Efficient Construction of Machine-Checked Symbolic Protocol Security Proofs

Simon Meier, Cas Cremers, David Basin
Institute of Information Security, ETH Zurich, Switzerland
Email: {simon.meier, cas.cremers, david.basin}@inf.ethz.ch

June 29, 2012

Abstract

We embed an untyped security protocol model in the interactive theorem prover Isabelle/HOL and derive a theory for constructing proofs of secrecy and authentication properties. Our theory is based on two key ingredients. The first is an inference rule for enumerating the possible origins of messages known to the intruder. The second is a class of protocol-specific invariants that formalize type assertions about variables in protocol specifications. The resulting theory is well-suited for interactively constructing human-readable, protocol security proofs. We additionally give an algorithm that automatically generates Isabelle/HOL proof scripts based on this theory. We provide case studies showing that both interactive and automatic proof construction are efficient. The resulting proofs provide strong correctness guarantees since all proofs, including those deriving our theory from the security protocol model, are machine-checked.

1 Introduction

Problem Context. Security protocols are standard components of systems that communicate over untrusted networks such as the Internet. Their relatively small size, combined with their critical role, makes them a suitable target for formal analysis. During the last twenty years, many successful symbolic methods have been developed for analyzing small to medium-sized protocols.

Ideally, when no attacks exist on a protocol, we would like to have a *proof* of the protocol's correctness. Such a proof provides an explanation of why the protocol is correct and makes the result verifiable for others. Given the complexity of the symbolic models used and the history of mistakes made in pen-and-paper proofs, it seems prudent to require *machine-checked proofs.*

In his seminal work [35], Paulson proposed the first approach for constructing machine-checked proofs of protocol correctness, using the interactive theorem prover Isabelle/HOL [34]. The protocols verified include the TLS handshake [36], Kerberos IV [8], and SET [7]. As reported in [35], the time required for an expert in interactive theorem proving to model and verify a protocol using this approach ranges from several days for small academic protocols to six weeks for a protocol like the TLS handshake [36].

An alternative approach to security protocol verification is to use automatic tools such as ProVerif [11] or Scyther [15]. Such tools have two substantial advantages over interactive approaches: they require less user expertise and produce results orders of magnitude faster. However none of these tools produces machine-checked proofs.

We combine the benefits of these two approaches to security protocol verification. We develop a tool-supported framework for the efficient construction of machine-checked protocol security proofs using automatic proof generation, where possible, and interactive proof construction, where required.

Approach Taken. Our approach is based on the model-checking algorithm underlying the Scyther tool. We formalize soundness theorems for all steps that this algorithm employs. The resulting set of theorems constitutes a verification theory for security protocols. As we will later see, this theory enables us to construct succinct security proofs by composing these soundness theorems. Based on this theory, we develop a proof-generating security protocol verification algorithm. Namely, we instrument an extended version of Scyther's model-checking algorithm so that it justifies each step it takes by logging the corresponding soundness theorem instance.

We state and prove security properties with respect to a symbolic security protocol model with a Dolev-Yao style, active intruder. Our protocol execution model is untyped, i.e., protocol variables may be instantiated with arbitrary messages. Hence, our security proofs also imply the absence of type-flaw attacks. We formalize our security protocol model as an operational semantics similar to the one proposed by Cremers and Mauw [16]. With respect to this operational semantics, we state and prove in Isabelle/HOL the soundness theorems constituting our security protocol verification theory. In our theory, security proofs are constructed from two key elements: the Chain rule and a class of invariants that formalize protocol-specific type assertions about the variables occurring in protocol specifications.

The Chain rule formalizes a finite and complete enumeration of the possible origins of messages known to the intruder. It is based on the observation that it suffices to check every possible chain of decryptions starting from a message sent by the protocol to determine if the intruder can learn a message by decryption. In such a chain, the input of each decryption, except the first one, is provided by the output of the preceding decryption.

We use the Chain rule to prove security properties as follows. We start by assuming that there exists an attack on the security property to be proven. Hence, the intruder must know certain messages. We make a case distinction on how the intruder learned these messages using the Chain rule. In each case, we gain additional assumptions about the structure of a possible attack. If these assumptions lead to a contradiction, then no attack exists. Otherwise, we again apply the Chain rule to iteratively refine the structure of possible attacks. Once we have shown that every case is contradictory, we have proven that the security property holds.

The second element of our theory is a class of invariants that formalize protocol-specific type assertions about the variables occurring in protocol specifications. They are required to make our proof strategy, based on the Chain rule, work in an untyped model. Our type assertions generalize the types naturally occurring in protocol specifications so that they also account for the cases where the intruder inserts his own messages. For example, suppose that a protocol specifies that a variable v is of type nonce. We then assert that v is always instantiated with either (1) a nonce or (2) some message that the intruder knew before v was instantiated. The first case accounts for receiving a message sent by another honest protocol participant. The second case accounts for receiving a message from the intruder, who need not follow any typing discipline. We use such type assertions together with the Chain rule to reason about what the intruder can learn from a variable's content. Note that we do not assume a-priori that a protocol satisfies its type assertions; this must be proven. We construct these proofs using a specialized induction scheme that we derive from our operational semantics.

Contributions. First, we develop a theory for constructing succinct proofs of secrecy and strong authentication properties in an untyped security protocol model without any bound on the number of parallel protocol executions. We demonstrate the wide applicability of our theory by constructing proofs for a number of case studies taken from literature.

Second, we formalize our theory in Isabelle/HOL and automate it such that it supports the efficient *interactive* construction of machine-checked protocol security proofs. Constructing proofs using our approach is almost two orders of magnitude faster than when using Paulson's Inductive Approach, i.e., hours instead of weeks for typical protocols.

Third, we develop an algorithm for *automatically* generating protocol security proofs that can be efficiently machine-checked by Isabelle. We implement this algorithm in a tool called `scyther-proof` [31]. This tool can be seen as a re-implementation of the Scyther tool [15], extending it with support for proof-generation and verification in an untyped protocol model. Compared to the other two existing approaches for the automatic generation of machine-checked protocol security proofs [13, 23], our approach is orders of magnitude faster than [13] and as fast as [23], but more expressive with respect to the security properties supported. Moreover, the Isabelle/HOL formalization of our theory and the proof-generation algorithm constitute the first framework that combines the benefits of manual, machine-checkable, proof construction with the efficiency of automatic protocol verification.

Finally, for both interactive and automated proofs, we have strong correctness guarantees. All theorems constituting our security protocol verification theory are *formally derived* from our operational semantics of security protocols in Isabelle/HOL. Hence, we have machine-checked proofs of their soundness. Furthermore, the protocol proofs themselves are machine-checked, both those interactively constructed and those automatically generated. Producing machine-checkable proofs is especially important when using complex (semi-)decision procedures whose correctness (both algorithmically and of the implementation) is non-trivial. For example, proof checking in Isabelle gives us a guarantee independent of any possible implementation errors in the `scyther-proof` tool.

Organization. In Section 2, we explain necessary background. In Section 3, we define the operational semantics underlying our verification theory. In Section 4, we derive and explain our verification theory. In Section 5, we automate proof construction in our theory. We also report on case studies in both interactive and automatic proof construction. We discuss related work in Section 6 and draw conclusions in Section 7.

2 Background

After some notational preliminaries, we explain the general approach we use to formalize our work in Isabelle/HOL.

2.1 Notational Preliminaries

For a set A, we denote its power set by $\mathcal{P}(A)$. We use standard notation for pairs and define the projections $\pi_i(x_1, x_2) = x_i$ for $i \in \{1, 2\}$. For a binary relation R, we denote its reflexive transitive closure by R^*.

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 the update of f at a, 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. We model partial functions in a logic of total functions by mapping all elements in $X \setminus \mathit{dom}(f)$ to the undefined value $\perp$, different from all other values.

For a set S, we write S^* to denote 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 $x \in s$ if and only if $\exists i.\, s_i = x$. 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 and only if s is duplicate-free.

To enhance readability, we identify sentences of the form

$$\forall x_1, \ldots, x_n.\, (A_1 \wedge \ldots \wedge A_m) \Rightarrow B$$

with inferences rules, written as

$$\frac{A_1 \qquad \cdots \qquad A_m}{B} \;,$$

where the free variables $x_1, \ldots, x_n$ are implicitly universally quantified.

2.2 Formalization in Isabelle/HOL

Isabelle is a generic, tactic-based theorem prover. We use Isabelle's implementation of higher-order logic, Isabelle/HOL [34], for our developments.

We formalize our protocol semantics as a conservative definitional extension of higher-order logic (HOL). This guarantees the consistency of our formalization, provided that HOL itself is consistent. Based on our semantics, we formalize several domain-specific predicates that serve as the basic building blocks for expressing security properties as HOL formulas. Such a formalization is sometimes called a *shallow embedding* [22] in the verification community, as we do not introduce a separate concrete syntax for formalizing security properties, but work directly with the formulas formalizing their semantics. The benefit of a shallow embedding is that we can reuse the existing reasoning infrastructure for HOL and only need to extend it to reason about our domain-specific predicates. We extend the existing infrastructure by deriving inference rules (HOL theorems) from our semantics that directly encode reasoning principles for constructing protocol security proofs.

Note that all definitions, rules, and theorems in this paper are justified by corresponding definitions and machine-checked (soundness) proofs in Isabelle/HOL. The corresponding proof scripts are distributed together with the `scyther-proof` tool [31].

3 Security Protocol Model

In this section, we define our security protocol model. It consists of three parts: (1) a *protocol specification language* based on role scripts (sequences of send and receive steps) and pattern matching, (2) an operational semantics defining *protocol execution* in the presence of an active intruder, and (3) a collection of predicates for formalizing *security properties* like secrecy and authentication.

3.1 Protocol Specification

We model security protocols as sets of roles where each role is given by a role script specified by a sequence of role steps. A role step sends or receives messages matching given message patterns. We first describe the elements of our specification language and then provide an example.

Let *Const* be a set of *constants*, *Fresh* be a set of messages to be freshly generated (nonces, coin flips, etc.), and *Var* be a set of *variables*. We assume that *Const*, *Fresh*, and *Var* are pairwise-disjoint. We further assume that the set of variables *Var* is partitioned into two sets, *AVar* and *MVar*, denoting *agent variables* and *message variables*. Agent variables are placeholders for agent names, which are chosen when creating a new role instance, and message variables are placeholders for messages (which may also be agent names) received during the execution of a role instance. We define the set *Pat* of *message patterns* as

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

We define $\mathit{vars}(pt)$ as the set of all variables in the pattern pt. The constructor $\mathsf{h}(\cdot)$ denotes hashing and $(\cdot,\cdot)$ denotes pairing. We use the pattern constructor $\{\!|\cdot|\!\}_\cdot$ to model the use of algorithms for public-key and symmetric encryption, private-key and symmetric decryption, signing, and signature verification. The semantics of this constructor is given in Section 3.2.3 by the *inst* function, which instantiates message patterns to their corresponding messages.

Informally, $\{\!|p|\!\}_k$ denotes public-key encryption of p when $k = \mathsf{pk}(pt)$, signing p when $k = \mathsf{sk}(pt)$, and symmetric encryption of p otherwise. Note that this allows for a composed message such as the hash of some keying material to be used as a symmetric key. For $a, b \in \mathit{AVar}$, the pattern $\mathsf{k}(a, b)$ denotes the long-term symmetric key shared between a and b, $\mathsf{pk}(a)$ denotes a's long-term public key, and $\mathsf{sk}(a)$ denotes a's long-term private key. We allow for freshly generated asymmetric keypairs by letting $n \in \mathit{Fresh}$ be the random seed, and using $\mathsf{pk}(n)$ and $\mathsf{sk}(n)$ to denote the corresponding freshly generated public and private keys. The pattern pt^{-1} denotes the inverse of the key denoted by the pattern pt.

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

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

A *send role-step* $\mathsf{Send}_l(pt)$ denotes sending the message corresponding to the instantiated pattern pt. A *receive role-step* $\mathsf{Recv}_l(pt)$ denotes receiving a message matching the pattern pt. The labels have no operational meaning: they serve just to distinguish different send (or receive) steps that contain the same message pattern. A *role* is a duplicate-free, finite sequence R of role steps such that

$$\begin{aligned} & \forall\, \mathsf{Send}_l(pt) \in R.\ \forall v \in \mathit{vars}(pt) \cap \mathit{MVar}. \\ & \quad \exists l',\, pt'.\ \mathsf{Recv}_{l'}(pt') <_R \mathsf{Send}_l(pt) \wedge v \in \mathit{vars}(pt') \ . \end{aligned}$$

This states that every message variable in a role must be instantiated in a receive step before its contents can be used in a send step. Note the we only restrict the use of message variables. Agent variables and freshly generated messages can be used freely in send steps. 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*. We illustrate protocol specifications with a simple challenge-response protocol.

Example 1 (*CR* Protocol)**.** Let $s \in \mathit{AVar}$, $k \in \mathit{Fresh}$, and $v \in \mathit{MVar}$. We define $\mathit{CR} \overset{\text{def}}{=} \{C, S\}$, where

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

In this protocol, a client, modeled by the C role, chooses a fresh session key k and sends it encrypted with the public key $\mathsf{pk}(s)$ of the server with whom he wants to share k. The server, modeled by the S role, confirms the receipt of k by returning its hash. We use this protocol as a running example. Hence, in subsequent examples, the expressions s, k, v, C, S, and CR refer to those introduced here.

3.2 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*. Threads may generate fresh messages, send messages to the network, and receive messages from the network as specified by the role script they execute. We assume that the network is completely controlled by an active Dolev-Yao style intruder. In particular, the intruder learns every message sent and can block and insert messages. Moreover, the intruder can access the long-term keys of arbitrarily many compromised agents.

We provide an operational semantics for protocol execution in the presence of the intruder, expressed as a state transition system, along the lines of [16]. The ingredients of the operational semantics are messages, the system state, agent threads, the intruder knowledge, and the transition system. We discuss each of these in turn.

3.2.1 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 $n \in \mathit{Fresh}$ 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 a set, we write $A\sharp \mathit{TID}$ to denote the set $\{a\sharp i \mid a \in A, i \in \mathit{TID}\}$.

We assume given a set Agent of agent names disjoint from Const. We define the set Msg of *messages*

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

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. Thus, depending on the key k, the message $\{\!|m|\!\}_k$ denotes the result of signing, public-key encryption, or symmetric encryption.

3.2.2 System State

The system state of our operational semantics is a triple (tr, th, σ). It consists of (1) a trace tr recording the history of the executed role steps and the events when messages are learned (i.e., become known for the first time) by the intruder,

(2) a thread pool *th* that stores for every thread the role it executes and the role steps still to be executed, and (3) a variable store σ recording the variable contents of all threads. We define these parts below.

