Strong Invariants for the Efficient Construction of Machine-Checked Protocol Security Proofs

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

***Abstract*—We embed an operational semantics for security protocols in the interactive theorem prover Isabelle/HOL and derive two strong protocol-independent invariants. These invariants allow us to reason about the possible origin of messages and justify a local typing assumption for the otherwise untyped protocol variables. The two rules form the core of a theory that is well-suited for interactively constructing natural, human-readable, correctness proofs. Moreover, we develop an algorithm that automatically generates proof scripts based on these invariants. Both interactive and automatic proof construction are faster than competing approaches. Moreover, we have strong correctness guarantees since all proofs, including those deriving the underlying theory from the semantics, are machine checked.**

***Keywords*-security protocols, formal methods, theorem proving, automatic tools.**

I. 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. This provides an explanation of why the protocol is correct and makes the result verifiable for others, e.g. for certification purposes. Given the complexity of the symbolic models used and the history of mistakes made in manual protocol proofs, it seems prudent to require *machine-checked proofs*.

In his seminal work [1], Paulson proposed the first approach to constructing machine-checked proofs of protocol correctness, using the interactive theorem prover Isabelle/HOL [2]. The protocols proved include TLS [3], Kerberos IV [4], and SET [5]. As reported in [1], the time required for an expert to model and prove a protocol correct using Paulson's approach ranges from several days for small academic protocols to six weeks for a protocol like TLS [3].

An alternative approach to security protocol verification is to use automatic tools, such as Athena [6], ProVerif [7], or Scyther [8]. Such tools have two main advantages over interactive approaches: they require less user expertise and produce results orders of magnitude faster. However, the state explosion problem makes these tools unsuitable for large protocols.

As originally suggested by Song, Berezin, and Perrig [6], [9], both methods may be combined to verify security protocols. The work by Song et. al. on formalizing the techniques underlying the Athena tool was never completed [10], but aimed at combining the benefits of both approaches: proofs are machine checked, they are generated automatically when possible, and can be developed interactively, if desired.

Approach Taken: We develop a methodology and tool set that combines the benefits described above. First, building on the work by Cremers and Mauw [11], we formalize an untyped execution model for security protocols as an operational semantics in Isabelle/HOL. Then, building on the work by Song et. al., we formally derive two strong protocol-independent invariants from our semantics. The first invariant gives rise to an inference rule for establishing the possible origins of a message. The second invariant characterizes the messages that may be assigned to the untyped protocol variables during execution. This second invariant holds only for a restricted but relevant class of protocols, which we call weakly atomic. Protocols like Yahalom, TLS, and Kerberos V all fall in this class. The benefit of the second invariant is that it allows us to prove strong authentication properties for protocols making use of untyped variables such as Kerberos V. We provide a specialized induction scheme for establishing that a protocol is weakly atomic.

Together, these two invariants form the basis of a verification method for security protocols. We explore two applications: First, we show how these invariants can be used to efficiently construct machine-checked protocol-correctness proofs using interactive theorem proving. Second, by extending an algorithm that builds on Athena and Scyther with weak-atomicity reasoning and support for proof-reuse, we develop a method to automatically generate Isabelle/HOL proof scripts that can be efficiently machine-checked.

Contributions: First, we develop two strong invariants for security protocol verification. These invariants lead to succinct, natural, protocol security proofs.

This work was supported by the Hasler Foundation within the CompoSec project.

Second, the invariants, together with a general protocol theory formalized in Isabelle/HOL, support the *interactive* construction of protocol-correctness proofs almost two orders of magnitude faster than when using Paulson's Inductive Approach; i.e., hours instead of weeks.

Third, we develop a method to *automatically* generate correctness proofs, which can be efficiently machine-checked by Isabelle/HOL. The time needed to generate and machine-check proofs is orders of magnitude faster than the results for the automatic generation of machine-checked proofs reported in [12], [13]. Moreover, the resulting framework, comprising both our Isabelle/HOL protocol theory and the proof-generation algorithm, is 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. Our protocol theory and the invariants are *formally derived* from the 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 our proof-generation algorithm.

