', nc', ns) \\
\mathit{cKey}_S &\overset{\text{def}}{=} \mathsf{h}(\mathit{CKEY}, nc', ns, \mathit{prf}_S) \\
\mathit{sKey}_S &\overset{\text{def}}{=} \mathsf{h}(\mathit{SKEY}, nc', ns, \mathit{prf}_S) \ .
\end{aligned}$$

We use prf_R to model the pseudorandom function that generates the client key cKey_R and the server key sKey_R using the pre-master secret pms respective pms' as input for the role $R \in \{C, S\}$. Note that the pseudorandom function prf is the same for both roles, if the nonces pms, ns, and nc agree on the values of the message variables pms', ns', and nc'. Hence the session-keys are also the same for both roles. Therefore, if the context does not require the view of a specific role, we use prf, cKey, and sKey without the role.

We define **TLS** $\overset{\text{def}}{=} \{C, S\}$. In the **TLS** protocol the client modeled by the role C sends its agent name together with the nonce nc, the session identifier sid, and its cryptographic preferences pc to the server. The server modeled by the role S answers upon such a message with

$$
\begin{aligned}
C \overset{\text{def}}{=} \langle\ & \mathsf{Note}_1(\mathsf{RandGen}, (nc, sid, pc)),\\
& \mathsf{Note}_2(\mathsf{State}, (nc, sid, pc)),\\
& \mathsf{Send}_3(c, nc, sid, pc),\\
& \mathsf{Note}_4(\mathsf{State}, (nc, sid, pc)),\\
& \mathsf{Recv}_5(ns', sid, ps'),\\
& \mathsf{Note}_6(\mathsf{State}, (nc, sid, pc, pms, ns', ps')),\\
& \mathsf{Send}_7(\{|0, pms|\}_{\mathsf{pk}_s}, \{|1, \mathsf{h}(2, ns', s, pms)|\}_{\mathsf{sk}_c}, \{|3, sid, prf_C, nc, pc, c, ns', ps', s|\}_{cKey_C}),\\
& \mathsf{Note}_8(\mathsf{State}, (nc, sid, pc, pms, ns', ps')),\\
& \mathsf{Recv}_9(\{|3, sid, prf_C, nc, pc, c, ns', ps', s|\}_{sKey_S}),\\
& \mathsf{Note}_{10}(\mathsf{SessKey}, (sKey_C, cKey_C))\ \rangle\\
S \overset{\text{def}}{=} \langle\ & \mathsf{Recv}_1(c, nc', sid', pc'),\\
& \mathsf{Note}_2(\mathsf{RandGen}, (ns, ps)),\\
& \mathsf{Note}_3(\mathsf{State}, (ns, sid', ps, nc', pc')),\\
& \mathsf{Send}_4(ns, sid', ps), \mathsf{Note}_5(\mathsf{State}, (ns, sid', ps, nc', pc')),\\
& \mathsf{Recv}_6(\{|0, pms'|\}_{\mathsf{pk}_s}, \{|1, \mathsf{h}(2, ns, s, pms')|\}_{\mathsf{sk}_c}, \{|3, sid', prf_S, nc', pc', c, ns, ps, s|\}_{cKey_S}),\\
& \mathsf{Note}_7(\mathsf{State}, pms', ns, ps, nc', sid', pc'),\\
& \mathsf{Send}_8(\{|3, sid', prf_S, nc', pc', c, ns, ps, s|\}_{sKey_S}),\\
& \mathsf{Note}_9(\mathsf{SessKey}, (sKey_S, cKey_S))\ \rangle
\end{aligned}
$$

Figure 5.1.: Definition of the roles of the **TLS** protocol.

a nonce ns, the receive session identifier, and its cryptographic preferences ps. Afterwards the client sends a triple consisting of the client key exchange message, the client certificate verify message, and the client-finish message. The client key exchange message is the freshly generated pre-master secret encrypted with the public key of the server. The client certificate verify message is a signature on the hash of the nonce of the server, the server agent name, and the pre-master secret. In the client-finish message, the client computes a message authentication code with the freshly computed client key over all messages transmitted so far, which we represent by an encryption over all variables that are in the messages sent and received so far. Upon the receipt of a triple consisting of these three messages, the server answers with a sever-finish message that is a message authentication code using the freshly generated sever key of all variables and nonces of the server role. Note that we use constants in all encryptions to distinguish them from each other.

We use an asymmetric trust function. In the direction from the role C to the role S all variables and fresh values of the client are in the conditions of the partnering function generator. In the direction from the role S to the role C only the agent names, ns, and the pre-master secret pms are in the conditions. We define the trust function $trusted_{\mathbf{TLS}}$ in Figure 5.2.

5.1.1. Protocol Guarantees

We are interested in the secrecy of the two session-keys $sKey$ and $cKey$. Since these are hashes and they are never sent except as keys, their secrecy depends only on the input of the hash

$$
\begin{aligned}
&\mathit{trusted}_{\mathbf{TLS}}(tr, th, \sigma) \overset{\text{def}}{=}\\
&\quad \mathit{Id} \cup\\
&\quad \mathit{mkPartnering}(C, S, \{\\
&\qquad (c, c, S_1), \quad (s, s, S_1),\\
&\qquad (pc, pc', S_1), \quad (nc, nc', S_1), \quad (sid, sid', S_1),\\
&\qquad (ns', ns, S_4), \quad (ps', ps, S_4), \quad (pms, pms', S_6)\\
&\quad \})(tr, th, \sigma) \cup\\
&\quad \mathit{mkPartnering}(S, C, \{\\
&\qquad (c, c, C_3), \quad (s, s, C_3),\\
&\qquad (ns, ns', C_5), \quad (pms', pms, C_7)\\
&\quad \})(tr, th, \sigma)
\end{aligned}
$$

Figure 5.2.: Definition of the $\mathit{trusted}_{\mathbf{TLS}}$ trust function

function. The only secret input is the pseudorandom function which is a hash that depends on the security of the pre-master secret. Hence the secrecy of the pseudorandom function and the keys depend only on the pre-master secret. Therefore, the origin properties of them depend on the origin property of the pre-master secret and they have the same reveal-conditions. Thus we only present the origin property for the pre-master secret.