The set of all *trace events* is defined as

$$\mathit{TraceEvent} ::= \mathsf{St}(\mathit{TID}, \mathit{RoleStep}) \mid \mathsf{Ln}(\mathcal{P}(\mathit{Msg}))\ .$$

A *step trace event* $\mathsf{St}(i, s)$ denotes that the thread i executed the role step s. A *learn trace event* $\mathsf{Ln}(M)$ denotes that the intruder learned the set of messages M. We use learn trace events to explicitly record the intruder message deduction steps in the trace. We will see why we use sets of messages rather than single messages later in the definition of our operational semantics. The *trace tr* is a sequence of *trace events*. The *thread pool th* is a partial function

$$\mathit{th} : \mathit{TID} \nrightarrow (\mathit{Role} \times \mathit{RoleStep}^*)\ ,$$

where $\mathit{dom}(\mathit{th})$ denotes the identifiers of all threads in the system. Where unambiguous, we identify threads with their corresponding thread identifiers. For $i \in \mathit{dom}(\mathit{th})$ and $\mathit{th}(i) = (R, \mathit{todo})$, R is the role executed by thread i and *todo* is a suffix of R denoting the role steps still to be executed by thread i. The *variable store* σ is a function

$$\sigma : \mathit{Var} \times \mathit{TID} \to \mathit{Msg}$$

that stores for each variable v and thread identifier i the contents $\sigma(v, i)$ assigned to v by thread i. We define *Trace* as the set of all traces, *ThreadPool* as the set of all thread pools, and *Store* as the set of all variable stores. Hence, a *system state* is a triple

$$(\mathit{tr}, \mathit{th}, \sigma) \in \mathit{Trace} \times \mathit{ThreadPool} \times \mathit{Store}\ .$$

Note that we explicitly record the history of the protocol execution and the intruder message deduction in the trace. As we will see later, this is crucial for formulating our invariants. It allows us, for example, to formulate the statement that if the intruder learns a message m by decrypting an encryption $\{\!|m|\!\}_k$, then he must have learned the inverse key k^{-1} before m.

3.2.3 Agent Threads

Let $(\mathit{tr}, \mathit{th}, \sigma)$ be a system state and $i \in \mathit{dom}(\mathit{th})$ be a thread in the system. The *role of thread* i is defined as $\mathit{role}_{\mathit{th}}(i) \overset{\text{def}}{=} \pi_1(\mathit{th}(i))$. Moreover, the thread i *instantiates* a message pattern *pt* occurring in its role to the message $\mathit{inst}_{\sigma,i}(\mathit{pt})$ as defined in Figure 1. During instantiation fresh message patterns are replaced with the actual fresh messages and all variables are replaced with the content assigned to them in thread i. Moreover, the key-inverse constructor on patterns is replaced with applications of the key-inverse function on messages. This is well-defined because messages are ground by definition.

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

Figure 1: Definition of the message pattern instantiation function $\mathit{inst}_{\sigma,i}(\cdot)$.

3.2.4 Intruder Knowledge

We assume that the set of all agents *Agent* is partitioned into a set *Compr* of compromised agents whose long-term keys are known to the intruder and a set of uncompromised agents. Thus, the intruder can impersonate any agent $c \in \mathit{Compr}$ and act as a legitimate protocol participant. The *initial intruder knowledge* IK_0 is therefore defined as

$$
\mathit{IK}_0 \overset{\text{def}}{=} \mathit{Const} \cup \mathit{Agent} \cup \bigcup_{a \in \mathit{Agent}, c \in \mathit{Compr}} \{\mathsf{pk}(a), \mathsf{sk}(c), \mathsf{k}(a, c), \mathsf{k}(c, a)\} \ .
$$

Note that we will define our operational semantics such that every trace starts with the learn trace event $\mathsf{Ln}(\mathit{IK}_0)$; i.e., the intruder always learns his complete initial knowledge before any other event happens. Our semantics also forces the intruder to explicitly perform all message deduction steps. Hence, a trace *tr* records all messages learned by the intruder. The *intruder knowledge* corresponding to *tr* is therefore

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

Our operational semantics also ensures that whenever the intruder learns a pair of messages (m_1, m_2), then he also learns its projections m_1 and m_2, provided he does not know them already. Therefore, the intruder knowledge is closed under the projection of pairs. This simplifies the reasoning in our security proofs as it allows keeping the construction and projection of pairs implicit. We use the function $\mathit{split} \colon \mathit{Msg} \to \mathcal{P}(\mathit{Msg})$ to formalize this closure.

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

We define the set of all messages *learned* by the intruder from a message m in the context of a trace tr as

$$learns_{tr}(m) \overset{\text{def}}{=} split(m) \setminus knows(tr) \ .$$

This states that the intruder learns all messages in $split(m)$ that he did not already know.

Example 2 (System state of the CR protocol)**.** Assume that some agent $a \in Agent$ executes the C role in thread $i \in TID$ and has sent his first message $\{\!|k\sharp i|\!\}_{\mathsf{pk}(b)}$ to establish the fresh session key $k\sharp i$ with an agent $b \in Agent$. Also, assume that agent b executed the $\mathsf{Recv}_1(\{\!|v|\!\}_{\mathsf{pk}(s)})$ step of the S role in the thread $j \in TID$ and received the first message that thread i sent. If i and j are the only threads running, then the system state is of the form (tr, th, σ), for

$$\begin{aligned} th &\overset{\text{def}}{=} \{i \mapsto (C,\, \langle C_2 \rangle),\, j \mapsto (S,\, \langle S_2 \rangle)\} \\ \sigma &\overset{\text{def}}{=} \sigma'[(s,i) \mapsto b,\, (s,j) \mapsto b,\, (v,j) \mapsto k\sharp i] \\ tr &\overset{\text{def}}{=} \langle \mathsf{Ln}(IK_0),\, \mathsf{St}(i, C_1),\, \mathsf{Ln}(\{\{\!|k\sharp i|\!\}_{\mathsf{pk}(b)}\}),\, \mathsf{St}(j, S_1) \rangle \end{aligned}$$

and some $\sigma' \in Store$.

3.2.5 Transition System

For a protocol P, the *state transition relation* $\longrightarrow$ is defined by the transition rules in Figure 2. We explain each rule in turn.

A Send transition is enabled whenever the next step of a thread i is $\mathsf{Send}_l(pt)$ for some label l and message pattern pt. The trace tr is extended with two trace events. The trace event $\mathsf{St}(i, \mathsf{Send}_l(pt))$ records that the send step has happened. The trace event $\mathsf{Ln}(learns_{tr}(inst_{\sigma,i}(pt)))$ records the set of messages learned by the intruder from the sent message $inst_{\sigma,i}(pt)$. We see here the benefit of being able to record that the intruder learns a set of messages at once. Namely, we can close the intruder knowledge under projection of pairs without explicitly ordering the events denoting the learning of the pairs' components. Note that $\mathsf{Ln}(learns_{tr}(inst_{\sigma,i}(pt)))$ is equal to $\mathsf{Ln}(\emptyset)$ if and only if the intruder has already learned the sent message $inst_{\sigma,i}(pt)$ before this Send transition.

A Recv transition is enabled whenever the next step of a thread i is $\mathsf{Recv}_l(pt)$, for some label l and some message pattern pt, and the intruder knows a message matching pt under the variable store σ. The trace tr is extended with the trace event $\mathsf{St}(i, \mathsf{Recv}_l(pt))$, recording that this receive step has happened.

A Pair, Hash, or Encr transition models the intruder learning respectively a pair, a hash, or an encryption by constructing it by himself. He can do this provided he does not yet know the constructed message. Because the intruder knowledge is closed under *split*, no projection transition is needed.

A Decr transition models the decryption of an encrypted message with the decryption key and learning all new messages accessible from the encrypted message using projection of pairs.

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

$$\frac{th(i) = (R, \langle \mathsf{Recv}_l(pt)\rangle \mathbin{\hat{}} todo) \qquad inst_{\sigma,i}(pt) \in knows(tr)}{(tr, th, \sigma) \longrightarrow (tr \mathbin{\hat{}} \langle\, \mathsf{St}(i, \mathsf{Recv}_l(pt))\,\rangle,\ th[i \mapsto (R, todo)],\ \sigma)} \ \textsc{Recv}$$

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

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

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

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

Figure 2: Transition rules of the execution model.

There is no explicit transition rule for creating new threads. Instead, for each thread pool that can occur in an execution, we construct a separate initial state. Thus all possible thread pools with threads executing roles of the protocol are represented in the set of *initial states* $Q_0(P)$ of our system.

$$\begin{array}{rl} Q_0(P) \stackrel{\text{def}}{=} \{\ (\langle \mathsf{Ln}(IK_0)\rangle,\ th,\ \sigma) \mid & (\forall v \in AVar,\ i \in TID.\ \sigma(v, i) \in Agent) \wedge \\ & (\forall i \in dom(th).\ \exists R \in P.\ th(i) = (R, R))\ \} \end{array}$$

For each initial state $(tr, th, \sigma) \in Q_0(P)$, the variable store σ is defined such that every agent variable is instantiated with an agent name and each message variable is instantiated with an arbitrary message. Thus, we model the set of possible executions by instantiating all variables non-deterministically at the beginning of a thread. The thread pool th is defined such that every thread $i \in dom(th)$ instantiates a role of P and has not executed any step yet.

For a protocol P, we define the set of *reachable states* as

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

3.3 Security Properties

We focus on security properties that are state properties. We say that a security protocol P *satisfies* a security property ϕ if and only if ϕ holds for every reachable state of P. Because we include the execution trace in the system state, many security properties from literature can be expressed as state properties, for example, all the authentication properties from [17, 29] and secrecy as in [16].

Note that when reasoning about security protocols and their properties, we are interested in when individual messages were learned and not in what messages were learned at the same time. Therefore, we introduce the set of *(proper) events.*

$$\mathit{Event} ::= \mathsf{ST}(\mathit{TID} \times \mathit{RoleStep}) \mid \mathsf{LN}(\mathit{Msg})$$

Analogous to step trace events, a *step event* $\mathsf{ST}(i, s)$ denotes that thread i has executed the role step s. In contrast to learn trace events, a *learn event* $\mathsf{LN}(m)$ denotes that the intruder learned the *single* message m.

In the remainder of this paper, we simplify our notation and definitions by identifying every tuple $(i, s) \in \mathit{TID} \times \mathit{RoleStep}$ with the step event $\mathsf{ST}(i, s)$ and every message $m \in \mathit{Msg}$ with the learn event $\mathsf{LN}(m)$. The previously defined function $\mathit{knows} : \mathit{Trace} \to \mathcal{P}(\mathit{Msg})$ can therefore be used to project a trace tr to the set of all learn events occurring in tr.

The projection of a trace tr to the set of all step events occurring in tr is

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

The *event order* relation $(\prec_{\mathit{tr}}) \subseteq \mathit{Event} \times \mathit{Event}$ denotes the order of events induced by the trace events in tr.

$$\begin{aligned} x \prec_{\mathit{tr}} y \overset{\text{def}}{=} \exists \mathit{tr}_1\ \mathit{tr}_2.\ \mathit{tr} = \mathit{tr}_1 \mathbin{\hat{}} \mathit{tr}_2 &\wedge (x \in \mathit{knows}(\mathit{tr}_1) \vee x \in \mathit{steps}(\mathit{tr}_1)) \\ &\wedge (y \in \mathit{knows}(\mathit{tr}_2) \vee y \in \mathit{steps}(\mathit{tr}_2)) \end{aligned}$$

Note that $\prec_{\mathit{tr}}$ is a strict partial order on *Event* for every trace tr of a reachable state $(\mathit{tr}, \mathit{th}, \sigma) \in \mathit{reachable}(P)$. We define $\preceq_{\mathit{tr}}$ as the reflexive closure of $\prec_{\mathit{tr}}$.

The event order allows us, for example, to formalize the statement "both the encryption $\{\!|m|\!\}_k$ and the inverse key k^{-1} must have been learned before the intruder learned m" as the proposition

$$\{\!|m|\!\}_k \prec_{\mathit{tr}} m \ \wedge \ k^{-1} \prec_{\mathit{tr}} m \ .$$

Note that the event order also relates learn events with protocol step events. We exploit this, for example, when stating and verifying temporal secrecy properties, i.e., properties stating that a message is secret as long as a certain protocol step, which leaks this message, has not been executed.

Security properties are formalized as logical formulas built using the previously defined functions and relations. We illustrate this in the following example.

Example 3 (Security properties of the *CR* protocol)**.** For a client who completes its role with an uncompromised server, the *CR* protocol guarantees (1) the secrecy of the session key k and (2) non-injective synchronization [17] (a strengthened variant of non-injective agreement [29]) with the server. We formalize property (1) by the formula ϕ_{sec}.

$$\begin{aligned} &\phi_{\text{sec}}(\mathit{tr}, \mathit{th}, \sigma) \overset{\text{def}}{=} \forall i \in \mathit{TID}. \\ &\qquad \mathit{role}_{\mathit{th}}(i) = C \wedge \sigma(s, i) \notin \mathit{Compr} \Rightarrow k\sharp i \notin \mathit{knows}(\mathit{tr}) \end{aligned}$$

Recall that C, s, and k, as well as S and v, were defined in Example 1.

Intuitively, property (2) states that whenever a client thread i communicates with an uncompromised server and receives its last message, then there exists a server thread that received the first message from client i and whose second message was received by client i. We formalize this by the formula ϕ_{auth}.

$$\begin{aligned}
&\phi_{\text{auth}}(tr, th, \sigma) \overset{\text{def}}{=} \forall i \in \mathit{TID}.\\
&\quad \mathit{role}_{th}(i) = C \wedge \sigma(s, i) \notin \mathit{Compr} \wedge (i, C_2) \in \mathit{steps}(tr)\\
&\quad \Rightarrow (\exists j \in \mathit{TID}.\ \mathit{role}_{th}(j) = S \wedge \sigma(s, i) = \sigma(s, j) \wedge k\sharp i = \sigma(v, j) \wedge\\
&\qquad\qquad\qquad\qquad (i, C_1) \prec_{tr} (j, S_1) \wedge (j, S_2) \prec_{tr} (i, C_2))
\end{aligned}$$

Recall that a role is a sequence of role-steps. Hence, C_2 denotes the second step of the C role.

4 Security Proofs Based on Decryption Chains

We now present our theory for proving security properties with respect to the semantics described in Section 3. As we already stated in the introduction, our theory consists of two key elements: the Chain rule and invariants constructed from protocol-specific type assertions.

We present the Chain rule together with the rest of the core inference rules in Section 4.1. We explain the intuition behind the Chain rule in Section 4.2 by describing our strategy for proving secrecy and authentication properties and by illustrating this strategy on the security properties of the *CR* protocol given in Example 3. Note that the *CR* protocol is one of the few protocols where our proof strategy works without auxiliary type assertions. The reasons for this will become clear when we explain our use of type assertions in Section 4.3. We conclude with a discussion of our proof construction method in Section 4.4.

4.1 Core Inference Rules

Our core inference rules are given in Figure 3. We have formally derived all of these rules from our operational semantics under the assumption that $(tr, th, \sigma) \in \mathit{reachable}(P)$.

The rules Kn_1 and Kn_2 state that if the intruder knows a pair of messages (m_1, m_2), then he also knows m_1 and m_2. Similarly, the rules Ord_1 and Ord_2 state that if a pair of messages (m_1, m_2) was learned before the event e happened, then both m_1 and m_2 were also learned before e happened. These four rules allow us to reduce statements about the knowledge of pairs to the knowledge of the contained nonces, hashes, and encryptions. These rules are sound because the intruder knowledge is closed under the projection of pairs.

The rules Known and Exec follow trivially from the definitions of $\prec_{tr}$, *knows*, and *steps*. They formalize the intuition that, if an event e happened before some other event, then e has happened.