Organization: In Section II, we define the protocol model underlying our verification theory. We describe the two invariants and the resulting proof construction method in Section III. In Section IV, we describe interactive proof construction and give our algorithm, which automatically generates machine-checked security proofs. We also report on experimental results in case studies of both interactive and automatic proof construction. We discuss related work in Section V and draw conclusions in Section VI.

II. Security Protocol Model

After some notational preliminaries, we define our security protocol model, which we have formalized in Isabelle/HOL. Our model consists of three parts: (1) a *protocol specification language* based on role scripts and pattern matching, (2) an operational semantics defining *protocol execution* in the presence of an active intruder, and (3) a set of predicates for formalizing *security properties* like secrecy and authentication.

A. Notational Preliminaries

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

For any set S, $\mathcal{P}(S)$ denotes the power set of S and S^* denotes the set of finite sequences of elements from S. We write $\langle s_1, \ldots, s_n \rangle$ to denote the sequence of elements s_1 through s_n. For a sequence s of length $|s|$ and $1 \leq i \leq |s|$, we write s_i to denote the i-th element. We write $s \hat{} s'$ for the concatenation of sequences s and s'. Abusing set notation, we write $e \in s$ iff $\exists i. s_i = e$. We write $x <_s y$ to denote that x precedes y in the sequence s, i.e., $\exists a\, b.\ s = a \hat{} b \wedge x \in a \wedge y \in b$. Note that $<_s$ is a strict total order on the elements in s iff s is duplicate-free.

We use standard notation for manipulating terms [14]. For a term t, the free variables of t are denoted by $FV(t)$.

B. Protocol Specification

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

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

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

The pattern $\mathsf{k}_{a,b}$ denotes a long-term symmetric key shared between a and b, pk_a denotes a's long-term public key, and sk_a denotes a's long-term private key. We use the single encryption constructor $\{|\cdot|\}_k$ to denote public-key encryption when $k = \mathsf{pk}_a$, signing when $k = \mathsf{sk}_a$, and symmetric encryption otherwise. Note that we allow for composed messages to be used as symmetric keys. The constructor $\mathsf{h}(\cdot)$ denotes hashing and $(\cdot, \cdot)$ denotes pairing.

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 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

$$\forall\, \mathsf{Send}_l(pt) \in R.\ \forall v \in FV(pt) \cap MVar.\\ \exists l', pt'.\ \mathsf{Recv}_{l'}(pt') <_R \mathsf{Send}_l(pt) \wedge v \in FV(pt')\ .$$

Hence, in a role, every message variable must be received before it can be sent. 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 specification with a simple challenge-response protocol.

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

$$C \stackrel{\text{def}}{=} \langle\, \mathsf{Send}_1(\{|k|\}_{\mathsf{pk}_s}),\quad \mathsf{Recv}_2(\mathsf{h}(k))\,\rangle$$
$$S \stackrel{\text{def}}{=} \langle\, \mathsf{Recv}_1(\{|v|\}_{\mathsf{pk}_s}),\quad \mathsf{Send}_2(\mathsf{h}(v))\,\rangle\ .$$

In this protocol, a client, modeled by the *C* role, chooses a fresh session key *k* and sends it encrypted with the public key 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 the ones defined here.

C. 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 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 also has access to the long-term keys of an unbounded number of 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 [11]. The ingredients of the operational semantics are (1) messages, (2) agent threads, (3) the system state, (4) the intruder knowledge, and finally (5) the transition system. We discuss each of these in turn.

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 *tid* and a message $n \in Fresh$ to be freshly generated, we write $n\sharp tid$ to denote the actual fresh message generated by the thread *tid* for n. We overload notation and for A a set, we write $A\sharp TID$ to denote the set $\{a\sharp tid \mid a \in A, tid \in TID\}$.

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

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

A message m is *atomic* iff $m \in Const \cup Fresh\sharp TID \cup Agent$. 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.