If the adversary knows the pre-master secret *pms* of a thread i running the role C, the adversary has either decrypted the second client message sent by C_7, or revealed a state or a session-key from a trusted thread to gain it. For decrypting this message the adversary needs to know the long-term private key of the agent $\sigma(s, i)$. The adversary can learn this key only from a long-term key reveal. For a thread i that instantiates the role S there is another possibility. In this possibility, the adversary can fake the second client message received in S_6. Therefore, he must know the private key of the client $\sigma(c, i)$.

$$
\begin{aligned}
\phi^{\mathbf{TLS}}_{origin,C}(tr, th, \sigma) \overset{\text{def}}{=} & \forall i \in \mathit{TID}.\ \mathit{role}_{th}(i) = C \wedge pms\sharp i \in \mathit{knows}(tr)\\
& \Rightarrow \mathsf{LKR}(\sigma(s, i)) \in \mathit{reveals}(tr) \vee\\
& \quad (\exists j.\ (i, j) \in \mathit{trusted}_{\mathbf{TLS}}(tr, th, \sigma) \wedge \mathsf{STDR}(\mathsf{SessKey}, i) \in \mathit{reveals}(tr)) \vee\\
& \quad (\exists j.\ (i, j) \in \mathit{trusted}_{\mathbf{TLS}}(tr, th, \sigma) \wedge \mathsf{STDR}(\mathsf{State}, i) \in \mathit{reveals}(tr))
\end{aligned}
$$

$$
\begin{aligned}
\phi^{\mathbf{TLS}}_{origin,S}(tr, th, \sigma) \overset{\text{def}}{=} & \forall i \in \mathit{TID}.\ \mathit{role}_{th}(i) = S \wedge (i, S_6) \in \mathit{steps}(tr) \wedge \sigma(pms', i) \in \mathit{knows}(tr)\\
& \Rightarrow \mathsf{LKR}(\sigma(s, i)) \in \mathit{reveals}(tr) \vee\\
& \quad \mathsf{LKR}(\sigma(c, i)) \in \mathit{reveals}(tr) \vee\\
& \quad (\exists j.\ (i, j) \in \mathit{trusted}_{\mathbf{TLS}}(tr, th, \sigma) \wedge \mathsf{STDR}(\mathsf{SessKey}, i) \in \mathit{reveals}(tr)) \vee\\
& \quad (\exists j.\ (i, j) \in \mathit{trusted}_{\mathbf{TLS}}(tr, th, \sigma) \wedge \mathsf{STDR}(\mathsf{State}, i) \in \mathit{reveals}(tr))
\end{aligned}
$$

We prove secrecy and authentication properties with various adversary compromise models. For secrecy properties, the strongest adversary compromise model which allows to prove the session-key secrecy for a test thread i is

$$\{\mathit{LKR}_{others}(i), \mathit{SessKeyR}(i, \mathit{trusted}_{\mathbf{TLS}}), \mathit{StateR}(i, \mathit{trusted}_{\mathbf{TLS}}), \mathit{RandGenR}\}\ .$$

For non-injective agreement we could also add LKR_{actor}. However, proving non-injective synchronization requires that the adversary does not have the *RandGenR* capability. Otherwise the adversary learns the fresh message of the first two messages before they are sent. Thereby he can fake them before they are sent. Hence he can break the order in which the communication steps should occur.

5.2. Needham-Schroeder-Lowe Public-Key Protocol

In this section, we explain our findings for the three step version of the public key protocol from Needham and Schroeder with the fix from Lowe [17]. We extend this protocol by including message tags to distinguish the three messages from each other. Moreover, we use the hash of the two nonces as session-keys after the protocol communication is finished. Given fresh values $ni, nr \in \mathit{Fresh}$, pairwise disjoint constants $1, 2, 3 \in \mathit{Const}$, the message variables $ni', nr' \in \mathit{MVar}$, and the agent variables $i, r \in \mathit{AVar}$, we define $\mathbf{NSL} \overset{\text{def}}{=} \{I, R\}$ where

$$\begin{aligned} I \overset{\text{def}}{=} \langle\ & \mathsf{Send}_1(\{\!|1, ni, I|\!\}_{\mathsf{pk}_R}),\ \mathsf{Recv}_2(\{\!|2, ni, nr', R|\!\}_{\mathsf{pk}_I}),\ \mathsf{Recv}_3(\{\!|3, nr'|\!\}_{\mathsf{pk}_R}), \\ & \mathsf{Note}_4(\mathsf{SessKey}, \mathsf{h}(ni, nr'))\ \rangle \\ R \overset{\text{def}}{=} \langle\ & \mathsf{Recv}_1(\{\!|1, ni', I|\!\}_{\mathsf{pk}_R}),\ \mathsf{Send}_2(\{\!|2, ni', nr, R|\!\}_{\mathsf{pk}_I}),\ \mathsf{Recv}_3(\{\!|3, nr|\!\}_{\mathsf{pk}_R}), \\ & \mathsf{Note}_4(\mathsf{SessKey}, \mathsf{h}(ni', nr))\ \rangle\ . \end{aligned}$$

In the protocol, the initiator modeled by the role I encrypts the constant 1, the fresh value ni, and its agent name with the public key of responder and then sends it. The responder, modeled by the role R, answers on the receipt of such a message. He encrypts the constant 2, the message variable ni', the fresh value nr, and its agent name with the public key of the initiator and sends it. Upon receiving such a message, the initiator responds by sending the encryption of the constant 3 and the received message nr' using the public key of the server. In the end both have a session-key consisting of the hash of the two nonces.

5.2.1. Guarantees

We use a trust function that includes all variables and nonces as conditions. We show that the adversary has to perform a long-term key reveal on one of the agents $\sigma(R, i)$ or $\sigma(I, i)$ for learning one of the nonces for an agent i. The reason why he cannot learn them from a session-key reveal is that we use the hash of both nonces as session-key.

The adversary can learn the session-key either by creating it himself or by a session-key reveal on one of the trusted threads. For creating the session-key himself the adversary must know both nonces and he can learn them only if he performs a long-term key reveal. Therefore, the nonces and the session-keys are secret against an adversary with the LKR_{others} and SessKey capabilities. Moreover, we also show non-injective synchronization for both roles against the adversary with these capabilities.

5.2.2. Discussion

We tried to extend the protocol further by adding note steps with the note type State between any two communication steps, i.e. both roles store only the long-term keys in secure storage. This resulted in the problem that the adversary can reveal ni from a non-trusted thread and thereby break the secrecy of it in an adversary compromise model containing the *StateR* adversary capability.