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

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

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

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

$$\frac{\mathit{role}_{th}(i) = R \qquad s' <_R s \qquad (i, s) \in \mathit{steps}(tr)}{(i, s') \prec_{tr} (i, s)}\ \textsc{Role}$$

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

$$\frac{m \in \mathit{knows}(tr)}{\begin{array}{l} (m \in IK_0) \lor \\ (\exists x.\quad m = \mathsf{h}(x) \quad \land x \prec_{tr} \mathsf{h}(x) \qquad\qquad\qquad) \lor \\ (\exists x\ k.\ \ m = \{\!|x|\!\}_k \ \land x \prec_{tr} \{\!|x|\!\}_k \ \land k \prec_{tr} \{\!|x|\!\}_k \quad) \lor \\ (\exists x\ y.\ \ m = (x, y) \ \land x \prec_{tr} (x, y) \ \land y \prec_{tr} (x, y) \quad) \lor \\ (\exists R \in P.\ \exists\, \mathsf{Send}_l(pt) \in R.\ \exists i.\ \mathit{role}_{th}(i) = R \land \\ \qquad\qquad\qquad\qquad \mathit{chain}_{tr}(\{(i, \mathsf{Send}_l(pt))\},\ \mathit{inst}_{\sigma,i}(pt),\ m)\ ) \end{array}}\ \textsc{Chain}$$

Figure 3: Core inference rules for decryption-chain reasoning, which hold under the assumption that $(tr, th, \sigma) \in \mathit{reachable}(P)$.

$$\begin{array}{l} \mathit{chain}_{tr}(E,\, m',\, m) \stackrel{\text{def}}{=} \\ \quad (\qquad m' = m \qquad \land (\forall e \in E.\ e \prec_{tr} m \qquad) \qquad\qquad\qquad\qquad\qquad\qquad) \lor \\ \quad (\exists x\, k.\ m' = \{\!|x|\!\}_k \land (\forall e \in E.\ e \prec_{tr} \{\!|x|\!\}_k) \land \mathit{chain}_{tr}(\{\{\!|x|\!\}_k, k^{-1}\},\, x,\, m) \qquad) \lor \\ \quad (\exists x\, y.\ m' = (x, y) \qquad\qquad\qquad\qquad \land (\mathit{chain}_{tr}(E,\, x,\, m) \lor \mathit{chain}_{tr}(E,\, y,\, m))) \end{array}$$

Figure 4: Definition of the *chain* predicate using recursion over the message m'.

The rules Irr and Trans formalize that $\prec_{tr}$ is a strict partial order. These rules are sound because roles are duplicate-free and our execution model therefore guarantees that all executed steps are unique and the intruder never learns the same message twice.

The rule Role states that if a thread i that is an instance of role R has executed role step s, then all the role steps $s' <_R s$ have been executed before s by the thread i. This rule is sound because both the Send and the Recv transitions execute role steps in the order specified by the roles.

The rule Input states that if a thread i has executed a receive step $\mathsf{Recv}_l(pt)$, then the instantiated pattern pt was learned before $\mathsf{Recv}_l(pt)$ was executed by the thread i. This rule is sound because the Recv transition ensures that the intruder knows the received message.

The rule Chain states that there are precisely five ways that an intruder can learn a message m.

1. He knew m from the start.
2. m is a hash $\mathsf{h}(x)$ of the known message x and the intruder built $\mathsf{h}(x)$ himself using the Hash transition.
3. m is an encryption $\{\!|x|\!\}_k$ of a known message x with a known key k and the intruder built $\{\!|x|\!\}_k$ himself using the Encr transition.
4. m is a pair (x, y) of two known messages x and y and the intruder built (x, y) himself using the Pair transition.
5. There was some step $\mathsf{Send}_l(pt)$ executed by some thread i such that the intruder learned the sent message $inst_{\sigma,i}(pt)$ and from this message he learned m using zero or more decryptions and projections.

We prove the soundness of this case distinction by induction over the reachable states $(tr, th, \sigma) \in reachable(P)$. The key idea behind Case (5), called the *decryption-chain case*, is that the intruder can only learn a message by decrypting an encryption that *he did not build himself*. Analogously, the intruder can only learn a message by projecting a pair that he did not build himself. Thus, whenever the intruder learns a message m by decrypting an encryption $\{\!|x|\!\}_k$, then he must have learned $\{\!|x|\!\}_k$ from a send step or by decrypting an encryption or projecting a pair containing $\{\!|x|\!\}_k$. As every message is of finite size, any such chain of repeated decryptions and projections is of finite length.

We formalize the notion of *decryption chains* using the *chain* predicate defined in Figure 4. For a set $E \subseteq Event$, the expression $chain_{tr}(E, m', m)$ formalizes that the intruder learned the message m using zero or more decryptions and projections after he learned some message in $split(m')$, which he learned after the events in E happened. The definition of *chain* distinguishes between three cases.

1. m' is the message m and the intruder learned m' after the events in E, or
2. m' is an encryption $\{\!|x|\!\}_k$ and the intruder learned m after he used the inverse key k^{-1} to decrypt $m' = \{\!|x|\!\}_k$, which he learned after the events in E, or

3. m' is a tuple (x, y) and the intruder learned m from a chain starting from x or from a chain starting from y. The set E is unchanged in this case because, in our protocol semantics, the messages x and y are learned at the same time or before the tuple (x, y).

4.2 Proof Strategy

Suppose we want to prove that a protocol P satisfies a security property ϕ. Our strategy for proving this consists of two main steps. First, we simplify the use of the Chain rule by instantiating it for the protocol P. This entails specializing the Chain rule for different outermost message constructors in its premise, enumerating P's roles and their send steps in the conclusion, and completely unfolding all occurrences of the *chain* predicate. Second, we prove that ϕ holds for every reachable state of the protocol P by repeatedly applying the simplified Chain rule to the messages that the intruder is supposed to know (e.g., received messages). Combined with straightforward logical reasoning formalized in HOL, this suffices to complete the proof in many cases.

The first step is completely mechanical. It allows us to share work between different security proofs of the same protocol. Moreover, it yields a compact description of the intruder's message derivation capabilities in the context of a given security protocol. We illustrate this first step on the *CR* protocol.

Example 4. Figure 5 shows the simplified instances of the Chain rule for the *CR* protocol. These rules capture the intruder's message derivation capabilities in the context of the *CR* protocol. For example, the *CR* protocol does not send private keys and long-term symmetric keys in an accessible position. Intuitively, the intruder can therefore learn such keys only from his initial knowledge. The rules SKChain_{CR} and KChain_{CR}, which were derived mechanically from the Chain rule, justify this intuition. Analogously, the rule NChain_{CR} shows that there is exactly one way for the intruder to learn a nonce $n\sharp i$. This nonce must be the freshly generated key $k\sharp i$ that the client thread i sent encrypted in its first step.

In the second step of our proof strategy, the Chain rule is used to prove the security property of interest, ϕ. In general, there will be multiple premises of the form $m \in \mathit{knows}(\mathit{tr})$ to which the Chain rule can be applied. Hence, a choice must be made. A rule application cannot render a previously provable security property unprovable. However, unnecessary applications of the Chain rule would needlessly increase the size of the resulting proof. To find short, direct proofs, we use the following strategy to prove both secrecy and authentication properties.

To prove *secrecy properties*, we use the Chain rule both for the message m to be proven secret as well as for the keys that must be kept secret if m is not to be decrypted. If the secrecy of some message depends on its authenticity (e.g., a key that is received), we use the proof strategy for authentication properties outlined below.

To prove *authentication properties*, we use the Chain rule on the received message m to justify why its receipt implies the existence of a partner thread

$$\frac{\mathsf{sk}(a) \in \mathit{knows}(tr)}{\mathsf{sk}(a) \in \mathit{IK}_0}\ \text{SKCHAIN}_{CR} \qquad \frac{\mathsf{k}(a,b) \in \mathit{knows}(tr)}{\mathsf{k}(a,b) \in \mathit{IK}_0}\ \text{KCHAIN}_{CR}$$

$$\frac{n\sharp i \in \mathit{knows}(tr)}{\mathit{role}_{th}(i) = C \ \wedge\ (i, C_1) \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}(\sigma(s,i))} \prec_{tr} k\sharp i}\ \text{NCHAIN}_{CR}$$
$$\wedge\ \mathsf{sk}(\sigma(s,i)) \prec_{tr} k\sharp i \ \wedge\ k = n$$

$$\frac{\mathsf{h}(x) \in \mathit{knows}(tr)}{\begin{array}{l}(x \prec_{tr} \mathsf{h}(x)) \vee \\ (\exists j.\ \mathit{role}_{th}(j) = S \wedge (j, S_2) \prec_{tr} \mathsf{h}(\sigma(v,j)) = \mathsf{h}(x))\end{array}}\ \text{HCHAIN}_{CR}$$

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

Figure 5: Simplified instances of the CHAIN rule for the *CR* protocol.

that sent m. If the authenticity of a message depends on the secrecy of another message (e.g., the key used for a MAC), then we use the strategy for secrecy properties outlined above.

We also factor out repeated subproofs, such as proofs about secrecy properties. We do this by introducing additional lemmas that generalize the properties proven by the repeated subproofs.

Note that we may switch multiple times between proving secrecy properties and proving authentication properties. In general, this does not result in cyclic dependencies between proofs because these proofs concern different messages. Some cases where decryption-chain reasoning fails could however be interpreted as a cyclic dependency of a proof on itself. We discuss this later in Section 4.4.

We illustrate the second step of our proof strategy on the two security properties of the *CR* protocol described in Example 3. We first prove a secrecy property. Afterwards, we prove an authentication property and show how we can use the already proven secrecy property to prove two secrecy subproofs.

Example 5 (Proof of session-key secrecy)**.** We prove $\forall q \in \mathit{reachable}(CR).\ \phi_{\text{sec}}(q)$, for ϕ_{sec} from Example 3.

Proof. Suppose the secrecy property ϕ_{sec} does not hold for some state $(tr, th, \sigma) \in \mathit{reachable}(CR)$. Then there is a thread i such that $\mathit{role}_{th}(i) = C$, $\sigma(s,i) \notin \mathit{Compr}$, and $k\sharp i \in \mathit{knows}(tr)$. Hence, the intruder must have learned $k\sharp i$.

There is only one premise, $k\sharp i \in \mathit{knows}(tr)$, to which we can apply the CHAIN rule. Instead of applying the CHAIN rule directly, we apply the corresponding simplified instance, NCHAIN_{CR}. From its conclusion, we see that the intruder can learn $k\sharp i$ only by decrypting the message $\{\!|k\sharp i|\!\}_{\mathsf{pk}(\sigma(s,i))}$, which implies that $\mathsf{sk}(\sigma(s,i)) \prec_{tr} k\sharp i$.

Using KNOWN, we have that $\mathsf{sk}(\sigma(s,i)) \in \mathit{knows}(tr)$ and, from SKCHAIN$_{CR}$, we conclude $\mathsf{sk}(\sigma(s,i)) \in \mathit{IK}_0$. Given the definition of IK_0, we have $\sigma(s,i) \in \mathit{Compr}$, which contradicts our assumptions.

We therefore conclude that ϕ_{sec} holds for all reachable states of the CR protocol. □

Example 6 (Proof of non-injective synchronization)**.** We prove that $\forall q \in \mathit{reachable}(CR).\ \phi_{\mathrm{auth}}(q)$, where ϕ_{auth} is defined in Example 3.

Proof. We must show that for every state $(tr, th, \sigma) \in \mathit{reachable}(CR)$ and every thread i such that $\mathit{role}_{th}(i) = C$, $\sigma(s,i) \notin \mathit{Compr}$, and $(i, C_2) \in \mathit{steps}(tr)$, there is a thread j such that *syncWith(j)* holds.

$$\begin{aligned} \mathit{syncWith}(j) \overset{\mathrm{def}}{=} \mathit{role}_{th}(j) = S \wedge \sigma(s,i) = \sigma(s,j) \wedge k\sharp i = \sigma(v,j) \\ \wedge (i, C_1) \prec_{tr} (j, S_1) \wedge (j, S_2) \prec_{tr} (i, C_2) \end{aligned}$$

We prove this by applying the CHAIN rule to the received messages.

From $(i, C_2) \in tr$, we have that $\mathsf{h}(k\sharp i) \prec_{tr} (i, C_2)$ using the INPUT rule and $\mathsf{h}(k\sharp i) \in \mathit{knows}(tr)$ using the KNOWN rule. Applying the HCHAIN$_{CR}$ rule yields the following conclusion, whose disjuncts we have numbered.

$$\begin{array}{ll} \textbf{(1)} & (k\sharp i \prec_{tr} \mathsf{h}(k\sharp i)) \\ \textbf{(2)} & \vee\ (\exists j \in \mathit{TID}.\ \mathit{role}_{th}(j) = S \wedge (j, S_2) \prec_{tr} \mathsf{h}(\sigma(v,j)) \wedge \mathsf{h}(\sigma(v,j)) = \mathsf{h}(k\sharp i)\ )\ . \end{array}$$

Case **(1)** is where the intruder builds the received message by himself. Using the KNOWN rule, we have that $k\sharp i \in \mathit{knows}(tr)$. This contradicts the secrecy property we proved in Example 5.

Case **(2)** implies that there is a server thread j, $\mathit{role}_{th}(j) = S$, that sent the message that the client thread i received. We show that client i synchronizes with server j.

From $\mathsf{h}(k\sharp i) = \mathsf{h}(\sigma(v,j))$ and the injectivity of $\mathsf{h}(\cdot)$, it follows that $k\sharp i = \sigma(v,j)$. From $(j, S_2) \prec_{tr} \mathsf{h}(\sigma(v,j))$, $\mathsf{h}(\sigma(v,j)) = \mathsf{h}(k\sharp i)$, and $\mathsf{h}(k\sharp i) \prec_{tr} (i, C_2)$, it follows that $(j, S_2) \prec_{tr} (i, C_2)$. To establish *syncWith(j)*, it remains to be shown that the first message of client i was received in the first step of server j; i.e., $\sigma(s,i) = \sigma(s,j)$ and $(i, C_1) \prec_{tr} (j, S_1)$.

From $(j, S_2) \prec_{tr} (i, C_2)$, we have $(j, S_1) \prec_{tr} (j, S_2)$ using the rules EXEC and ROLE. Hence, $\{\!|k\sharp i|\!\}_{\mathsf{pk}(\sigma(s,j))} \prec_{tr} (j, S_1)$ using the rules EXEC and INPUT and the fact $k\sharp i = \sigma(v,j)$. From the KNOWN rule, we have $\{\!|k\sharp i|\!\}_{\mathsf{pk}(\sigma(s,j))} \in \mathit{knows}(tr)$. Applying the ECHAIN$_{CR}$ rule yields

$$\begin{array}{ll} \textbf{(2.1)} & (\ k\sharp i \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}(\sigma(s,j))} \wedge \mathsf{pk}(\sigma(s,j)) \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}(\sigma(s,j))}\ ) \\ \textbf{(2.2)} & \vee\ (\ \exists i'.\ \mathit{role}_{th}(i') = C \wedge (i', C_1) \prec_{tr} \{\!|k\sharp i'|\!\}_{\mathsf{pk}(\sigma(s,i'))} \wedge \\ & \qquad \{\!|k\sharp i'|\!\}_{\mathsf{pk}(\sigma(s,i'))} = \{\!|k\sharp i|\!\}_{\mathsf{pk}(\sigma(s,j))}\ )\ . \end{array}$$

Case **(2.1)** states that the intruder fakes the message, which again contradicts the secrecy property proven in Example 5 due to $k\sharp i \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}(\sigma(s,j))}$ and the rule Known.

Case **(2.2)** implies $i' = i$ since $k\sharp i' = k\sharp i$. Hence, we have

$$(i, C_1) \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}(\sigma(s,i))} = \{\!|k\sharp i|\!\}_{\mathsf{pk}(\sigma(s,j))} \prec_{tr} (j, S_1) \ .$$

This implies that $\sigma(s,i) = \sigma(s,j)$ and $(i, C_1) \prec_{tr} (j, S_1)$, which concludes the proof. $\square$

4.3 Type Assertions

The proof strategy outlined in the previous subsection relies on our ability to instantiate the Chain rule and *completely* unfold its conclusion. This is straightforward in our *CR* example because it does not have a send step with a variable in an *accessible position*, i.e., a position that is neither below a hash nor below an encryption-key position.