2) Agent threads: For each thread, the system state stores the role it executes and the role steps still to be executed. We model this information as a partial function

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

where $dom(th)$ denotes the identifiers of all threads in the system. We call *th* a *thread pool* and define the set *ThreadPool* as the set of all thread pools. Furthermore, the system state contains a *variable store* $\sigma : Var \times TID \to Msg$ storing for each variable v and thread identifier *tid* the contents $\sigma(v, tid)$ assigned to v by thread *tid*. We define the set of all variable stores as $Store \stackrel{\text{def}}{=} Var \times TID \to Msg$.

Threads execute roles, which are sequences of role steps, which specify the messages to be sent and received as message patterns. As we will see later, we abstract from the process of instantiating variables when receiving messages by considering all possible assignments of messages to a thread's variables. During the execution of a thread *tid* and in the context of a variable store σ, a message pattern *pt* is *instantiated* to the message $inst_{\sigma,tid}(pt)$ by replacing all fresh message patterns with the actual fresh messages and all variables with the content assigned to them by thread *tid*.

$$inst_{\sigma,tid}(pt) \stackrel{\text{def}}{=} \begin{cases} pt & \text{if } pt \in Const \\ pt\sharp tid & \text{if } pt \in Fresh \\ \sigma(pt, tid) & \text{if } pt \in Var \\ \mathsf{h}(inst_{\sigma,tid}(x)) & \text{if } pt = \mathsf{h}(x) \\ (inst_{\sigma,tid}(x), inst_{\sigma,tid}(y)) & \text{if } pt = (x, y) \\ \{|inst_{\sigma,tid}(x)|\}_{(inst_{\sigma,tid}(k))} & \text{if } pt = \{|x|\}_k \\ \mathsf{k}_{inst_{\sigma,tid}(a),\, inst_{\sigma,tid}(b)} & \text{if } pt = \mathsf{k}_{a,b} \\ \mathsf{pk}_{inst_{\sigma,tid}(a)} & \text{if } pt = \mathsf{pk}_a \\ \mathsf{sk}_{inst_{\sigma,tid}(a)} & \text{if } pt = \mathsf{sk}_a \end{cases}$$

3) System state: The system state keeps track of the thread pool, the variable contents, and a trace recording what role steps were executed by what thread and what messages were learned by the intruder. We use this trace both to keep track of the intruder knowledge in a system state as well as to specify the security properties of a protocol. A *trace* is a sequence of *basic events*.

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

The *basic step event* $\mathsf{St}(tid, s)$ denotes that the thread *tid* executed the role step s. The *basic learn event* $\mathsf{Ln}(M)$ denotes that the intruder learned the messages M. We define

$$\frac{th(tid) = (R, \langle \mathsf{Send}_l(pt)\rangle \mathbin{\hat{}} todo)}{(tr, th, \sigma) \longrightarrow (tr \mathbin{\hat{}} \langle\, \mathsf{St}(tid, \mathsf{Send}_l(pt)),\, \mathsf{Ln}(newMsgs_{tr}(inst_{\sigma,tid}(pt)))\,\rangle,\ th[tid \mapsto (R, todo)],\ \sigma)}\ \textsc{Send}$$

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

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

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

Figure 1. Transition rules of the execution model.

the set of all traces as $Trace \stackrel{\text{def}}{=} BasicEvent^*$. Finally, a *system state* is a triple $(tr, th, \sigma) \in Trace \times ThreadPool \times Store$.

4) Intruder knowledge: We assume that there exists a subset $Compr \subseteq Agent$ of compromised agents whose long-term keys are known to the intruder. Thus, the intruder can impersonate any agent $Eve \in Compr$ and act as a legitimate participant of a protocol. The *initial intruder knowledge* IK_0 is therefore defined as

$$IK_0 \stackrel{\text{def}}{=} Const \cup Agent \cup \bigcup_{a \in Agent, Eve \in Compr} \{\mathsf{pk}_a, \mathsf{sk}_{Eve}, \mathsf{k}_{a,Eve}, \mathsf{k}_{Eve,a}\}\ .$$

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

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

Our execution model ensures that the intruder always learns the initial intruder knowledge as the first basic event in a trace. Furthermore, whenever the intruder knows a tuple (x, y), then he also knows the messages x and y. To formalize this invariant, we define the function $split : Msg \to \mathcal{P}(Msg)$ such that $split(m)$ denotes the set of all messages that can be obtained from m using projection.