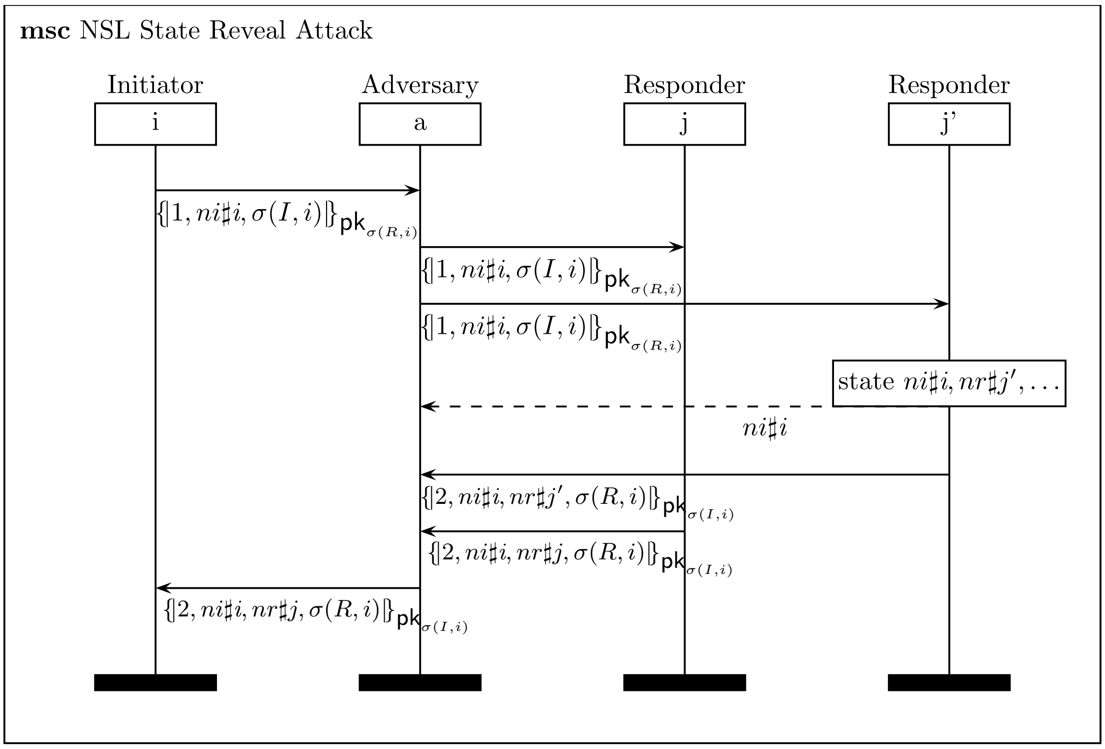
msc NSL State Reveal Attack
Initiator
Adversary
Responder
Responder
i
a
j
j'
$\{\!|1, ni\sharp i, \sigma(I,i)|\!\}_{\mathsf{pk}_{\sigma(R,i)}}$
$\{\!|1, ni\sharp i, \sigma(I,i)|\!\}_{\mathsf{pk}_{\sigma(R,i)}}$
$\{\!|1, ni\sharp i, \sigma(I,i)|\!\}_{\mathsf{pk}_{\sigma(R,i)}}$
state $ni\sharp i, nr\sharp j', \ldots$
$ni\sharp i$
$\{\!|2, ni\sharp i, nr\sharp j', \sigma(R,i)|\!\}_{\mathsf{pk}_{\sigma(I,i)}}$
$\{\!|2, ni\sharp i, nr\sharp j, \sigma(R,i)|\!\}_{\mathsf{pk}_{\sigma(I,i)}}$
$\{\!|2, ni\sharp i, nr\sharp j, \sigma(R,i)|\!\}_{\mathsf{pk}_{\sigma(I,i)}}$

Figure 5.3.: Attack on the **NSL** protocol with state reveals.

We demonstrate this problem in the message sequence chart in Figure 5.3. Assume the thread i instantiates the role I and that the threads j and j' are two threads running the role R. After the thread i sends its first message I_1, both threads j and j' receive this message. Then the adversary reveals $\sigma(ni', j') = ni\sharp i$ from the unprotected system state of the thread j'. Afterwards the threads j and j' answer with their respective message from R_2. Note that the fresh value nr is different in both messages. Therefore, the thread i can receive only one of them. Assume the thread i receives the message of the thread j. Hence he does not agree on the value $\sigma(nr', i) \neq nr\sharp j$ with the thread j' and j' is not trusted by i anymore. Hence, the state reveal to j' is legitimate in the *StateR* adversary capability.

5.3. wPFS

In this section we introduce a protocol we call **wPFS**. We have created this protocol to test the capability $\mathit{LKR}_{aftercorrect}$ and thereby get experiences in expressing weak perfect forward secrecy in our proof method. As we explain later, we were disappointed, but gained useful insights in modeling different adversary models nevertheless.

Let $k, n, sid \in \mathit{Fresh}$, $1, 2 \in \mathit{Const}$, $sid', k', pkN \in \mathit{MVar}$ and $s, c \in \mathit{AVar}$. We define the protocol **wPFS** $\overset{\text{def}}{=} \{C, S\}$ where C and S are defined in Figure 5.4.

In this protocol, the client modeled by the role C uses the fresh value n to generate a fresh public and private pair pk_n and sk_n. He signs the tuple of the constant 1, the session identifier sid, the server's agent name, and the freshly generated public key pk_n by his private key sk_c and sends it. The server, modeled by the role S, signs upon receive of such a message the

$$
\begin{aligned}
C \overset{\text{def}}{=} \langle\ & \mathsf{Note}_1(\mathsf{State}, (sid, n, \mathsf{pk}_n, \mathsf{sk}_n, s, c)),\\
& \mathsf{Send}_2(\{\!|(1, sid, s, \mathsf{pk}_n)|\!\}_{\mathsf{sk}_c}),\\
& \mathsf{Note}_3(\mathsf{State}, (sid, n, \mathsf{pk}_n, \mathsf{sk}_n, s, c)),\\
& \mathsf{Recv}_4(\{\!|\{\!|(2, sid, c, k')|\!\}_{\mathsf{sk}_s}|\!\}_{\mathsf{pk}_n}),\\
& \mathsf{Note}_5(\mathsf{State}, (k', sid, s, c)),\\
& \mathsf{Note}_6(\mathsf{SessKey}, k')\ \rangle\\
S \overset{\text{def}}{=} \langle\ & \mathsf{Recv}_1(\{\!|(1, sid', s, pkN)|\!\}_{\mathsf{sk}_c}),\\
& \mathsf{Note}_2(\mathsf{State}, (sid', pkN, k, s, c)),\\
& \mathsf{Send}_3(\{\!|\{\!|(2, sid', c, k)|\!\}_{\mathsf{sk}_s}|\!\}_{pkN}),\\
& \mathsf{Note}_4(\mathsf{State}, (k, sid', s, c)),\\
& \mathsf{Note}_5(\mathsf{SessKey}, k)\ \rangle
\end{aligned}
$$

Figure 5.4.: Definition of the roles C and S in the **wPFS** protocol.

tuple consisting of the constant 2, the session identifier sid', the client's agent name and a fresh session-key k by his private key sk_s, encrypts the result with the received public key pkN and sends it.[1]

Besides the secret long-term keys of the agents, no other information is stored protected. Therefore, before and after every send role step and after every receive role step, there is data currently available in the unprotected system memory. Moreover, in the end a new session-key is available in both roles.