However, most protocols do have send steps with variables in accessible positions and this results in expressions of the form $\mathit{chain}_{tr}(E,\, \sigma(v,i),\, m)$ in the conclusions of the simplified Chain rule instances. In general, $\sigma(v,i)$ can be an arbitrary message; hence, there *may* be a decryption chain starting from $\sigma(v,i)$ and resulting in m. However, for a concrete protocol, the assumption that $\sigma(v,i)$ is an arbitrary message is too pessimistic because the set of possible instantiations of protocol variables is restricted by both the protocol specification and the operational semantics. What we are missing to completely unfold these $\mathit{chain}_{tr}(E,\, \sigma(v,i),\, m)$ expressions is a formalization of these restrictions. We capture these using invariants that are constructed from protocol-specific type assertions.

We explain our use of type assertions in three parts. First, we show how we formalize type assertions. Then, we show how we use them to completely unfold the conclusion of Chain rule instances. Finally, we give a theorem for proving the soundness of type assertions using decryption-chain reasoning.

4.3.1 Syntax and Semantics of Type Assertions

A *type* is a term constructed according to the following grammar.

$$\begin{aligned} \mathit{Type} ::= \ & \mathit{Const} \mid \mathsf{Ag} \mid \mathit{Role.Fresh} \mid (\mathit{Type}, \mathit{Type}) \mid \mathsf{h}(\mathit{Type}) \mid \{\!|\mathit{Type}|\!\}_{\mathit{Type}} \\ & \mid \mathsf{k}(\mathit{Type}, \mathit{Type}) \mid \mathsf{pk}(\mathit{Type}) \mid \mathsf{sk}(\mathit{Type}) \mid \mathit{Type} \cup \mathit{Type} \mid \mathsf{kn}(\mathit{RoleStep}) \end{aligned}$$

Intuitively, types denote sets of messages. For every global constant $\gamma \in \mathit{Const}$, the type γ denotes the set $\{\gamma\}$. The type Ag denotes the set of all agent names. The type $R.f$ denotes all fresh messages named f that were generated by some thread executing role R. The type constructors $(\cdot,\cdot)$, $\mathsf{h}(\cdot)$, $\{\!|\cdot|\!\}_{\cdot}$, $\mathsf{k}(\cdot)$, $\mathsf{pk}(\cdot)$, and $\mathsf{sk}(\cdot)$ reflect message constructors to the type level. The type $\alpha \cup \beta$ denotes the sum of the two types α and β. We use type-sums to type variables that are instantiated with messages of different types.

The denotation of a type $\mathsf{kn}(s)$ is context-dependent. It depends on the system state and the thread i whose variables' instantiation we are specifying. The denotation of $\mathsf{kn}(s)$ is the set of all messages known to the intruder before the role step s was executed by the thread i. We use $\mathsf{kn}(s)$ types to account for the interaction with the active intruder, as we illustrate in the following example.

Example 7 (Type assertion)**.** The variable v of the server role S of the *CR* protocol is always instantiated with messages of type $C.k \cup \mathsf{kn}(S_1)$.

Note that the type $C.k$ does not cover all possible instantiations of v. It covers all instantiations that occur when receiving messages sent by client threads. The type-sum with $\mathsf{kn}(S_1)$ also covers those instantiations that are the result of receiving messages constructed by the intruder. For such an instantiation σ, the variable v of a thread i executing the S role is obviously not guaranteed to be instantiated with some message $k\sharp j$ that was freshly generated by some thread j executing the C role. However, it is guaranteed that the intruder knew $\sigma(v,i)$ before (i, S_1) was executed, as he constructed the received message himself.

In general, we construct the type of a variable v as follows. We use type constructors other than $\mathsf{kn}(\cdot)$ to specify the shape of v's instantiations which result from receiving messages sent by other threads that execute protocol roles. We use the $\mathsf{kn}(s)$ type to account for message parts that could be injected by the intruder, where the role step s is the step where v is instantiated. This construction works in all our case studies. Moreover, we can even automatically infer the types of almost all variables in our case studies using a simple heuristic.

Formally, the meaning of a type is given by the *type interpretation function* $[\![\cdot]\!]$, which associates to every type γ the set of messages $[\![\gamma]\!]^i_q$ denoted by the type γ in the context of a thread $i \in \mathit{TID}$ and a system state $q = (tr, th, \sigma)$.

$$[\![\gamma]\!]^i_q \overset{\text{def}}{=} \begin{cases} \{\gamma\} & \text{if } \gamma \in \mathit{Const} \\ \mathit{Agent} & \text{if } \gamma = \mathsf{Ag} \\ \{n\sharp j \mid \mathit{role}_{th}(j) = R\} & \text{if } \gamma = R.n \\ \{\mathsf{h}(x) \mid x \in [\![\alpha]\!]^i_q\} & \text{if } \gamma = \mathsf{h}(\alpha) \\ \{(x,y) \mid x \in [\![\alpha]\!]^i_q \wedge y \in [\![\beta]\!]^i_q\} & \text{if } \gamma = (\alpha, \beta) \\ \{\{\!|x|\!\}_k \mid x \in [\![\alpha]\!]^i_q \wedge k \in [\![\beta]\!]^i_q\} & \text{if } \gamma = \{\!|\alpha|\!\}_\beta \\ \{\mathsf{k}(x,y) \mid x \in [\![\alpha]\!]^i_q \wedge y \in [\![\beta]\!]^i_q\} & \text{if } \gamma = \mathsf{k}(\alpha, \beta) \\ \{\mathsf{pk}(x) \mid x \in [\![\alpha]\!]^i_q\} & \text{if } \gamma = \mathsf{pk}(\alpha) \\ \{\mathsf{sk}(x) \mid x \in [\![\alpha]\!]^i_q\} & \text{if } \gamma = \mathsf{sk}(\alpha) \\ [\![\alpha]\!]^i_q \cup [\![\beta]\!]^i_q & \text{if } \gamma = \alpha \cup \beta \\ \{m \mid m \prec_{tr} (i,s)\} & \text{if } \gamma = \mathsf{kn}(s) \end{cases}$$

We specify all type assertions for a single protocol together as a *typing*, which is a partial function $ty : (\mathit{Role} \times \mathit{Var}) \nrightarrow \mathit{Type}$. A state is *well-typed* with respect to a typing ty if and only if every variable is instantiated with a message denoted

$$\frac{\mathsf{h}(x) \in \mathit{knows}(tr)}{\begin{array}{l} \quad (x \prec_{tr} \mathsf{h}(x)\,) \\ \vee\, (\exists j.\ \mathit{role}_{th}(i) = C' \wedge (j, C'_1) \prec_{tr} \{\!|\sigma(c,j), k\sharp j|\!\}_{\mathsf{pk}(\sigma(s,j))} \\ \qquad\qquad\qquad\qquad\quad \wedge\ \mathit{chain}_{tr}(\{\{\!|\sigma(c,j), k\sharp j|\!\}_{\mathsf{pk}(\sigma(s,j))}, \mathsf{sk}(\sigma(s,j))\},\ \sigma(c,j),\ \mathsf{h}(x))\,) \\ \vee\, (\exists j.\ \mathit{role}_{th}(i) = S' \wedge (j, S'_2) \prec_{tr} \{\!|\sigma(v,j)|\!\}_{\mathsf{pk}(\sigma(c,j))} \\ \qquad\qquad\qquad\qquad\quad \wedge\ \mathit{chain}_{tr}(\{\{\!|\sigma(v,j)|\!\}_{\mathsf{pk}(\sigma(c,j))}, \mathsf{sk}(\sigma(c,j))\},\ \sigma(v,j),\ \mathsf{h}(x)) \qquad) \end{array}}\ \textsc{HChain}_{CR'}$$

Figure 6: Incomplete unfolding of the $\textsc{HChain}_{CR'}$ rule.

by its type; that is

$$\begin{aligned} \mathit{well\text{-}typed}_{ty}(tr, th, \sigma) \overset{\text{def}}{=} \forall (i,s) \in \mathit{steps}(tr).\ \forall v \in \mathit{vars}(s). \\ \sigma(v,i) \in [\![\mathit{ty}(\mathit{role}_{th}(i), v)]\!]^{i}_{(tr,th,\sigma)}\ . \end{aligned}$$

We assume that *vars* is extended to role steps such that $\mathit{vars}(s)$ denotes the variables of the message pattern of the role step s.

A protocol P is *well-typed* with respect to a typing *ty* if and only if all its reachable states are well-typed with respect to *ty*; that is

$$\mathit{well\text{-}typed}_{ty}(P) \overset{\text{def}}{=} \forall q \in \mathit{reachable}(P).\ \mathit{well\text{-}typed}_{ty}(q)\ .$$

Conversely, we say that a typing *ty* is *sound* with respect to a protocol P if and only if P is well-typed with respect to *ty*.

We illustrate the use of type assertions on the CR' protocol, which is identical to the CR protocol from Example 1 except that in the first message the client identity is also sent and the server uses public key encryption to return the session key.

Example 8 (CR' Protocol)**.** We define $CR' \overset{\text{def}}{=} \{C', S'\}$, where C' and S' are defined as follows for $c, s \in AVar$, $k \in \mathit{Fresh}$, and $v \in MVar$.

$$\begin{aligned} C' &\overset{\text{def}}{=} \langle\, \mathsf{Send}_1(\{\!|c,k|\!\}_{\mathsf{pk}(s)}), \quad \mathsf{Recv}_1(\{\!|k|\!\}_{\mathsf{pk}(c)})\,\rangle \\ S' &\overset{\text{def}}{=} \langle\, \mathsf{Recv}_1(\{\!|c,v|\!\}_{\mathsf{pk}(s)}), \quad \mathsf{Send}_2(\{\!|v|\!\}_{\mathsf{pk}(c)})\,\rangle \end{aligned}$$

The CR' protocol is well-typed with respect to the typing

$$\begin{aligned} CR'_{ty} \overset{\text{def}}{=} \{\, & (C', c) \mapsto \mathsf{Ag},\ (C', s) \mapsto \mathsf{Ag}, \\ & (S', c) \mapsto \mathsf{Ag},\ (S', s) \mapsto \mathsf{Ag},\ (S', v) \mapsto (C'.k \cup \mathsf{kn}(S'_1))\,\}\ . \end{aligned}$$

We provide a formal proof of this claim in Example 10 after we have shown how to exploit type assertions and how to prove their soundness.

4.3.2 Exploiting Type Assertions

The following example illustrates the problem that we will solve using type assertions.

Example 9. As the CR' protocol does not send any hashes, one may expect its CHAIN rule for hashes to be

$$\frac{\mathsf{h}(x) \in \mathit{knows}(tr)}{x \prec_{tr} \mathsf{h}(x)} \text{ HCHAIN}_{CR'} \quad .$$

However, as we can see in Figure 6, there are also two decryption-chain cases, subsequently named (a) and (b), that arise when instantiating and unfolding the CHAIN rule. Case (a) states that it may be possible to learn $\mathsf{h}(x)$ from the content $\sigma(c, j)$ of the agent variable c that the client sends in its first message. Case (b) states that it may be possible to learn $\mathsf{h}(x)$ from the content $\sigma(v, j)$ of the variable v that the server sends in its second message.

We can remove both of these cases, as they are always false. This is obvious for Case (a), as agent variables contain agent names and agent names are never equal to hashes. For Case (b), the intuition is that the intruder must have faked the message received in step (j, S'_1) for the variable $\sigma(v, j)$ to contain the hash $\mathsf{h}(x)$. Hence, the intruder must have known $\mathsf{h}(x)$ already before (j, S'_1) was executed. This contradicts the statement in Case (b) that the intruder did *not* know $\mathsf{h}(x)$ before step (j, S'_2).

Formally, the above arguments exploit the fact that the CR' protocol is well-typed with respect to the type assertion CR'_{ty}. From this and $(tr, th, \sigma) \in \mathit{reachable}(CR')$, we have $\mathit{well\text{-}typed}_{CR'_{ty}}(tr, th, \sigma)$, which we use as follows to show that the cases (a) and (b) are contradictory.

In Case (a), we use $\mathit{well\text{-}typed}_{CR'_{ty}}(tr, th, \sigma)$ to derive

$$\sigma(c, j) \in [\![CR'_{ty}(C', c)]\!]^{j}_{(tr,th,\sigma)} \ ,$$

which is equivalent to $\sigma(c, j) \in [\![\mathsf{Ag}]\!]^{j}_{(tr,th,\sigma)}$ and hence also to $\sigma(c, j) \in \mathit{Agent}$. Therefore, only the first case of the *chain* predicate applies, which implies $\sigma(c, j) = \mathsf{h}(x)$ and hence $\mathsf{h}(x) \in \mathit{Agent}$, which is false. Thus we can remove the first decryption-chain case in Figure 6.

In Case (b), we use $\mathit{well\text{-}typed}_{CR'_{ty}}(tr, th, \sigma)$ to show that $\sigma(v, j) \in [\![C'.k \cup \mathsf{kn}(S'_1)]\!]^{j}_{(tr,th,\sigma)}$, which is equivalent to

$$\begin{array}{ll} \textbf{(1)} & (\exists i.\ \mathit{role}_{th}(i) = C' \wedge \sigma(v, j) = k\sharp i) \vee \\ \textbf{(2)} & (\sigma(v, j) \prec_{tr} (j, S'_1)) \ . \end{array}$$

For subcase **(1)**, we can proceed as for the agent variable above. We unfold the *chain* predicate and conclude $k\sharp i = \mathsf{h}(x)$, which is a contradiction. For subcase **(2)**, we conclude

$$\begin{aligned} &\sigma(v, j) \prec_{tr} (j, S'_1) \prec_{tr} (j, S'_2) \prec_{tr} \{\!|\sigma(v, j)|\!\}_{\mathsf{pk}(\sigma(c,j))} \wedge \\ &\qquad\qquad \mathit{chain}_{tr}(\{\{\!|\sigma(v, j)|\!\}_{\mathsf{pk}(\sigma(c,j))}, \mathsf{sk}(\sigma(c, j))\},\ \sigma(v, j),\ \mathsf{h}(x)) \end{aligned}$$

$$
\frac{m \in \mathit{knows}(tr) \qquad \mathit{well\text{-}typed}_{ty}(tr, th, \sigma)}{\begin{array}{l}
(m \in \mathit{IK}_0) \lor \\
(\exists x.\ \ m = \mathsf{h}(x) \ \land x \prec_{tr} \mathsf{h}(x) \qquad\qquad) \lor \\
(\exists x\ k.\ \ m = \{\!|x|\!\}_k \ \land x \prec_{tr} \{\!|x|\!\}_k \ \land k \prec_{tr} \{\!|x|\!\}_k \ ) \lor \\
(\exists x\ y.\ \ m = (x, y) \ \land x \prec_{tr} (x, y) \ \land y \prec_{tr} (x, y) \ ) \lor \\
(\exists R \in P.\ \exists\, \mathsf{Send}_l(pt) \in R.\ \exists j.\ \mathit{role}_{th}(j) = R \land \\
\qquad\qquad (\forall v \in \mathit{vars}(pt).\ \sigma(v, j) \in [\![ty(R, v)]\!]^{j}_{(tr, th, \sigma)}) \land \\
\qquad\qquad \mathit{chain}_{tr}(\{(j, \mathsf{Send}_l(pt))\},\ \mathit{inst}_{\sigma,j}(pt),\ m) \qquad)
\end{array}} \ \textsc{TypChain}
$$

Figure 7: The TypChain rule, a typed version of the Chain rule.

$$
\frac{m \prec_{tr} e \qquad e \in E \qquad \mathit{chain}_{tr}(E,\ m,\ x)}{\mathit{false}} \ \textsc{ChainIrr}
$$

Figure 8: The ChainIrr rule, which holds for $(tr, th, \sigma) \in \mathit{reachable}(P)$.

by combining all facts and additionally using the rules Exec and Role to derive $(j, S'_1) \prec_{tr} (j, S'_2)$. Note that $\mathit{chain}_{tr}(E,\ m,\ m')$ implies that there exists an $x \in \mathit{split}(m)$ such that $e \prec_{tr} x$ for every $e \in E$. Hence there exists an $x \in \mathit{split}(\sigma(v, j))$ such that $x \prec_{tr} (j, S'_1) \prec_{tr} \{\!|\sigma(v, j)|\!\}_{\mathsf{pk}(\sigma(c,j))} \prec_{tr} x$. This contradicts the irreflexivity of $\prec_{tr}$ and we can also remove the second decryption-chain case in Figure 6.