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

Intuitively, the intruder can *learn* a message only once. We use $newMsgs_{tr}(m)$ to denote the *new messages* learned when seeing a message m in the context of a trace tr.

$$newMsgs_{tr}(m) \stackrel{\text{def}}{=} split(m) \setminus knows(tr)$$

Example 2 (System state of the *CR* protocol)**.** Assume 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 some $\sigma' \in Store$ and

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

5) Transition system: For a protocol P, the *state transition relation* $\longrightarrow$ is defined by the transition rules in Figure 1. We explain each rule in turn.

A SEND transition is enabled whenever the next step of a thread tid is $\mathsf{Send}_l(pt)$ for some label l and some message pattern pt. The trace tr is extended with two basic events. The basic event $\mathsf{St}(tid, \mathsf{Send}_l(pt))$ records that this send step has happened. The basic event $\mathsf{Ln}(newMsgs_{tr}(inst_{\sigma,tid}(pt)))$ records that the intruder learns all *new* messages accessible from the sent message $inst_{\sigma,tid}(pt)$ using projection.

A RECV transition is enabled whenever the next step of a thread tid 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 basic event $\mathsf{St}(tid, \mathsf{Recv}_l(pt))$, 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 construction. Because the intruder knowledge is closed under *split*, no projection transition is needed.

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

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

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

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 message variables are instantiated with some arbitrary message; i.e., we overapproximate 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 *tid* $\in$ *dom*(*th*) instantiates a role of P and has not executed any step yet.

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

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

Using the above definition, we can formalize trace-based security properties that are expressible as invariants over the set of reachable states of a security protocol.

D. Security Properties

We focus on security properties expressible as a *security predicate* ϕ that must hold for every state $(tr, th, \sigma) \in \mathit{reachable}(P)$. Many security properties from literature fit this pattern, for example, all the authentication properties from [15], [16] or secrecy as in [11].

To formalize and reason about security properties, we introduce the notion of an *event*, which is either a *learn event* m denoting that the intruder learned the message m or a *step event* (tid, s) denoting that thread *tid* has executed the role step s. We define

$$\mathit{Event} \stackrel{\text{def}}{=} \mathit{Msg} \cup (\mathit{TID} \times \mathit{RoleStep})$$

as the set of all events. The difference between events and basic events is that a basic learn event denotes the learning of a set of messages, while a learn event denotes the learning of a single message. We make this distinction, as the use of basic learn events simplifies our semantics, while the restriction to learning single messages simplifies the inference rules that we use to reason about protocols.

The previously defined function *knows* can 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 denoted by *steps*(*tr*).

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

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

$$\begin{aligned} x \prec_{tr} y \stackrel{\text{def}}{=} \exists tr_1\ tr_2.\ tr = tr_1 \hat{}\ tr_2 \wedge \\ (x \in \mathit{knows}(tr_1) \vee x \in \mathit{steps}(tr_1)) \wedge \\ (y \in \mathit{knows}(tr_2) \vee y \in \mathit{steps}(tr_2)) \end{aligned}$$

Our semantics guarantees that $\prec_{tr}$ is a strict partial order on *Event*. We define $\preceq_{tr}$ as the reflexive closure of $\prec_{tr}$.

The explicit representation of the order between executed steps and messages known by the intruder is a key difference between our work and Paulson's Inductive Approach [1]. It allows us, for example, to represent the statement "both the encryption $\{|m|\}_k$ and the inverse key k^{-1} must have been known before the intruder learned m" as the proposition

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

We will make extensive use of such propositions in our security proofs.