A thread i trusts a thread j, if $i = j$ or both threads agree on the contents of the agent variables s and c and they have the same session-identifier. We increase the strength of this trust function by adding the condition that a thread in the role C trust a thread in the role S only if they agree on the session-key after S_3. We formalize this in the trust function $trusted_{\mathbf{wPFS}}$.

$$
\begin{aligned}
& trusted_{\mathbf{wPFS}}(tr, th, \sigma) \overset{\text{def}}{=}\\
& \quad Id \cup\\
& \quad mkPartnering(C, S, \{\\
& \qquad (sid, sid', S_1), \quad (k', k, S_3),\\
& \qquad (c, c, S_1), \qquad\ \ (s, s, S_1)\\
& \quad \})(tr, th, \sigma) \cup\\
& \quad mkPartnering(S, C, \{(sid', sid, C_1), (c, c, C_1), (s, s, C_1)\})(tr, th, \sigma)
\end{aligned}
$$

5.3.1. Protocol Guarantees

We are interested in the secrecy of the session-key. Therefore, we create an origin property for the session-key for both roles. Intuitively, the adversary can learn the session-key either by decrypting or faking the message sent by the server in S_3, or by performing a state or

[1] Note that due to the use of a fresh asymmetric key pair, we have to ensure that whenever the fresh value n is noted or sent, the fresh secret key sk_n and fresh pk_n are too. We provide further explanations about the modeling of fresh public and private keys in Section 2.3.

session-key reveal. For decrypting this message, the adversary must know the fresh private key sk_n and he can learn this fresh private key only by a state reveal on the client.

Another possibility is that the adversary fakes the message sent by the client in C_2. Thereby he includes a public-key of which he knows the corresponding private key and breaks the decryption of the message sent by the server in S_3. Therefore, he must know the long-term private sk_c before S_1. Moreover, he also needs to fake the message the client receives in C_4. Therefore, he must know the long-term private key sk_s before C_4 to fake the signature. We formalize these reveal-conditions in the origin properties $\phi^{\mathbf{wPFS}}_{origin,S}(tr, th, \sigma)$ for the server role and $\phi^{\mathbf{wPFS}}_{origin,C}(tr, th, \sigma)$ for the client role.[2]

$$\begin{aligned}\phi^{\mathbf{wPFS}}_{origin,S}(tr, th, \sigma) \overset{\text{def}}{=} {} & \forall i \in \mathit{TID}.\ \mathit{role}_{th}(i) = S \wedge k\sharp i \in \mathit{knows}(tr)\\ & \Rightarrow \mathsf{LKR}(\sigma(c,i)) \prec_{tr} (i, S_3) \vee\\ & \quad (\exists j.\ (i,j) \in \mathit{trusted}_{\mathbf{wPFS}}(tr, th, \sigma) \wedge \mathsf{STDR}(\mathsf{SessKey}, i) \in \mathit{reveals}(tr)) \vee\\ & \quad (\exists j.\ (i,j) \in \mathit{trusted}_{\mathbf{wPFS}}(tr, th, \sigma) \wedge \mathsf{STDR}(\mathsf{State}, i) \in \mathit{reveals}(tr))\end{aligned}$$

$$\begin{aligned}\phi^{\mathbf{wPFS}}_{origin,C}(tr, th, \sigma) \overset{\text{def}}{=} {} & \forall i \in \mathit{TID}.\ \mathit{role}_{th}(i) = C \wedge (i, C_4) \in \mathit{steps}(tr) \wedge \sigma(k', i) \in \mathit{knows}(tr)\\ & \Rightarrow \mathsf{LKR}(\sigma(c,i)) \prec_{tr} (i, C_4) \vee\\ & \quad \mathsf{LKR}(\sigma(s,i)) \prec_{tr} (i, C_4) \vee\\ & \quad (\exists j.\ (i,j) \in \mathit{trusted}_{\mathbf{wPFS}}(tr, th, \sigma) \wedge \mathsf{STDR}(\mathsf{SessKey}, i) \in \mathit{reveals}(tr)) \vee\\ & \quad (\exists j.\ (i,j) \in \mathit{trusted}_{\mathbf{wPFS}}(tr, th, \sigma) \wedge \mathsf{STDR}(\mathsf{State}, i) \in \mathit{reveals}(tr))\end{aligned}$$

These two origin properties hold for all reachable states of the **wPFS** protocol.

According to our reveal-conditions, we can allow the adversary to perform session-key and state reveals to untrusted threads and still prove the secrecy of the session-key. Moreover, we can also allow long-term key reveals on agents not communicating to the test thread and after the test thread has finished communicating. We formalize this in the function CK that given a thread returns an adversary compromise model [3, Table 2].

$$\begin{aligned}CK \overset{\text{def}}{=} \lambda i.\ \{ & \mathit{LKR}_{others}(i), \mathit{LKR}_{after}(i),\\ & \mathit{SessKeyR}(i, \mathit{trusted}_{\mathbf{wPFS}}), \mathit{StateR}(i, \mathit{trusted}_{\mathbf{wPFS}})\}\end{aligned}$$

5.3.2. Discussion

In our case study of the **wPFS** protocol, we show that the session-key of a thread i is secret for both roles in all states satisfying the adversary compromise model $CK(i)$. That means we prove that a two communication step protocol can achieve perfect secrecy which should not be possible due to Bellare, Pointcheval, and Rogaway [5]. The reason why our protocol still achieves it, is due to the use of a different model. In their model, the adversary can reveal also short-term data after the test thread is finished with executing its role. This is not captured in the compromising adversaries model. Note that due to the event order, adversary capabilities that allow short-term data reveals after the last communication role step can be created for our proof method.

[2]Note that we do not use conjunctions of reveals in our reveal conditions. The reason is that in the adversary compromise model function *acm*, we check that each reveal is justified by an adversary capability independent of any other reveals. Therefore the reveal condition is disallowed, if one of the reveals in the conjunction is not allowed. Hence we use the reveal as reveal condition that will be disallowed by the adversary compromise model anyway.