In general, we exploit type assertions by instantiating the rule TypChain given in Figure 7 instead of the Chain rule in the first step of our proof strategy. The TypChain rule is a version of the Chain rule that additionally states that all variables contained in send steps are instantiated according to their type. Hence, whenever we encounter an expression of the form $\mathit{chain}_{tr}(E,\ \sigma(v, j),\ m)$ in a decryption-chain case, we can proceed by instantiating $\sigma(v, j)$ with an arbitrary message m corresponding to v's type and simplifying the resulting cases as follows.

The type constructors corresponding to the message constructors constrain the structure of m sufficiently that we can unfold the *chain* predicate. A union-type $\alpha \cup \beta$ results in an additional case split. For a $\mathsf{kn}(s)$ type, we can reduce the case to false, provided that s is the role step where v is instantiated. We ensure this side condition by constructing our type assertions accordingly. To justify the reduction to false, we use ChainIrr rule given in Figure 8, which generalizes the argument used in the example above.

4.3.3 Proving the Soundness of Type Assertions

The core inference rules together with the rules for exploiting type assertions allows one to prove security properties for a protocol P under the assumption that P is well-typed for some fixed type assertion *ty*. Such a proof can be seen as a proof in a typed model, which ensures *by definition* that variables are only instantiated according to their specified type. However, such a proof does not exclude the existence of *type-flaw attacks*, which are reachable states that are not well-typed and violate the security property. To exclude the existence of type-flaw attacks, we must prove the soundness of the type assertion *ty*; i.e., we must prove that

$$\forall q \in \mathit{reachable}(P).\ \mathit{well\text{-}typed}_{ty}(q)\ .$$

We prove this by induction over the reachable states of the protocol P. The only non-trivial case of this induction stems from the RECV transition. It is the only transition where a thread i could add a role step s that instantiates a variable $v \in \mathit{vars}(s)$ to the trace. If v is an agent variable, then it suffices to show that it is mapped to the Ag type, as agent variables are always instantiated with agent names. If v is a message variable, then we exploit that the intruder must know the message m received by the role step s and that, from the induction hypothesis, the current state is well-typed with respect to the type assertion *ty*. Hence, we can use the TYPCHAIN rule to establish the possible origins of the received message m. If the type assertion *ty* is sound, then we expect the kn part of v's type to cover the case where the intruder fakes m and the rest of v's type to cover the case where the message is (part of) a message sent by another thread. The following theorem, proven in Isabelle/HOL, formalizes this argument.

Theorem 1 (Soundness of Type Assertions)**.** *A protocol P is well-typed with respect to the typing ty provided the following two propositions hold.*

1. *For every $R \in P$, $s \in R$, and $v \in \mathit{vars}(s) \cap \mathit{AVar}$, it holds that $ty(R, v) = \mathsf{Ag}$.*
2. *For every $R \in P$, $s \in R$, and $v \in \mathit{vars}(s) \cap \mathit{MVar}$, every state $(tr, th, \sigma) \in \mathit{reachable}(P)$, every thread $i \in \mathit{dom}(th)$, and all sequences of role steps* done *and* rest*, the assumptions*
   - $\mathit{inst}_{\sigma,i}(pt) \in \mathit{knows}(tr)$
   - $th(i) = (R, \langle \mathsf{Recv}_l(pt)\rangle \mathbin{\hat{}} \text{rest})$
   - $R = \text{done} \mathbin{\hat{}} \langle \mathsf{Recv}_l(pt)\rangle \mathbin{\hat{}} \text{rest}$
   - $v \in (\mathit{vars}(pt) \cap \mathit{MVar}) \setminus \bigcup_{s \in \text{done}} \mathit{vars}(s)$
   - $\mathit{well\text{-}typed}_{ty}(tr, th, \sigma)$

   imply
   $$\sigma(v, i) \in [\![ty(R, v)]\!]^{i}_{(tr \mathbin{\hat{}} \langle\ \mathsf{St}(i, \mathsf{Recv}_l(pt))\ \rangle,\ th,\ \sigma)}\ .$$

We illustrate the use of this theorem in the following example.

$$\frac{\{\!|m|\!\}_x \in \mathit{knows}(tr) \qquad \mathit{well\text{-}typed}_{CR'_{ty}}(tr, th, \sigma)}{\begin{array}{ll}(m \prec_{tr} \{\!|m|\!\}_x \wedge x \prec_{tr} \{\!|m|\!\}_x) \vee & \\ (\exists j.\ \mathit{role}_{th}(j) = C' \wedge (j, C'_1) \prec_{tr} \{\!|\sigma(c,j), k\sharp j|\!\}_{\mathsf{pk}(\sigma(s,j))} = \{\!|m|\!\}_x & \\ \qquad \wedge\, \sigma(c,j) \in \mathit{Agent} \wedge \sigma(s,j) \in \mathit{Agent} &) \vee \\ (\exists j.\ \mathit{role}_{th}(j) = S' \wedge (j, S'_2) \prec_{tr} \{\!|\sigma(v,j)|\!\}_{\mathsf{pk}(\sigma(c,j))} = \{\!|m|\!\}_x & \\ \qquad \wedge\, \sigma(c,j) \in \mathit{Agent} \wedge \sigma(v,j) \in [\![C'.k \cup \mathsf{kn}(S'_1)]\!]^i_{(tr,th,\sigma)} &)\end{array}} \ \textsc{EChain}_{CR'}$$

Figure 9: The $\textsc{EChain}_{CR'}$ rule derived by instantiating the $\textsc{TypChain}$ rule with an arbitrary encryption, the CR' protocol, and the CR'_{ty} typing.

Example 10 (Soundness of the CR'_{ty} assertion)**.** Recall the typing

$$\begin{aligned} CR'_{ty} \overset{\text{def}}{=} \{\, &(C', c) \mapsto \mathsf{Ag}, (C', s) \mapsto \mathsf{Ag}, \\ &(S', c) \mapsto \mathsf{Ag}, (S', s) \mapsto \mathsf{Ag}, (S', v) \mapsto (C'.k \cup \mathsf{kn}(S'_1))\,\}\ . \end{aligned}$$

We prove that the protocol CR' is well-typed with respect to this typing.

Proof. Applying Theorem 1 yields five cases. The first four cases deal with the agent variables of the roles C' and S' and are trivial. The fifth case deals with the instantiation of variable v in step (i, S'_1) of some thread i executing the S' role.

We use the assumption $\mathit{inst}_{\sigma,i}(\{\!|c, v|\!\}_{\mathsf{pk}(s)}) \in \mathit{knows}(tr)$ provided by Theorem 1 to show that

$$\sigma(v,i) \in [\![C'.k \cup \mathsf{kn}(S'_1)]\!]^i_{(tr\hat{\ }\langle\, \mathsf{St}(i, S'_1)\,\rangle,\ th,\ \sigma)}$$

holds for every state $(tr, th, \sigma) \in \mathit{reachable}(CR')$ satisfying $\mathit{well\text{-}typed}_{CR'_{ty}}(tr, th, \sigma)$.

After applying the $\textsc{EChain}_{CR'}$ rule given in Figure 9 to

$$\{\!|\sigma(c,i), \sigma(v,i)|\!\}_{\mathsf{pk}(\sigma(s,i))} \in \mathit{knows}(tr)$$

and dropping some conjuncts that are not required in the remainder of the proof, we are left with the following conclusion.

$$\begin{array}{lll} \textbf{(1)} & (\sigma(c,i), \sigma(v,i)) \prec_{tr} \{\!|\sigma(c,i), \sigma(v,i)|\!\}_{\mathsf{pk}(\sigma(s,i))} &) \vee \\ \textbf{(2)} & (\exists j.\ \mathit{role}_{th}(j) = C' \wedge \{\!|\sigma(c,j), k\sharp j|\!\}_{\mathsf{pk}(\sigma(s,j))} = \{\!|\sigma(c,i), \sigma(v,i)|\!\}_{\mathsf{pk}(\sigma(s,i))}) &) \vee \\ \textbf{(3)} & (\,\exists j.\ \mathit{role}_{th}(j) = S' \wedge \sigma(v,j) \in [\![C'.k \cup \mathsf{kn}(S'_1)]\!]^j_{(tr,th,\sigma)} & \\ & \qquad \wedge\, \{\!|\sigma(v,j)|\!\}_{\mathsf{pk}(\sigma(c,j))} = \{\!|\sigma(c,i), \sigma(v,i)|\!\}_{\mathsf{pk}(\sigma(s,i))} &) \end{array}$$

Case **(1)** states the intruder could have faked the received message. This case is covered by the $\mathsf{kn}(S'_1)$ part of v's type, as we have $\sigma(v,i) \in \mathit{knows}(tr)$ using rules $\textsc{Ord}_2$ and $\textsc{Known}$. Hence, $\sigma(v,i) \in [\![\mathsf{kn}(S'_1)]\!]^i_{(tr\hat{\ }\langle\, \mathsf{St}(i,S'_1)\,\rangle,\ th,\ \sigma)}$, as

$$\sigma(v,i) \prec_{tr\hat{\ }\langle\, \mathsf{St}(i,S'_1)\,\rangle} (i, S'_1) \ \Leftrightarrow\ \sigma(v,i) \in \mathit{knows}(tr)\ .$$

Case **(2)** states that the received message could have been sent by a client thread j. This case is covered by the $C'.k$ part of v's type, as we have

$$\mathit{role}_{th}(j) = C' \ \wedge\ \sigma(v,i) = k\sharp j$$

due to the injectivity of $\{\!|\cdot|\!\}$ and pairing.

Case **(3)** states that a server thread j could have sent the message matching the received message in his second step. In a typed model, this case would be impossible, as the types of the sent and received messages do not match. In our untyped model, a similar argument applies. However, it involves an additional case, as the types may not match due to the interaction with the intruder.

Formally, we have $\sigma(v,j) = (\sigma(c,i), \sigma(v,i))$ due to the injectivity of $\{\!|\cdot|\!\}$. Given this equality, we have

$$(\sigma(c,i), \sigma(v,i)) \in [\![C'.k \cup \mathsf{kn}(S'_1)]\!]^{j}_{(tr,th,\sigma)} \ .$$

We have $(\sigma(c,i), \sigma(v,i)) \in [\![\mathsf{kn}(S'_1)]\!]^{j}_{(tr,th,\sigma)}$ because $[\![C'.k]\!]^{j}_{(tr,th,\sigma)}$ does not contain any pairs. Unfolding the definition of $[\![\cdot]\!]$ yields

$$(\sigma(c,i), \sigma(v,i)) \prec_{tr} (j, S'_1) \ ,$$

which implies $\sigma(v,i) \in \mathit{knows}(tr)$ due to the rules Ord$_2$ and Known. Hence, $\sigma(v,i) \in [\![\mathsf{kn}(S'_1)]\!]^{i}_{(tr\,\hat{}\,\langle\, \mathsf{St}(i,S'_1)\,\rangle,\ th,\ \sigma)}$, which concludes the proof of Case **(3)**.

Thus, the CR' protocol is well-typed with respect to the typing CR'_{ty}. $\square$

4.4 Discussion of Decryption-Chain Reasoning

We call the proof strategy that we described in the previous two sections *decryption-chain reasoning*. Decryption-chain reasoning suffices for verifying many security protocols; we give examples in our case studies in Section 5.2.4.

In contrast to other security protocol verification methods, decryption-chain reasoning does not require a typed model. Nevertheless, the notion of types plays an important role in decryption-chain reasoning. Types yield a uniform construction of protocol-specific invariants (i.e., well-typedness with respect to a protocol-specific typing) that are strong enough to reason about messages of unbounded size. Said differently, decryption-chain reasoning illustrates that we can shift the notion of types from being an a-priori assumption on the semantics of security protocol execution to serving as a powerful tool for constructing security proofs.

Nevertheless, decryption-chain reasoning is not a silver bullet. It may fail in two ways. (1) A protocol may not be typeable; i.e., there does not exist a typing with respect to which the protocol is well-typed. (2) The case distinctions provided by the TypChain rule are too weak to prove the security property of interest.

Problem (1) is inherent to all approaches based on types. It must be solved per protocol by extending the set of types such that all reachable states are described. Afterwards, our approach of using the $\mathsf{kn}(\cdot)$ type constructor to describe the intruder interaction can be applied again.

Problem (2) exists as, even for typeable protocols, decryption-chain reasoning is necessarily incomplete. If it were complete, we could then decide the secrecy problem for typeable protocols because we could enumerate both proofs as well as attacks. This would contradict the undecidability of the secrecy problem with unbounded sessions and nonces [20], as the proof also applies to typeable protocols.

The following example illustrates the incompleteness of decryption-chain reasoning for a typeable protocol.

Example 11 (Case distinctions without progress)**.** Consider the (artificial) protocol $P \overset{\text{def}}{=} \{I, R\}$, where

$$I \overset{\text{def}}{=} \langle \mathsf{Send}_1(\{\!|n, n|\!\}_{\mathsf{k}(a,b)})\rangle$$
$$R \overset{\text{def}}{=} \langle \mathsf{Recv}_1(\{\!|v, w|\!\}_{\mathsf{k}(a,b)}), \mathsf{Send}_2(\{\!|v, n'|\!\}_{\mathsf{k}(a,b)})\rangle$$

and $n, n' \in \mathit{Fresh}$, $v \in \mathit{MVar}$, and $a, b \in \mathit{AVar}$. We can show that a and b have type Ag, v has type $I.n \cup \mathsf{kn}(R_1)$, and w has type $I.n \cup R.n' \cup \mathsf{kn}(R_1)$. Moreover, the contents of variable v in role R are obviously secret, provided that both a and b are uncompromised. However, we cannot prove the above secrecy property using decryption-chain reasoning.

The problem in this example is that the message sent in step R_2 can be received in step R_1. Hence, there can be an *unbounded* chain of threads executing the R role where each thread receives the message that the previous thread sent. This unbounded chain of threads manifests itself during proof construction as follows. At some point in the proof construction, the only inference step left is to apply the TypChain rule to determine the possible origin of the message m that a thread i executing the R role receives in its first step. After this application, we are left with a case stating that m was sent in the second step of some other thread j also executing the R role. In this case, our proof makes no progress, as we know as much about thread j as we already knew about thread i before the case distinction. Said differently, the proof of the authenticity of the message m that is sent in the second step of some role R depends on itself, when considering the limited perspective of decryption-chain reasoning.