We also show that whenever a thread that instantiates the role C and has executed C_3 then there is a server thread such that they agree on the session-key, the agent variables, the fresh public key, and the session identifier. We cannot give such strong guarantees for a thread running the role S. There we prove that whenever the thread has executed S_3 then there exists a client thread that agrees on the session identifier, the agent variables, and the fresh public key. The reason why we cannot show that they also agree on the session-key is because the adversary can drop the message sent from the server to the client by not delivering it. Thereby the client never learns the session-key.

Our trust function $trusted_{\mathbf{wPFS}}(tr, th, \sigma)$ does not have symmetric conditions for the two roles. The reason why we do not include the session-key in the direction from the role S to the role C is that it allows an attack on the secrecy on the session-key from the perspective of the role S. We demonstrate this attack in the message sequence chart in Figure 5.5. Assume a thread i instantiates the role S and a thread j instantiates the role C. The thread j sends the message

$$\{\!|(1, sid\sharp j, \sigma(s,j), \mathsf{pk}_{n\sharp j})|\!\}_{\mathsf{sk}_{\sigma(c,j)}}$$

and the thread i receives the message

$$\{\!|(1, \sigma(sid', i), \sigma(s,i), \sigma(pkN, i))|\!\}_{\mathsf{sk}_{\sigma(c,i)}}$$

such that the following equations hold.

$$sid\sharp j = \sigma(sid', i) \qquad \sigma(s,j) = \sigma(s,i) \qquad \mathsf{pk}_{n\sharp j} = \sigma(pkN, i) \qquad \sigma(c,j) = \sigma(c,i)$$

Then the adversary reveals the unprotected system state of the thread i that C_2 provides. Thereby he learns $\mathsf{sk}_{n\sharp j}$. This enables him to decrypt the message sent

$$\{\!|\{\!|(2, \sigma(sid', j), \sigma(c,j), k\sharp j)|\!\}_{\mathsf{sk}_{\sigma(s,j)}}|\!\}_{\sigma(pkN,j)}$$

by the thread j in S_3. S_3 is the last communication step of S therefore the adversary is allowed to perform a long-term key reveal on the agent running j, i.e. $\sigma(s,j)$. Thereby he learns $\mathsf{sk}_{\sigma(s,j)}$. The adversary can create the message

$$\{\!|\{\!|(2, \sigma(sid', i), \sigma(c,i), \sigma(k', i))|\!\}_{\mathsf{sk}_{\sigma(s,i)}}|\!\}_{\mathsf{pk}_{n\sharp j}}$$

such that the session-key is different, i.e. $\sigma(k', i) \neq k\sharp j$ holds. A state containing this sequence of events is in $acm(CK(i))$, if the state reveal on the thread i in C_3 is justified. This is the case, if j is not trusted by i and this is the case, if we include the session-key into the trust function of the direction.

Note that this attack is not possible, if we relax the adversary compromise model in the CK function by replacing LKR_{after} with $\mathit{LKR}_{aftercorrect}$. This does not contradict the fact that this protocol achieves perfect forward secrecy. The reason is that many models from the computational approach use only the session identifier and the agent names to determine partners and in this setting the **wPFS** protocol achieves perfect forward secrecy.

5.4. General Discussion

During our case studies, we made the following three protocol-independent observations. The first concerns modeling randomly generated numbers. The second concerns variables used as conditions for the trust functions. The third concerns the proof construction times.

We model the randomly generated numbers that are available at some point in a role with a note step of note type RandGen. Therefore, we should do this also for freshly generated

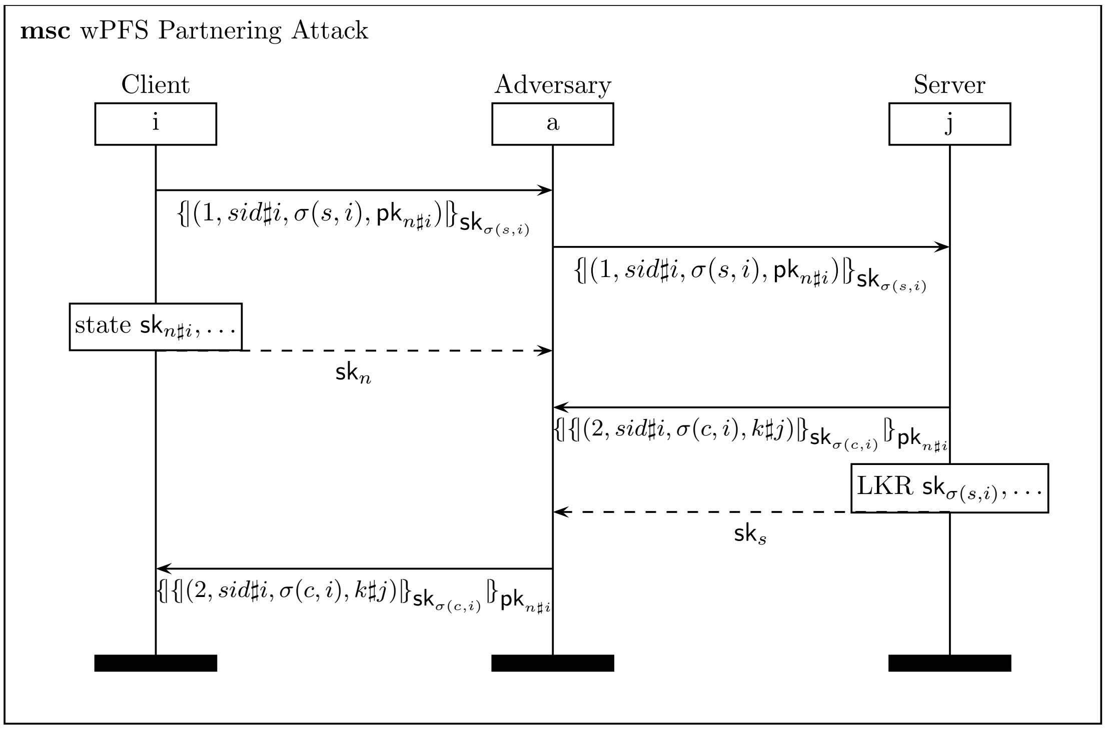
msc wPFS Partnering Attack
Client
Adversary
Server
i
a
j
$\{\!|(1, sid\sharp i, \sigma(s,i), \mathsf{pk}_{n\sharp i})|\!\}_{\mathsf{sk}_{\sigma(s,i)}}$
$\{\!|(1, sid\sharp i, \sigma(s,i), \mathsf{pk}_{n\sharp i})|\!\}_{\mathsf{sk}_{\sigma(s,i)}}$
state $\mathsf{sk}_{n\sharp i}, \ldots$
sk_n
$\{\!|\{\!|(2, sid\sharp i, \sigma(c,i), k\sharp j)|\!\}_{\mathsf{sk}_{\sigma(c,i)}}|\!\}_{\mathsf{pk}_{n\sharp i}}$
LKR $\mathsf{sk}_{\sigma(s,i)}, \ldots$
sk_s
$\{\!|\{\!|(2, sid\sharp i, \sigma(c,i), k\sharp j)|\!\}_{\mathsf{sk}_{\sigma(c,i)}}|\!\}_{\mathsf{pk}_{n\sharp i}}$

Figure 5.5.: Attack on the **wPFS** protocol with a stronger trust function.

values used as session-keys for which we want to show that they are secret. However, since the *RandGenR* adversary capability has no restrictions on which thread the adversary is allowed to perform random number generator reveals, we cannot give the adversary this capability. Otherwise no fresh value can be secret.

All the models of protocols that we introduce in this chapter have an asymmetric trust function. The reason is that without these trust functions messages are not secret anymore. We observed this also for other non trivial protocols. The reason is that they all depend upon the secrecy of fresh values sent in messages of the first send step that sends an encrypted message. The adversary reveals these values from the state of the other thread and breaks the trust or partnering between the two threads. Thereby this attack becomes legitimate and the secrecy of the fresh value is broken.

We use a modified version of decryption chain reasoning of Meier, Cremers, and Basin [19] for our protocol security proofs. We achieve almost the same proof construction time as the original version of the decryption chain reasoning method in interactive poofs, if we do not model any short-term data. The reason is that in this situation, our proof procedure does only differ by the use of origin properties and adversary compromise models. The proofs of origin properties are the same as secrecy proofs in the original decryption chain reasoning. Moreover, our secrecy proofs are automatically derived by the theorem prover with the origin property as input.

Our proof construction time increases when we model that short-term data is available. The reason is that not only the increased modeling labor for the protocol and the trust functions needs time, but also the proofs of origin properties need more time. The reason is that the adversary has more possibilities to learn a message. Therefore, each application of the chain rule results in more cases. The cases where the adversary learns the message by a short-term data reveal on the test thread are solved automatically. However, short-term data reveals on other threads need further applications of the chain rule to ensure that these threads are trusted. Moreover, we can also only make use of origin properties form different roles, if we prove additional properties.

It is hard to give an estimate on how much more time we need to prove a protocol with short-term reveals compared to one without short-term data because it depends on the number of times short-term data is available, i.e. the number of note steps in the roles, the number of roles, and the trust function. Moreover, in interactive proving the person that proves the properties also has an influence on the time and thereby might falsify the results. Therefore we decided to not only consider the actual time we needed for the proofs, but back this information up with the proof script length and the number of applications of the chain rule.

Based on our case studies, we guess that with an almost linear amount of note steps compared to communication steps[3], we need around two to three times the time we need without short-term data. The length of the proofs for the security properties increases by a maximum of three times the size without short-term data reveals. Moreover, the additional number of chain rule applications is also limited by three times the number of applications in the original proof. Therefore, we only model short-term data, if we allow the adversary the corresponding reveal in the adversary compromise models.

[3] Note that we use a linear amount of note steps because we usually model that there is a state between any two communication role steps, at the beginning randomly generated numbers are computed, and in the end session-keys have been established

6. Related Work

We discuss related work concerning other symbolic approaches for machine-checkable protocol security proofs, other partnering definitions, and other tools that support compromising adversaries.

6.1. Machine-Checkable Security Protocol Proofs

Paulson proposed in [21] the inductive approach, which is the first method to construct machine-checkable protocol security proofs. This method uses a shallow embedding of the protocol in higher-order logic and uses the theorem prover Isabelle/HOL to verify and build the protocol security proofs. On one hand his method is very powerful because it has the whole power of higher-order logic to express the protocol. On the other hand the proof construction times are two orders of magnitude slower than those from Meier, Cremers, and Basin [19]. Therefore the inductive approach is also slower than our method. Paulson introduces *Oops* rules to represent the loss of session-keys. Therefore Oops rules can be seen as session-key reveals. These rules add note events to the trace. However, these note events have a different meaning than the note role steps we introduce in this thesis. While we are representing the leakage of data with note role steps that are in the trace, in the inductive approach note events can be added to the trace without increasing the knowledge of the adversary. In the inductive approach the adversary learns the noted value only, if the agent is corrupt.

We have extended the work from Meier, Cremers, and Basin [19] in several directions. First, our security protocol model allows modeling short-term data reveals and the long-term key reveals on agents are dynamic. Therefore, we can express properties like perfect forward secrecy and resistance against session-key compromises. Second, we have extended their proof method for efficient interactive proof construction by including support for the stronger adversary models the compromising adversaries offer us. Our proof construction times are comparable to them in the same adversary models. However, when we model the availability of short-term data and use a stronger adversary model then the proof construction time increase. Nevertheless, we estimate that with a linear amount of note steps, we need around three times more proof construction time. Therefore, we guess that our proof construction time increases asymptotically linear with the number of short-term data that is available in the roles.

6.2. Partnering Definitions

In the cryptographic approach, several models for key exchange protocols are proposed such as in [5–8] by Bellare, Rogaway, Pointcheval, Canetti, Krawczyk. The corresponding proof methods currently do not provide support for machine-checked protocol security proofs. Moreover, there are many different and incompatible models defined in this research community, which is pointed out in publications such as in [11] from Cremers and [16] from Choo, Boyd, and Hitchcock. By using the compromising adversary models from Cremers and Basin [4], we can express many of them. Moreover, due to our partnering function, we can also express subtle model differences due to the partnering definition. This is important because incompatible partnering definitions is one reason that makes many of the proposed models and even model families incompatible.

Matching histories is a partnering definition that is used by Bellare and Rogaway in [6], by Cremers and Basin in [3, 4], and others. With matching histories a thread j is a partner of a thread i, if every message sent by i is received by j and every message received by i was sent by j. We model matching histories with our partnering function generator *mkPartnering* by adding conditions for all variables and fresh values in the role or adding conditions for all messages. As pointed out by Choo, Boyd, and Hitchcock in [16] and others, this notion can be too strong. Assume that the adversary changes an unprotected part of a message exchanged between j and i. Although this modification could have no relevance on the security and the functionality of the protocol, the two threads are not partners anymore after one of them receives the modified message. Therefore the adversary is allowed to reveal short-term data of them and thereby break the secrecy of other messages.