Note that we can resort to induction over the reachable states to reason about protocols like the one above. We can then use decryption-chain reasoning for the individual induction steps. Moreover, the above problem is of a theoretical nature. Many practical protocols have an intended message flow that is acyclic and ensure that this intended message flow is achieved, e.g., using tagging.

Example 12 (Ensuring an acyclic message flow)**.** We can ensure an acyclic message flow for protocol P from Example 11 by redefining $P \overset{\text{def}}{=} \{I, R\}$ as

$$I \overset{\text{def}}{=} \langle \mathsf{Send}_1(\{\!|1, n, n|\!\}_{\mathsf{k}(a,b)})\rangle$$
$$R \overset{\text{def}}{=} \langle \mathsf{Recv}_1(\{\!|1, v, w|\!\}_{\mathsf{k}(a,b)}), \mathsf{Send}_2(\{\!|2, v, n'|\!\}_{\mathsf{k}(a,b)})\rangle$$

for $1, 2 \in \mathit{Const}$. After inserting these tags, decryption-chain reasoning works without resorting to induction over the reachable states.

5 Machine-Checked Security Proofs

In this section, we present two approaches for constructing machine-checked security proofs. The first approach uses Isabelle/HOL to interactively construct the corresponding proof script using our verification theory. The second approach uses an algorithm to automatically generate the corresponding Isabelle/HOL proof script. We discuss both approaches below.

5.1 Interactive Proof Construction

To simplify the interactive construction of security proofs, we extended Isabelle's proof language [43] with commands to define roles, protocols, and type assertions as well as a tactic that automates the application of the TypChain rule.

The commands to define roles and protocols introduce corresponding constants and set up Isabelle's automation infrastructure to simplify reasoning about the steps and roles of the protocol. The command to define a type assertion automatically derives the simplified instances of the TypChain rule under the assumption that this type assertion is sound for the given protocol.[1] Provided that this soundness assumption holds, it can be discharged in an interactive proof by applying Theorem 1 and using decryption-chain reasoning for the resulting proof obligations, i.e., the theorem's premises.

Case distinctions on the possible origins of a message $m \in \mathit{knows}(\mathit{tr})$ are automated using the tactic "sources". A call "sources m" selects the simplified TypChain rule instance corresponding to the outermost constructor of the message m and uses it to enumerate the possible origins of m. The resulting cases are discharged automatically using Isabelle's built-in tools, if possible. Otherwise, they are named and presented to the user for further processing.

The following example and the proofs in our case studies [31], show that our mechanization of decryption-chain reasoning allows for succinct, machine-checkable security proofs.

Example 13. The session-key secrecy proof given in Example 5 corresponds to the proof script given in Figure 10, which is checked by Isabelle in under a second. Note that we have taken minor liberties in pretty-printing to improve readability.

Line 1 begins the lemma, named "client-k-secrecy". The statement "(**in** CR-state)" expresses that this lemma is proven under the assumption that $(\mathit{tr}, \mathit{th}, \sigma)$ is a reachable state of the CR protocol. Lines 2–6 state the secrecy property. The expression NO(″k″, i) denotes the freshly generated message $k\sharp i$. The expression AV(″s″) denotes the agent variable s.

[1] The type assertion soundness assumptions and the simplified TypChain rule instances are managed using Isabelle's *locale* infrastructure [2].

```
 1: lemma (in CR-state) client-k-secrecy:
 2:   assumes
 3:     “role_th(i) = C”
 4:     “σ(AV(″s″), i) ∉ Compr”
 5:     “NO(″k″, i) ∈ knows(tr)”
 6:   shows “False”
 7: proof(sources “NO(″k″, i)”)
 8:   case C_1-k thus “False”
 9:   proof(sources “SK (σ(AV(″s″), i))”)
10:     case ik0 thus “False” by auto
11:   qed
12: qed
```

Figure 10: Proof script of session-key secrecy for the *CR* protocol.

Lines 7–12 give the proof, which has the same structure as the pen-and-paper proof from Example 5. Isabelle supports its interactive construction as follows. After parsing the lemma's statement, Isabelle prints all assumptions and the goal that we must prove. There is only one assumption to which we can apply the TypChain rule[2], which we do in Line 7. Isabelle computes the resulting non-trivial cases and prints them together with their additional assumptions (i.e., what agents are compromised, equalities between messages, what events happened, the order between events, and the roles of the involved threads). Only one non-trivial case, C_1-k, results from this application of the TypChain rule. We select it in Line 8 and state that it is contradictory; i.e., we can prove “False” from its assumptions. We prove “False” by applying the TypChain rule to “SK (σ(AV(″s″), i))”, the private key of the server that thread i is communicating with. The only non-trivial case is “ik0”, which states that the server “σ(AV(″s″), i)” was compromised. This case contradicts the assumption in Line 4. We use Isabelle's built-in tactic “auto” to prove this.

In Appendix A, we also provide Isabelle formalizations of the authentication proof from Example 6 and the type assertion soundness proof from Example 10. In both cases, our extension of Isabelle's proof language allows for succinct formalizations highlighting the main argument underlying these proofs. This succinctness is one of the key properties of decryption-chain reasoning. Other examples of this are the security proofs [31] that we interactively constructed for the Yahalom [37], the Kerberos V [6], and the TLS handshake [36] protocols based on the models developed using the Inductive Approach [35]. Modeling each protocol took under an hour. Proving the security properties took 1.5 hours for Yahalom, 2 hours for Kerberos V, and 2.5 hours for the TLS handshake protocol. These times represent roughly a two orders of magnitude improvement over the Inductive Approach, as we will see in Section 6.

[2] We can use the TypChain rule because we prove after the definition of the *CR* protocol that it is well-typed with respect to the typing $\{(C, s) \mapsto \mathsf{Ag}, (S, s) \mapsto \mathsf{Ag}, (S, v) \mapsto Ck \cup \mathsf{kn}(S_1)\}$.

5.2 Automatic Proof Generation

We now describe an algorithm for automatically generating Isabelle/HOL proof scripts for secrecy and authentication properties of security protocols. The input of the algorithm is a protocol $P \in \mathit{Protocol}$, a set of type assertions ty, and a security property ϕ. If the algorithm succeeds in proving the soundness of the type assertions ty and the validity of ϕ, then it outputs an Isabelle proof script. The script contains the specification of the protocol P, the type assertions ty and its soundness proof, and a lemma stating ϕ and its proof.

Our algorithm uses symbolic messages and variables for thread identifiers as part of its proof state representation. Symbolic messages are built from the message constructors given in Section 3.2.1, the uninterpreted function $\sigma : \mathit{Var} \times \mathit{TID} \to \mathit{Msg}$ denoting a variable store, and the function $\_^{-1} : \mathit{Msg} \to \mathit{Msg}$ denoting symbolic key inversion. In the following sections, we use i and j (possibly primed) to denote thread identifier variables, a to denote symbolic agent variables, m to denote symbolic messages, s to denote role-steps, R to denote roles, and e to denote events built from symbolic messages. Our algorithm handles security properties that can be represented as closed formulas of the form

$$\forall (\mathit{tr}, \mathit{th}, \sigma) \in \mathit{reachable}(P).\ \forall i_1 \dots i_l.\ (\bigwedge\nolimits_{A \in \Gamma} A) \Rightarrow \exists i_{l+1} \dots i_n.\ (\bigwedge\nolimits_{B \in \Delta} B)$$

where $0 \le l \le n$ and Γ, Δ are sets of *atoms* of the following form.

$$\begin{array}{lllll} i = i' & m = m' & \mathit{role}_{\mathit{th}}(i) = R & \sigma(a,i) \in \mathit{Compr} & \mathit{false} \\ e \prec_{\mathit{tr}} e' & (i,s) \in \mathit{steps}(\mathit{tr}) & m \in \mathit{knows}(\mathit{tr}) & \sigma(a,i) \notin \mathit{Compr} & \end{array}$$

We abbreviate such a security property as a *judgment* $\Gamma \vdash_P \Delta$, where implicitly all thread identifier variables in Γ are universally quantified and all thread identifier variables in Δ that do not occur in Γ are existentially quantified.

Example 14. The secrecy property ϕ_{sec} from Example 3 is represented as

$$\mathit{role}_{\mathit{th}}(i) = C,\ \sigma(s,i) \notin \mathit{Compr},\ k\sharp i \in \mathit{knows}(\mathit{tr}) \vdash_{CR} \mathit{false}.$$

The authentication property ϕ_{auth} from the same example is represented as

$$\begin{array}{rcl} \mathit{role}_{\mathit{th}}(i) = C, & & \\ \sigma(s,i) \notin \mathit{Compr}, & \vdash_{CR} & \begin{array}{l} \mathit{role}_{\mathit{th}}(j) = S,\ \sigma(s,i) = \sigma(s,j),\ k\sharp i = \sigma(v,j), \\ (i, C_1) \prec_{\mathit{tr}} (j, S_1),\ (j, S_2) \prec_{\mathit{tr}} (i, C_2). \end{array} \\ (i, C_2) \in \mathit{steps}(\mathit{tr}) & & \end{array}$$

Note that the above class of formulas covers typical secrecy and non-injective authentication properties of multi-party protocols [17, 29]. It does not include injective authentication properties, as they require an additional quantifier alternation. Showing injectiveness is however easy once one has shown non-injective agreement over the fresh data of all involved parties. Our algorithm does not do this automatically, but a user can derive injectiveness with minor effort in Isabelle/HOL from the non-injective authentication properties proven in the automatically generated proof script.

```
procedure $\textsc{CorePrfGen}(\Gamma \vdash_P \Delta,\ ty)$
    solve all equality premises of $\Gamma \vdash_P \Delta$
    saturate $\Gamma$ under all rules except $\textsc{TypChain}$
    if $\Gamma \vdash_P \Delta$ is trivially valid then
        print "by auto"
    else
        if there is no new $(m \in knows(tr)) \in \Gamma$ then
            fail (possible attack found: $\Gamma \vdash_P \Delta$)
        else
            select new $(m \in knows(tr)) \in \Gamma$
            print "proof(sources $m$)"
            $\mathcal{J} \leftarrow$ apply $\textsc{TypChain}$ to $m \in knows(tr)$ and $well\text{-}typed_{ty}(tr, th, \sigma)$
            for each $J \in \mathcal{J}$ do
                print "case $nameOf(J)$"
                $\textsc{CorePrfGen}(J,\ ty)$
            end for
            print "qed"
        end if
    end if
end procedure
```

Figure 11: The core proof generation algorithm CorePrfGen.

We present our algorithm in three steps. First, we describe the core proof generation algorithm, which mechanizes the use of the TypChain rule assuming the soundness of the given type assertions. Second, we describe an algorithm for proving the soundness of type assertions based on Theorem 1 and the core proof generation algorithm. We then show how to combine these two algorithms in our proof generation algorithm. Third, we describe two extensions that increase the scope and efficiency of our proof generation algorithm as well as the readability of the generated proofs. Afterwards, we present results from case studies.

5.2.1 Core Algorithm

The core proof-generation algorithm CorePrfGen is given in Figure 11. Given a judgment $\Gamma \vdash_P \Delta$ and a set of type assertions *ty*, this algorithm tries to prove the validity of $\Gamma \vdash_P \Delta$ under the assumption that the type assertions *ty* are sound with respect to the protocol P as follows.

First, the equalities between symbolic messages, thread identifiers, and roles in Γ are solved using unification in the equational theory that respects the definition of key inversion given in Section 3.2.1 and that regards $\sigma : Var \times TID \to Msg$ as an uninterpreted function and $\sharp : Fresh \times TID \to Msg$ as a free constructor for symbolic messages.

For example, the symbolic equation $x^{-1} = \mathsf{h}(y)$ is solved with the substitution $\{x \mapsto \mathsf{h}(y)\}$, the equation $((x^{-1})^{-1})^{-1} = \mathsf{sk}(a)$ is solved with $\{x \mapsto \mathsf{pk}(a)\}$, the equation $x\sharp i = x'\sharp i'$ is solved with $\{x \mapsto x', i \mapsto i'\}$, and the equation $x\sharp i = \sigma(x', i')$ is solved with the substitution $\{\sigma(x', i') \mapsto x\sharp i\}$. Note that we require such non-standard substitutions because we regard σ as an uninterpreted function, whose function values are therefore regarded as unknowns during unification. Note also that our non-standard construction of symbolic messages stems from our decision to formalize them in Isabelle/HOL using a shallow embedding; i.e., we define $^{-1}$, $\sharp$, and σ over ground messages only and use HOL terms and variables to represent symbolic messages. This simplifies the formalization of symbolic messages, but complicates the corresponding meta-theoretic results. See our work on extending decryption-chain reasoning to Diffie-Hellman exponentiation [38] for a pen-and-paper formalization of symbolic messages using standard notions from equational term rewriting.

If unification fails, then $\Gamma \vdash_P \Delta$ is trivially valid because there are contradicting equalities in Γ. This case is handled by the trivial validity check in Line 4. If unification succeeds, the resulting substitution of thread identifiers and messages is applied to $\Gamma \vdash_P \Delta$. Second, the premises Γ are saturated by extending them with all atoms derivable using inference rules other than the TypChain rule. Third, the resulting judgment $\Gamma \vdash_P \Delta$ is checked for *trivial validity*; namely, whether one of the following holds:

1. $\mathit{false} \in \Gamma$,
2. $e \prec_{tr} e \in \Gamma$,
3. $(x \in \mathit{Compr}) \in \Gamma$ and $(x \notin \mathit{Compr}) \in \Gamma$, or
4. there exists a substitution τ of the existentially quantified thread identifiers in Δ such that $\tau(\Delta) \setminus \Gamma$ consists of reflexive equalities only.

If the judgment $\Gamma \vdash_P \Delta$ is trivially valid, then Isabelle can prove the validity of $\Gamma \vdash_P \Delta$ using its built-in tactic "auto". Otherwise, the algorithm checks whether there exists an atom $(m \in \mathit{knows}(tr)) \in \Gamma$ that has not yet been selected. If no such atom exists, proof-generation fails and the corresponding proof state indicates a possible attack, as we explain in the next paragraph. Otherwise, we pick the next atom and apply the TypChain rule to it. In our case studies, we use a simple heuristic that first picks atoms with messages containing long-term keys, then atoms with messages containing nonces supposedly known to the intruder, and finally atoms that have been in the proof state for the longest time. The application of the TypChain rule to the selected atom $m \in \mathit{knows}(tr)$ results in a case distinction on how the intruder learned m. Each case is represented again as a judgment J of the form $\Gamma \cup \Sigma \vdash_P \Delta$, where Σ are the new assumptions introduced by the case that J represents. For each case, we output the information necessary for Isabelle to know which case is being proven and generate the corresponding proof script by recursively calling CorePrfGen.

Despite the undecidability of protocol security, the CorePrfGen algorithm terminates for many practical protocols, including all the case studies from Table 1. Moreover, analogous to the algorithm underlying Scyther [15], the failure to generate a proof often indicates an attack. In particular, for secrecy properties written as a judgment $\Gamma \vdash_P \mathit{false}$ that only mentions roles, role-steps,

```
procedure TypePrfGen($P$, $ty$)
    $\mathcal{J} \leftarrow$ proof obligations of Theorem 1 showing that
        the type assertion $ty$ is sound for $P$
    for each $J \in \mathcal{J}$ do
        CorePrfGen($J$, $ty$)
    end for
end procedure
```

Figure 12: The TypePrfGen algorithm for generating type assertion soundness proofs.

```
procedure PrfGen($\Gamma \vdash_P \Delta$, $ty$)
    TypePrfGen($P$, $ty$)
    CorePrfGen($\Gamma \vdash_P \Delta$, $ty$)
end procedure
```

Figure 13: The proof generation algorithm PrfGen.

nonces, and variables of the protocol P, a failure to generate a proof always indicates an attack. An attack on such a property is a reachable state that satisfies the premises Γ. The only non-trivial constraints on such a state stem from the $m \in knows(tr)$ premises in Γ. Each case of an application of the TypChain rule provides an explanation of how the intruder learned m. Once we reach a non-contradictory set of premises Γ' without any remaining unsolved $m \in knows(tr)$ premises, we therefore know how to construct a reachable state satisfying Γ'; i.e., we construct such a state by choosing some instantiation of all variables in Γ' and executing the protocol steps and message deductions in Γ' according to their dependencies in Γ'. Analogously, we can construct attacks from failed proofs of authentication properties, provided they can be written as a judgment $\Gamma \vdash_P \Delta$ that only mentions roles, role-steps, nonces, and variables of the protocol P.

5.2.2 Generating Type Assertion Soundness Proofs

The TypePrfGen algorithm given in Figure 12 generates soundness proofs for type assertions. It uses an extended version of the CorePrfGen algorithm that also allows judgments of the form

$$\Gamma \vdash_P \sigma(v, i) \in [\![\alpha]\!]^{i}_{(tr\hat{}\langle\, \mathsf{St}(i, \mathsf{Recv}_l(pt))\,\rangle,\, th,\, \sigma)} \ ,$$

where $\alpha \in Type$ and Γ is a set of atoms as before. These judgments are sufficient to represent the proof obligations stemming from applications of Theorem 1. The $[\![\alpha]\!]^{i}_{(\ldots)}$ expressions are handled by unfolding the definition of $[\![\cdot]\!]$ for the given type α and the trivial validity of judgments is redefined accordingly. Note that Theorem 1 provides $well\text{-}typed_{ty}(tr, th, \sigma)$ as an induction hypothesis. This justifies our use of CorePrfGen in Line 5, which assumes $well\text{-}typed_{ty}(tr, th, \sigma)$.

The PrfGen algorithm given in Figure 13 combines the previous two algorithms. Analogous to CorePrfGen, a failure of the PrfGen algorithm provides useful information about the protocol under investigation. If the type assertion soundness proof fails, then there are other variable instantiations possible than the ones specified by the type assertions. These instantiations might be exploited in a type-flaw attack. If proving the judgment $\Gamma \vdash_P \Delta$ fails, then there is a possible attack on the security property being verified, as explained in the previous section.

5.2.3 Extensions

In this section, we describe two extensions of our proof generation algorithm: lemma instantiation and minimal proof generation. The aim of both extensions is to reduce the size of the generated proofs, which in turn improves proof checking time as well as human readability. Moreover, lemma instantiation also extends the scope of our algorithm, as we explain below.

Lemma instantiation shortens proofs by referring to already proven security properties. The idea is to instantiate the universally quantified thread variables of a security property modeled as a judgment such that the judgment's premises are satisfied in the current proof state and its conclusions can therefore be added to the proof state. Lemma instantiation is crucial for sharing subproofs between different cases of a proof. We can see this in the authentication proof from Example 6, where we instantiate the previously proven secrecy property twice. Without lemma instantiation, we are forced to prove the secrecy of the exchanged session key twice. Indeed it is not difficult to construct protocols where lemma instantiation even results in exponentially smaller proofs.

For some security protocols, we also use lemma instantiation to model additional assumptions (axioms) about a protocol's execution. A typical example is the assumption that a certificate authority's long-term private key is never compromised. We model such an assumption by assuming the corresponding security property instead of proving it. We can then exploit this property in subsequent proofs using lemma instantiation. This construction extends the scope of our algorithm. It allows us to verify properties of the form

$$\left(\bigwedge\nolimits_{1\leq i\leq n}(\Gamma_i \vdash_P \Delta_i)\right) \Rightarrow (\Gamma \vdash_P \Delta)$$

where the judgments $\Gamma_i \vdash_P \Delta_i$ model n additional assumptions about the execution of the security protocol P.

We implement lemma instantiation in the CorePrfGen algorithm using resolution of judgments. Assume given a lemma $\Gamma_1 \vdash_P \Delta$ with no existentially quantified thread identifiers in Δ. When proving a judgment of the form $\Gamma_1 \cup \Gamma_2 \vdash_P \Pi$, we can exploit this lemma using the following resolution rule.

$$\frac{\Gamma_1 \vdash_P \Delta \qquad \Delta \cup \Gamma_1 \cup \Gamma_2 \vdash_P \Pi}{\Gamma_1 \cup \Gamma_2 \vdash_P \Pi}$$

The thread identifier variables in $\Gamma_1 \vdash_P \Delta$ may need to be renamed for this rule to apply, which can be done as these variables are all universally quantified.

	Protocol	generation	checking
1	Amended Needham Schroeder Symmetric Key	0.36	45
2	Lowe's fixed Needham-Schroeder Public Key	0.20	25
3	Paulson's strengthened version of Yahalom	0.15	35
4	Lowe's modified Denning-Sacco Shared Key	0.15	37
5	BAN modified version of CCITT X.509 (3)	0.80	19
6	Lowe's modified BAN Concrete Andrew Secure RPC	0.01	10
7	Woo and Lam Pi 3	0.02	13
8	Kerberos V	2.40	202
9	Kerberos IV	12.87	308
10	TLS Handshake	0.84	48

Table 1: Timings in seconds for generating and checking minimal proofs.

Note that lemma instantiation is especially useful when $\Delta = \mathit{false}$; e.g., secrecy properties have this form. In this case, the second premise of the resolution rule becomes trivial, as $\mathit{false} \cup \Gamma_1 \cup \Gamma_2 \vdash_P \Pi$ always holds.

We also implement the generation of *minimal proofs*, which are proofs with a minimal number of TypChain rule applications. In a minimal proof every case distinction is required. This makes minimal proofs especially well-suited for human understanding, as every case distinction conveys information about how the protocol achieves the security property. We generate minimal proofs using a branch-and-bound strategy to minimize the number of case distinctions in the generated proofs. Moreover, we instantiate lemmas eagerly to ensure that their consequences are available before making a further case distinction. We remove superfluous lemma instantiations after a minimal proof is found, thereby further improving its readability. See Appendix A for two examples of minimal proofs generated using our algorithm.

5.2.4 Case Studies

Table 1 shows the case studies we performed using our `scyther-proof` tool, which implements the PrfGen algorithm and its extensions. The protocol models are distributed together with the source code of the tool [31]. Protocols 1-8 are modeled based on to their description in the SPORE [41] security protocol library. Protocols 9 and 10 are based on the corresponding models that were verified by Bella and Paulson using the Inductive Approach [8,36]. The times for proof generation and checking were measured using Isabelle2009-1 on an Intel Core 2 Duo 2.20GHz laptop with 2GB RAM. For all protocols, we prove non-injective agreement [29] of all shared data for all roles of the protocol where possible and secrecy for the freshly generated keys and payloads, if they are present. For protocols 8 and 10, we additionally prove non-injective synchronization [17], which is a strengthened variant of non-injective agreement.

The case studies demonstrate that our implementation of decryption-chain reasoning efficiently constructs machine-checked security proofs for complex protocols like Kerberos or the TLS handshake. On average 2.5 applications of the TypChain rule are required to prove a security property in our case studies. The comparatively high generation time for Kerberos IV is due to generating minimal proofs. It suggests that our minimization-strategy can be improved. The comparatively high checking times for the Kerberos protocols are due to the number of authentication proofs (one for each of the four roles) and the complexity of these proofs (each proof needs several nested case distinctions to determine the ordering of the individual steps). However, these times are definitely fast enough for machine-checking the security proofs once a protocol design is finished.

Most variables of the protocols in our case studies have a simple type: either Ag, $\mathsf{kn}(s)$, or $\mathsf{kn}(s) \cup R.n$, for the receive role-step s where the variable is instantiated and the nonce n that is sent in the corresponding send role-step in role R. We use a simple heuristic to infer such types by matching the receive role-steps with their corresponding send role-steps. Our heuristic suffices to infer the types of 115 out of the 118 variables of the protocols from Table 1. It fails only for the three variables inside an encryption that are instantiated with composed terms in Protocol 1 (Amended NS) and Protocol 9 (Kerberos IV). We specified their type by hand. For example, the client of the Kerberos IV protocol receives the authentication ticket sent by the Authentication server inside an encryption. The type of the corresponding variable is

$$\mathsf{kn}(C_2) \cup \{\!|\mathsf{kn}(C_2),\ \mathsf{Ag},\ A.\mathit{AuthKey},\ A.\mathit{Ta}|\!\}_{\mathsf{k}(\mathsf{Ag},\mathsf{Ag})}\ ,$$

where C_2 is the second step of the client role, which receives the authentication ticket, A is the Authentication server role, $\mathit{AuthKey}$ is the freshly generated authentication key, and Ta is a nonce that we use to model the timestamp of the authentication ticket. The innermost $\mathsf{kn}(C_2)$ type stems from a variable containing the client's name, which is received by the Authentication server as plaintext and could therefore be any message known to the intruder.

Note that our security protocol model has no built-in support for timestamps. In our models, we verify non-injective agreement on timestamps, which must therefore be distinct for a successful attack to be possible. We guarantee this by modeling timestamps as nonces. We model the fact that timestamps are guessable by leaking them to the adversary before their first use. The only thing not modeled are the validity checks on the timestamps.

6 Related Work

We discuss related work from five areas: interactive theorem proving, automatic proof methods, related proof methods where proofs are not machine-checked, types in security protocol verification, and related algorithms.

Interactive Methods for Machine-Checked Proofs. The Inductive Approach is one of the most successful approaches for interactively constructing machine-checked symbolic security proofs. It was initially developed by Paulson [35] and later extended by Bella [6] and Blanqui [12]. A protocol is defined indirectly as an inductively-defined set of traces, which denotes the protocol's executions in the context of an active adversary. Security properties are verified by formulating corresponding (possibly strengthened) protocol-specific invariants and proving them by induction. Formulating and proving these invariants constitutes the main effort when using this approach. In contrast, our protocol-independent invariants suffice for verifying protocols in all our case studies: we never needed to prove additional protocol-specific invariants using induction. This is the main reason for the reduction in proof construction time of almost two orders of magnitude in our case studies. Paulson reports that several days were needed for each of the three protocols analyzed in [35] and the analysis of the TLS handshake protocol took six weeks [36]. Note that these six weeks also include building the formal model. However, even if we assume the actual verification took only half this time, then our approach still reduces verification time by almost two orders of magnitude.

Two other approaches for the interactive proof construction of symbolic security proofs were developed using the PVS theorem prover. The first approach was developed by Evans and Schneider [21] based on a formalization of rank-functions [39]. Our improvement in proof construction time also applies to their work, as they state that their approach requires more interaction than Paulson's inductive approach. The second approach was developed by Jacobs and Hasuo [27]. It is based on a formalization of a variant of strand spaces [42] and authentication tests [24]. They do not provide proof construction times.

It is well known that symbolic security definitions such as those used in this paper, may miss attacks with respect to their corresponding computational security definitions. One would therefore ultimately like to construct machine-checked, computational security proofs. CertiCrypt [4] and EasyCrypt [3] provide a first step towards this goal. CertiCrypt formalizes the theory required for machine-checking computational security proofs of cryptographic primitives (e.g., ElGamal encryption) in the interactive theorem prover Coq. EasyCrypt leverages on SMT solvers to (partially) automate the construction of such proofs. Note that automatically constructing machine-checked, computational security proofs of complete protocols is an active research area. One interesting approach would be to formalize the theory underlying the CryptoVerif tool [10] in CertiCrypt and make it proof generating analogous to what we have done in this paper for the Scyther tool.

Automatic Generation of Machine-Checked Proofs. There are two existing approaches for automatically generating machine-checked protocol security proofs. The first approach is by Goubault-Larrecq [23]. He models a protocol and its properties as a set S of Horn-clauses whose consistency implies that the protocol satisfies its properties. A finite model finder is then used to find a certificate (i.e., a model) for S's consistency. This certificate is machine-checked using a model checker embedded in Coq.

The secrecy properties of protocols 1-5 from Table 1 were also analyzed by Goubault-Larrecq in [23]. Note that what is referred to in [23] as the Kerberos protocol is in fact the simpler Denning-Sacco shared key protocol (Protocol 4 in Table 1). The times reported in [23] are in the same range as ours. The approach can be used directly with equational theories, but currently cannot handle the strong authentication properties considered in our work. Moreover, the approach in [23] requires trusting the soundness of the (non-trivial) abstractions required to model security protocols using Horn-clauses. In contrast, our method uses a straightforward protocol model, from which we formally derive all verification rules.

Brucker and Mödersheim describe an approach for the automatic generation of machine-checkable proofs in [13]. They use the OFMC model checker [33] to compute a fixpoint of an abstraction of the transition relation of the protocol P of interest. This fixpoint overapproximates the set of reachable states of the protocol P. It is then translated to an Isabelle proof script certifying both that this fixpoint (and hence the protocol P) does not contain an attack and that the abstraction is sound with respect to an automatically generated trace-based execution model of P formalized in the style of the Inductive Approach. This execution model is typed and uses a non-standard intruder who can only send messages matching patterns occurring in the protocol.

In our previous work on decryption-chain reasoning [32], we provided a detailed timing comparison to their approach. We found that proof generation times are similar, but our proof checking times are orders of magnitude faster, ranging from a factor 10 to a factor 1700. These results also apply here, as our new implementation is as fast as the one used in [32].

Related Proof Methods. Similar to our approach, the TAPS security protocol verifier developed by Cohen [14] is based on a protocol-independent invariant construction. Given a protocol specification, TAPS heuristically derives a protocol specific secrecy invariant and tries to prove its soundness by delegating the corresponding proof obligations to a first-order theorem prover. If the theorem prover succeeds, the proven secrecy invariant is strong enough to prove typical secrecy and authentication properties using ordinary first-order reasoning.

The input for TAPS' secrecy invariant construction is a protocol specification together with hints that are often heuristically computed and, in some cases, manually provided. These hints state some form of type assertions and Floyd-Hoare style loop invariants, which are used to handle recursive protocols. In our approach, this construction can be seen as instantiating the TypChain rule for a specific protocol and a type assertion. The proof obligations generated by TAPS then correspond to the proof obligations that we discharge after an application of the type soundness theorem.

In contrast to our approach, TAPS does not construct machine-checked protocol security proofs. Only the proofs of the proof obligations delegated to the first-order theorem prover are machine-checked, while the crucial proof that these discharged proof obligations imply the soundness of the constructed secrecy invariant and the security of the protocol is not machine-checked.

However, the ideas underlying TAPS' secrecy invariant construction are similar to those underlying the TypChain rule. Hence, we expect that TAPS could be extended to generate fully machine-checked proofs using an approach similar to ours. The benefit of using TAPS' secrecy invariant construction is its support for reasoning about recursive protocols. The drawback of TAPS' secrecy invariants is that they do not represent the causal order of events explicitly, which complicates proving ordering related properties such as strong authentication properties or temporal secrecy properties (e.g., perfect forward secrecy). Hence, we believe that a better option to combine TAPS' advantages with our approach is to incorporate TAPS' use of hints in the form of a parametrized invariant with an associated soundness theorem similar to our support for type assertions.