In other definitions such as in [5] by Bellare, Pointcheval, and Rogaway, session identifiers with other information such as agent names are used to determine the partners. To extend this approach also to protocols without session identifiers, Choo, Boyd, and Hitchcock suggest that the concatenation of all fresh values can be used as session identifier [16]. We call the method with a session identifier the *explicit session identifier* method and the method that builds a session identifier from the fresh values the *implicit session identifier* method. We implement the explicit session identifier method by creating a condition for the session identifier and one for each agent variable. For the implicit session identifier method, we create a condition for each fresh value of the roles.

Bellare and Rogaway use a partnering function in [7]. Therefore, compared to the other introduced partnering definitions, this definition matches our definition the most. However, the difference is that their partnering function maps to at most one partner. Hence this definition does not allow the modeling of matching histories and the session-identifier methods because they might allow multiple partners. We can model this partnering function with our partnering-function generator by adding a condition containing a fresh value of the partner with the first role step as guard step. Although this does not match their definition completely, there is also at most one partner that has executed its first role step.

6.3. Tool Support for Compromising Adversaries

Cremers and Basin report in [3] that adversary compromise models are implemented in the automatic protocol security verification tool Scyther [10,12]. Scyther does automatic verification and therefore requires less user expertise and is orders of magnitudes faster. However, this tool does not support the generation of machine-checked proofs. Therefore, by using such a verification tool the protocol verifier must trust the tool. In our approach the protocol verifier trust assumption is limited to our model definition and the theorem prover Isabelle/HOL.

In the following, we compare our integration of compromising adversaries to the one proposed by Cremers and Basin in [4]. First, we provide a general comparison of the security protocol models. Then we focus on how short-term data is modeled and how it can be revealed by the adversary. Finally, we compare the expressiveness.

6.3.1. General Comparison

A major difference is the formalization of compromising adversaries in the security protocol model. While they are proposing a *parameterized security protocol model* that is parameterized with the adversary compromise model, our security protocol model produces all states even if the adversary performed disallowed reveals. We propose a filtering approach that checks in a pre-condition of the security properties whether the adversary has not exceeded his capabilities. As a consequence, our transition system is not dependent on the test thread or the test role.

We do not model the creation of new threads and do not update free variables in receive steps. However, we consider all possible sets of threads in the initial states of our transition system and overapproximate the content of the variables by instantiating them arbitrarily in the initial state. Therefore we do not have to consider the test substitution.

6.3.2. Availability of Short-Term Data

Cremers and Basin use *state*, *generate*, and *sesskeys* agent events to mark a term as short-term data of the respective type. Terms are like a combination of our messages and message patterns. The semantics of generate and sesskeys is that the adversary can reveal all marked terms of the category of a thread by performing a corresponding reveal on the thread. For terms marked as state, the adversary has only time until either the next state or the end of the thread execution to reveal them. The reveals are modeled by adversary compromise transition rules. These rules have premises to ensure that the adversary does not reveal data from certain threads which would make him fundamentally too strong. The adversary might not be able to perform a reveal at the time when the term is marked, but at some later point, if the premises of the corresponding adversary compromise rule are satisfied.

In contrast, the note steps in our model do not mark messages for later reveal, but mark the positions in which the adversary can reveal short-term data. Therefore, in our security protocol model, the adversary either reveals short-term data at a note step or never. Note that this does not weaken the adversary. The reason is that a later action might legitimate a reveal that happened before in our implementation. Hence, if a state is in the adversary compromise model, then a state that consists of a subtrace of this state might not be legitimate. The method of Cremers and Basin satisfies this prefix closedness. However, it is not clear, if we would have an advantage by this prefix closedness or if it just weakens the adversary. It weakens the adversary because in our implementation more state reveals are possible. The reason is that these state reveals might be made legitimate after the next state. In their model the adversary cannot perform reveals on such states because they are overwritten by the next state.

A consequence of the modeling of short-term data reveals rather than of data markers is that we do not have to ensure that threads are not finished executing their role. The reason is that the note steps are in the role. Therefore the thread cannot be finished, if there are still steps in the role. Hence the session-key reveals are allowed when the state reveals are allowed. We also model the LKR_{after} capability different from the corresponding adversary compromise rule. They allow the adversary to perform long-term key reveals before the test thread has finished executing his role. We only consider states in which the adversary has not performed long-term key reveals after the last communication role step. Hence, if there are note role steps at the end of a role, our adversary can already start revealing the long-term keys, while in their model he needs to wait until the corresponding marker agent events happened.

6.3.3. Expressiveness

We provide a flexible partnering implementation in our security protocol model while they propose a solution based on matching histories. Thereby they cannot model certain adversary models from the literature, if these use a different partnering implementation.

Our adversary capabilities are also not restricted to the ones we present in this report. With our filtering based approach new ones can be defined easily and then integrated into the adversary compromise models for security protocol proofs.

7. Conclusion

In this thesis, we provide a method for the efficient interactive construction of machine-checked protocol security proofs in the context of compromising adversaries. For this method, we have successfully adopted the security protocol model from [19] to support compromising adversaries. However, we decided to define the partnering and the integration of adversary compromise models differently than proposed by Cremers and Basin in [3]. We use a partnering function to define which threads are partners. With this function, we can model all partnering definitions known to use. That means we generalize these definitions. Our adversary compromise models are a part of the security property rather than of the transition system for the protocol execution. This allows use to integrate new adversary capabilities easily. Thereby we can express any capability along the dimensions of *whose* data is revealed, *when* the reveal occurs, and *which* kind of data is revealed. This allows to generalize the adversary compromise rules of [3] along all three dimensions of compromising adversaries.

To support our definition of compromising adversaries, we have developed a guideline of how we express security properties for our security protocol model. Security properties that are expressed according to this guideline are well integrated in our method because we can reuse already established security properties for the proof of other security properties. As base property we show an origin property per message meant to be secret. This origin property lists all attack possibilities the adversary has to learn the message. This gives us an adversary compromise model independent view on the guarantees of the protocol. For secrecy proofs we verify that all attacks listed in the origin property are not allowed according to the adversary model in use. Therefore, we can use an origin property for secrecy proofs in different adversary compromise models. As already proposed by [19], we use secrecy properties to shorten authentication proofs. For proving such properties we use an adopted version of the decryption chain reasoning [19]. This provides us with an efficient method for constructing machine-checked protocol security proofs interactively. We have assessed the effectiveness of our method in several case studies of well-known protocols from the literature.

The proof construction times of our method are comparable to the original version of decryption chain reasoning with a comparable adversary model. However, due to short-term data reveals there are more ways in which the adversary can learn a message. Hence more case distinctions in the protocol security proofs are needed. This results in increased proof construction times. We estimate the additional time based on our case studies to two to three times the time of the original decryption chain reasoning.