Types in Security Protocol Verification. Many security protocol verification methods assume a typed execution model where variables are always instantiated with messages according to the variable's type. It is well-known that this assumption is not always sound and might lead to missed type-flaw attacks. Heather et al. [25] were the first to formally address the question of when a type assumption is sound. They propose a protocol transformation that ensures the soundness of type assumptions by reifying the types as tags in the messages. Li et al. [28] optimize this tagging scheme so that it requires fewer tags. Arapinis et al. [1] improve these results further. They propose a syntactic well-formedness condition that ensures the soundness of a type assumption for a more fine-grained type system for security protocols. Similarly, Delaune et al. [18] propose a syntactic condition for ensuring the well-typedness of models of cryptographic APIs.

In contrast to [1, 18, 25, 28], we do not rely on a syntactic criterion or a fixed protocol transformation to prove type assertion soundness. In principle, our use of type assertions is therefore applicable to more protocols. The price to pay for this increased scope is that we must prove type assertion soundness for each protocol individually. Fortunately, proving type assertion soundness turned out to be simple in all our case studies. The resulting proof obligations were easily discharged using decryption-chain reasoning.

Bhargavan et al. [9] show how to verify the source code of implementations of security protocols using a type system based on refinement types and delegating proof obligations to an SMT solver. They use type assertions as a means to specify the invariants that the protocol is expected to satisfy. Some of these invariants correspond to expected security properties, while the remaining ones are auxiliary invariants required to discharge the proof obligations that arise during type-checking. Therefore, their approach requires more type assertions, which are also of greater complexity, than our approach. Additionally, almost all type assertions in our case studies can be inferred heuristically from the protocol specification. However, their type assertions are more expressive than ours, as they can be used to provide loop invariants for handling recursive protocols, similar to the hints used in Cohen's TAPS tool.

Related Algorithms. Our proof-generation algorithm represents a substantial extension of the Scyther algorithm [15], which in turn is a descendant of the Athena algorithm [40]. Our algorithm extends the Scyther algorithm with support for a larger range of security properties, automatic generation of type assertion soundness proofs, lemma instantiation, and proof minimization.

7 Conclusions

We formally derive in Isabelle/HOL a verification theory for security protocols from a straightforward operational semantics. Security proofs constructed in this theory therefore provide strong correctness guarantees. We additionally develop and implement an algorithm for the automatic generation of human-readable, machine-checked security proofs based on this theory. The corresponding `scyther-proof` tool and our Isabelle/HOL theories are freely available [31]. We thereby provide tool support for both interactive and automatic construction of machine-checked proofs of secrecy and strong authentication properties. We show that this proof construction works efficiently in a number of case studies.

Benefits for practitioners. Intuitively, our verification theory provides protocol designers with a formal language for expressing correctness arguments about their protocols. Using the `scyther-proof` tool, a protocol designer can also automatically search for a short correctness argument expressed in this language and inspect the result to understand why the protocol is correct. In some cases, this may help the protocol designer to simplify his protocols by detecting and removing messages and assumptions that are not exploited in the correctness proofs. We further support the inspection of a protocol's security proof by implementing `scyther-proof` such that it can also generate HTML pages that visualize the proof obligations of each proof step as a partial attack.

Our tool set also simplifies certifying protocols at the highest evaluation levels, where machine-checked proofs are a formal requirement [30]. This is especially interesting, as such certification is currently pursued in several countries [23, 30]. As an example, we demonstrate the applicability of our tool set to a real world standard in [5]. There, we analyze, repair, and certify our repairs of the ISO/IEC 9798 standard for entity authentication [26]. This standard comprises 17 protocols, which are used as a core building block in numerous other standards. It takes `scyther-proof` less than 20 seconds to generate the proof script that justifies the correctness of the parallel composition of all our repaired versions of these 17 protocols. Checking this proof script takes Isabelle/HOL less than three hours. These times are practical and significantly lower than the time required to manually construct corresponding proofs.

Extension to equational theories. The approach presented in this paper uses the free term algebra to represent cryptographic messages. This suffices for modeling classical cryptographic operators like symmetric encryption and signing, but not for modeling operators such as Diffie-Hellman exponentiation and XOR. In a parallel development [38], we show how to adapt decryption-chain reasoning to a rich protocol execution model based on multiset rewriting and a non-free term algebra modeling Diffie-Hellman exponentiation and standard cryptographic operators. In contrast to this work, we have formalized the theory of [38] on paper only.

Future work. Overall, this work is one step towards making the formal verification of security protocols a routine engineering task. We see extending [38] with support for generating machine-checked proofs and exploiting type assertions as a worthwhile next step. It would also be interesting to integrate decryption-chain reasoning with additional reasoning methods, such as authentication tests [19] and abstraction-based overapproximations as used by ProVerif [11] and OFMC [33].

Acknowledgments

This work was supported by the Hasler Foundation within the CompoSec project. We thank Christoph Sprenger and the anonymous reviewers for their comments, which greatly improved the presentation of this paper.

References

[1] M. Arapinis and M. Duflot. Bounding messages for free in security protocols. In *Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science*, FSTTCS'07, pages 376–387, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. M. Borwein and W. M. Farmer, editors, *MKM*, volume 4108 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2006.

[3] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[4] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009.

[5] D. Basin, C. Cremers, and S. Meier. Provably repairing the ISO/IEC 9798 standard for entity authentication. In P. Degano and J. D. Guttman, editors, *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, volume 7215 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2012.

[6] G. Bella. *Formal Correctness of Security Protocols (Information Security and Cryptography)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[7] G. Bella, F. Massacci, and L. C. Paulson. Verifying the SET purchase protocols. *J. Autom. Reasoning*, 36:5–37, 2006.

[8] G. Bella and L. C. Paulson. Kerberos version 4: Inductive analysis of the secrecy goals. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *ESORICS*, volume 1485 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 1998.

[9] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 445–456. ACM, 2010.

[10] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, Oct.–Dec. 2008. Special issue IEEE Symposium on Security and Privacy 2006. Electronic version available at http://doi.ieeecomputersociety.org/10.1109/TDSC.2007.1005.

[11] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[12] F. Blanqui. An Isabelle formalization of protocol-independent secrecy with an application to e-commerce. *CoRR*, abs/cs/0610069, 2006. available as `http://arxiv.org/abs/cs/0610069`.

[13] A. Brucker and S. Mödersheim. Integrating automated and interactive protocol verification. In P. Degano and J. Guttman, editors, *Formal Aspects in Security and Trust*, volume 5983 of *Lecture Notes in Computer Science*, pages 248–262. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-12459-4_18.

[14] E. Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

[15] C. 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.

[16] C. Cremers and S. Mauw. Operational semantics of security protocols. In S. Leue and T. Systä, editors, *Scenarios: Models, Transformations and Tools, International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers*, volume 3466 of *Lecture Notes in Computer Science*. Springer, 2005.

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

[18] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *CSF '08: Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 331–344, Pittsburgh, Pennsylvania, USA, 2008.

[19] S. F. Doghmi, J. D. Guttman, and F. J. Thayer. Searching for shapes in cryptographic protocols. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 523–537. Springer, 2007.

[20] N. A. Durgin, P. Lincoln, and J. C. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[21] N. Evans and S. A. Schneider. Verifying security protocols with PVS: widening the rank function approach. *J. Log. Algebr. Program.*, 64(2):253–284, 2005.

[22] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current trends in hardware verification and automated theorem proving*, pages 387–439. Springer-Verlag New York, Inc., 1989.

[23] J. Goubault-Larrecq. Finite models for formal security proofs. *Journal of Computer Security*, 18(6):1247–1299, 2010.

[24] J. D. Guttman. Authentication tests and disjoint encryption: A design method for security protocols. *Journal of Computer Security*, 12:409–433, 2004.

[25] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. *Journal of Computer Security*, 11(2):217–244, 2003.

[26] International Organization for Standardization, Genève, Switzerland. ISO/IEC 9798-1:2010, Information technology – Security techniques – Entity Authentication – Part 1: General, 2010. Third edition.

[27] B. Jacobs and I. Hasuo. Semantics and logic for security protocols. *Journal of Computer Security*, 17(6):909–944, 2009.

[28] Y. Li, W. Yang, and C.-W. Huang. On preventing type flaw attacks on security protocols with a simplified tagging scheme. *J. Inf. Sci. Eng.*, 21(1):59–84, 2005.

[29] G. Lowe. A hierarchy of authentication specifications. *Computer Security Foundations Workshop, IEEE*, 0:31, 1997.

[30] S. Matsuo, K. Miyazaki, A. Otsuka, and D. A. Basin. How to evaluate the security of real-life cryptographic protocols? - the cases of ISO/IEC 29128 and CRYPTREC. In *Financial Cryptography and Data Security, FC 2010 Workshops, RLCPS, WECSR, and WLC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers*, volume 6054 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 2010.

[31] S. Meier. Source code of the `scyther-proof` tool and the corresponding Isabelle/HOL security protocol verification theory, May 2012. `http://hackage.haskell.org/package/scyther-proof-0.5.0.0`.

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

[33] S. Mödersheim and L. Viganò. The Open-source Fixed-point Model Checker for symbolic analysis of security protocols. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 166–194. Springer, 2009.

[34] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

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

[36] L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Trans. Inf. Syst. Secur.*, 2(3):332–351, 1999.

[37] L. C. Paulson. Relations between secrets: Two formal analyses of the Yahalom protocol. *Journal of Computer Security*, 9(3):197–216, 2001.

[38] B. Schmidt, S. Meier, C. Cremers, and D. A. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Proceedings of the 25rd IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge MA, USA, June 25-27, 2012*. IEEE Computer Society, 2012. to appear.

[39] S. Schneider. Verifying authentication protocols with CSP. *Computer Security Foundations Workshop, IEEE*, 0:3, 1997.

[40] D. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9:47–74, 2001.

[41] SPORE — Security Protocols Open Repository, 2005. `http://www.lsv.ens-cachan.fr/spore`.

[42] F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1), 1999.

[43] M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle framework. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer, 2008.

A Additional Examples of Machine-Checked Security Proofs

In Section 5.1, we explained how we extended Isabelle's proof language to construct machine-checked security proofs based on decryption-chain reasoning. We also showed the formalization of the secrecy proof from Example 5. In this appendix, we also give the formalizations of the authentication proof from Example 6 and the type assertion soundness proof from Example 10. Both of these formalizations were generated automatically using the `scyther-proof` tool. We only took minor liberties in pretty printing them to improve their readability. These examples demonstrate not only that our mechanization of decryption-chain reasoning allows for succinct machine-checked proofs, but also that our automatic algorithm generates human-readable proofs.

Example 15. The statement and the proof of the non-injective synchronization property ϕ_{auth} from Example 6 are formalized in Isabelle/HOL by the proof script given in Figure 14.

Lines 2–11 are a direct translation of the security property ϕ_{auth}. Note that Isabelle stores the conclusion stated in lines 7–11 under the name "?thesis" for later reference. The proof begins in line 12 by applying the TypChain rule to the message received in the second step of the client role C. This message is the instantiation in the thread i of the pattern of the role step C_2, which is available under the name "C_2-pt". The "fake" case in Line 13 corresponds to Case **(1)** from Example 6. This case is discharged by calling Isabelle's built-in tactic "auto" configured to use the previously proven secrecy lemma "client-k-secrecy" and the Known rule. The "S_2-hash" case in Line 16 corresponds to Case **(2)** and denotes that some server role j sent the hash that thread i received in step C_2. In Line 17, as in the pen-and-paper proof, the TypChain rule is applied to the message received by the first message of server j. The necessary applications of the Input and Role rules are handled automatically. The "fake" case in Line 18 corresponds to Case **(2.1)** and is dealt with as before. The case "C_1-enc" in Line 21 corresponds to Case **(2.2)** and denotes that some client i sent the encryption

```
1:  lemma (in CR-state) client-nisynch:
2:    assumes
3:      "(i, $C_2$) ∈ steps(tr)"
4:      "$\text{role}_{\text{th}}$(i) = C"
5:      "σ(AV(′′s′′), i) ∉ Compr"
6:    shows
7:      "∃ j. $\text{role}_{\text{th}}$(j) = S ∧
8:            σ(AV(′′s′′), i) = σ(AV(′′s′′), j)  ∧
9:            NO(′′k′′, i)    = σ(MV(′′v′′), j) ∧
10:           St(i, $C_1$) ≺ St(j, $S_1$)              ∧
11:           St(j, $S_2$) ≺ St(i, $C_2$)"
12: proof(sources "$\text{inst}_{\sigma,\text{i}}$($C_2$-pt)")
13:   case fake thus ?thesis
14:     by (auto dest!: client-k-secrecy[OF known])
15: next
16:   case ($S_2$-hash j) thus ?thesis
17:     proof(sources "$\text{inst}_{\sigma,\text{j}}$($S_1$-pt)")
18:       case fake thus ?thesis
19:         by (auto dest!: client-k-secrecy[OF known])
20:     next
21:       case ($C_1$-enc i) thus ?thesis by auto
22:     qed
23: qed
```

Figure 14: Proof script formalizing the non-injective synchronization proof from Example 6.

received by the server j. In this case, the premises directly imply the conclusion, which corresponds to *syncWith(j)*. Hence, calling "auto" solves this case.

Example 16. The Isabelle/HOL proof script given in Figure 15 formalizes the definition of the type assertion CR'_{ty} for the CR' protocol together with its corresponding soundness proof from Example 10. The type assertion soundness proof is more complicated than the previous proofs due to the required bookkeeping expressed using Isabelle's locale infrastructure [2].

In Lines 1-3, the "type_invariant" command that we implemented is used to define the constant CR'_typing and to create a locale CR'_typing_state that contains all the pre-instantiated variants of the TYPCHAIN rule for the CR' protocol under the assumption that the CR'_typing assertion is sound. The definition of CR'_typing only states the type assertion $(S', v) \to C'.k \cup \mathsf{kn}(S'_1)$, as our formalization of type assertions always assigns the type Ag to agent variables.

In Line 5, the "sublocale" command is used to claim that every lemma proven under the assumption that the type assertion CR'_typing is sound is also a valid lemma without this assumption. This claim depends on the lemma given in Line 7, which states that every reachable state of the CR' protocol is well-typed. The corresponding proof given in Lines 8-15 relies on the tactic "type_soundness" that simplifies applications of Theorem 1. The only non-trivial case is given on Line 9. It handles the instantiation of the variable v in the first step of some thread i executing the server role in the context of some arbitrary well-typed

```
 1: type_invariant CR'_typing for CR'
 2: where "CR'_typing =
 3:   [((S',″v″), SumT (NonceT C' ″k″) (KnownT $S'_1$))]"
 4:
 5: sublocale CR'_state ⊆ CR'_typing_state
 6: proof -
 7:   have "(tr,th,$\sigma$) ∈ well-typed CR'_typing"
 8:   proof(type_soundness "CR'_typing")
 9:     case($S'_1$-v tr′ th′ $\sigma'$ i)
10:       note facts = this
11:       then interpret state: CR'_typing_state tr′ th′ $\sigma'$
12:         by unfold_locales auto
13:       show ?case using facts
14:         by (sources "$\text{inst}_{\sigma',i}$($S'_1$-pt)") auto
15:   qed
16:   thus "CR'_typing_state tr th $\sigma$"
17:     by unfold_locales auto
18: qed
```

Figure 15: Proof script formalizing the definition of the type assertion CR'_{ty} for the CR' protocol and its its corresponding soundness proof from Example 10.

reachable state (tr′,th′,σ'). Lines 10-12 ensure that the "sources" tactic is also applicable in this state. In Line 13-14, we discharge the proof obligations for this case by enumerating all sources for the first message received by thread i and delegating the resulting cases to Isabelle's built-in tactic "auto". In Lines 16-17, we use the lemma from Line 7 to discharge the proof obligation stemming from the "sublocale" command.