7.1. Future Work

We see future work in the following areas.

- Although our proof construction method is more efficient than others, our proof construction is still labor intensive. Therefore, the adoption of the automatic proof generation tool from [19] to our method is desirable.
- In the computational approach, security notions for key-exchange protocols have two conditions. The first is that without adversary interference, a session-key is established

and the second is that the adversary cannot learn the session-key. We have concentrated our work effort on the second condition. Therefore, our proof method should be extended by proofs that there is a reachable state in a protocol where two threads have shared a session-key.

- In Section 2.3, we explain how we model freshly generated asymmetric key pairs. However, this method is error prone. Therefore, the integration of a better model for freshly generated asymmetric key pairs is preferable.

- We have two long-term symmetric keys shared between any two agents a and b, i.e. $\mathsf{k}_{a,b}$ and $\mathsf{k}_{b,a}$. However, some protocols require only one shared key between them. We cannot currently model this situation.

- During our case studies, we could not test our $\mathit{LKR}_{aftercorrect}$ capability to express weak perfect forward secrecy. Therefore, a possible future work consists of finding a protocol expressible in our security protocol model that only satisfies $\mathit{LKR}_{aftercorrect}$ and not LKR_{after}. Then the $\mathit{LKR}_{aftercorrect}$ capability can get tested by proving security properties with the adversary capability $\mathit{LKR}_{aftercorrect}$ for this protocol.

- Many modern key exchange protocols like HMQV [15] are authenticated extensions of the Diffie-Hellman key exchange [14]. The Diffie-Hellman key exchange is based on group operations in finite fields, e.g. modular exponentiation. Such operations are not supported by our security protocol model. Therefore, it must be extended to support such protocols.

Bibliography

[1] Source code of the Isabelle/HOL formalizations of the security protocol model, the proof method, and the examples presented in this thesis. April 2011. `http://www.infsec.ethz.ch/research/software`.

[2] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[3] David Basin and Cas Cremers. Modeling and analyzing security in the presence of compromising adversaries. In *Computer Security - ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 340–356. Springer, 2010.

[4] David A. Basin and Cas J.F. Cremers. Degrees of security: Protocol guarantees in the face of compromising adversaries. In *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010.

[5] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. Cryptology ePrint Archive, Report 2000/014, 2000.

[6] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Proceedings of the 13th annual international cryptology conference on Advances in cryptology*, pages 232–249, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[7] Mihir Bellare and Phillip Rogaway. Provably secure session key distribution: the three party case. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 57–66, New York, NY, USA, 1995. ACM.

[8] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. pages 453–474. Springer-Verlag, 2001.

[9] Kim-Kwang Raymond Choo, Colin Boyd, and Yvonne Hitchcock. Errors in computational complexity proofs for protocols, 2005.

[10] Cas J.F. Cremers. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 119–128, New York, NY, USA, 2008. ACM.

[11] Cas J.F. Cremers. Formally and practically relating the ck, ck-hmqv, and eck security models for authenticated key exchange. June 2009.

[12] C.J.F. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc.*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.

[13] C.J.F. Cremers, S. Mauw, and E.P. de Vink. Injective synchronisation: An extension of the authentication hierarchy. *Theor. Comput. Sci.*, 367:139–161, 2006.

[14] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[15] Hugo Krawczyk. HMQV: A high-performance secure diffie-hellman protocol. In *Protocol, Advances in Cryptology - CRYPTO 05, LNCS 3621*, pages 546–566. Springer-Verlag, 2005.

[16] Kim kwang Raymond Choo, Colin Boyd, Yvonne Hitchcock, and Greg Maitland. On session identifiers in provably secure protocols - the bellare-rogaway three-party key distribution protocol revisited. In *In Carlo Blundo and Stelvio Cimato, editors, Fourth Conference on Security in Communication Networks - SCN 2004 Proceedings, volume 3352 of Lecture Notes in Computer Science*, pages 351–366. Springer-Verlag, 2004.

[17] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag.

[18] Gavin Lowe. A hierarchy of authentication specifications. In *CSFW*, pages 31–44. IEEE Computer Society, 1997.

[19] Simon Meier, Cas J. F. Cremers, and David A. Basin. Strong invariants for the efficient construction of machine-checked protocol security proofs. In *CSF*, pages 231–245. IEEE Computer Society, 2010.

[20] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[21] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[22] Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Trans. Inf. Syst. Secur.*, 2:332–351, August 1999.

A. Font and Variable Convention

Table A.1 gives an overview of our font conventions. To improve readability, we also restrict the use certain variable names to a specific type. Table A.2 gives our variable naming conventions.

Element	Font Style	Starting Character
function	italic	lowercase letter
sets	italic	uppercase letter
types	italic	uppercase letter
constructors	sans serif	uppercase letter
adversary capability	sans serif and italic	uppercase letter
INFERENCE/TRANSITION RULE	majuscule	uppercase letter
protocol	bold	uppercase letter
role	italic	uppercase letter

Table A.1.: Formating Conventions

Variable	Description
a,b	agents
l	label
m	messages
nt	note type
pt	pattern
P	protocol
R	role
st	role step
i,j	thread identifiers

Table A.2.: Variable Naming Conventions

B. List of new Concepts and Ideas

We extend the interactive construction method for machine-checked protocol security proofs of Meier, Cremers, and Basin [19] with compromising adversaries from Cremers and Basin [3]. Therefore many definitions and concepts are taken from them or are at least inspired by them. In this report, we accurately mark these concepts by citing the respective work. However, there are so many citations that it might be difficult to find our contributions. Therefore, we list our concepts, ideas, and proposals for the convenience of our readers.

For the security protocol model, we introduce the note types and the note role steps. Moreover, we define short-term data reveals and dynamic long-term key reveals in the transition system. Due to this work, we must also adopt other definitions such as the initial states or the thread pool. Note that the note role steps and the transition system changes are inspired by [3].

In the security properties chapter, we introduce the union type reveal to generalize reveals. We introduce adversary capabilities, which are inspired by adversary compromise rules from [3]. Furthermore, we demonstrate how we use adversary compromise models in our security properties. We propose the concept of a trust function and a method to specify partnering functions.

We introduce new inference rules and adopt rules from the decryption chain reasoning. Furthermore, we propose origin properties for adversary compromise model independent secrecy reasoning and give background information about reveal conditions. We propose a solution for using origin properties for the proof of other origin properties. Finally, we propose criteria for a proof heuristic.

We present interesting results we have gained from constructing protocol security proofs for our case studies.