We also define the partial function $\mathit{role}_{th} : \mathit{TID} \nrightarrow \mathit{Role}$, where $\mathit{role}_{th}(\mathit{tid}) = R$ denotes that thread *tid* executes an instance of role R.

$$\mathit{role}_{th}(\mathit{tid}) \stackrel{\text{def}}{=} \begin{cases} R & \text{if } \mathit{th}(\mathit{tid}) = (R, \mathit{todo}) \\ \bot & \text{if } \mathit{th}(\mathit{tid}) = \bot \end{cases}$$

Security predicates 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 [16] (a strengthened variant of non-injective agreement [15]) with a server. We formalize property (1) as the security predicate ϕ_{sec}.

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

Recall that C, s, and k, as well as S and v, have been 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 who received the first message from client i and whose second message was received by client i. We formalize this as the predicate ϕ_{auth}.

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

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

III. Security Proofs based on Decryption Chains

In the last section, we described the embedding of our operational semantics in higher-order logic. This is sometimes called a *shallow embedding* [17] in the verification community as the operational semantics is given by a (conservative) definitional extension of higher-order logic. From the semantics, we derive inference rules that directly encode reasoning principles for constructing protocol security proofs. This derivation is carried out in Isabelle/HOL, thereby giving us a machine-checked proof of the soundness of our rules with respect to the operational semantics.

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

$$\frac{m \prec_{\mathit{tr}} e}{m \in \mathit{knows}(\mathit{tr})}\ \textsc{Known} \qquad \frac{(\mathit{tid}, s) \prec_{\mathit{tr}} e}{(\mathit{tid}, s) \in \mathit{steps}(\mathit{tr})}\ \textsc{Exec} \qquad \frac{e \prec_{\mathit{tr}} e}{\mathit{false}}\ \textsc{Irr} \qquad \frac{e_1 \prec_{\mathit{tr}} e_2 \qquad e_2 \prec_{\mathit{tr}} e_3}{e_1 \prec_{\mathit{tr}} e_3}\ \textsc{Trans}$$

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

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

$$\frac{m \in \mathit{knows}(\mathit{tr})}{\begin{array}{l}(m \in \mathit{IK}_0) \vee \\ (\exists x.\quad m = \mathsf{h}(x) \quad \wedge x \prec_{\mathit{tr}} \mathsf{h}(x) \quad) \vee \\ (\exists x\ k.\ m = \{\!|x|\!\}_k \ \wedge x \prec_{\mathit{tr}} \{\!|x|\!\}_k \ \wedge k \prec_{\mathit{tr}} \{\!|x|\!\}_k \ ) \vee \\ (\exists x\ y.\ m = (x, y) \ \wedge x \prec_{\mathit{tr}} (x, y) \ \wedge y \prec_{\mathit{tr}} (x, y) \ ) \vee \\ (\exists R \in P.\ \exists\, \mathsf{Send}_l(\mathit{pt}) \in R.\ \exists \mathit{tid}.\ \mathit{role}_{\mathit{th}}(\mathit{tid}) = R \wedge \\ \qquad \mathit{chain}_{\mathit{tr}}(\{(\mathit{tid}, \mathsf{Send}_l(\mathit{pt}))\},\ \mathit{inst}_{\sigma,\mathit{tid}}(\mathit{pt}),\ m)\ )\end{array}}\ \textsc{Chain}$$

Figure 2. Core inference rules of the decryption-chain reasoning technique.

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

Figure 3. Definition of the *chain* predicate using recursion over the message m' in the second argument.

We present our inference rules in two parts. First, in Section III-A, we present the core inference rules. We illustrate their usage on the *CR* protocol. Second, in Section III-B, we show why these rules are insufficient for reasoning about the content of variables due to the untyped model that we are using. We then provide a solution to this problem for a practically relevant class of protocols.

A. Core Inference Rules

The core inference rules are given in Figure 2. We derive all of these rules from our semantic embedding under the assumption that $(\mathit{tr}, \mathit{th}, \sigma) \in \mathit{reachable}(P)$.

The rules $\textsc{Kn}_1$ and $\textsc{Kn}_2$ express 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 $\textsc{Ord}_1$ and $\textsc{Ord}_2$ express 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 tuples to the knowledge of the contained nonces, hashes, and encryptions. Intuitively, these rules are sound because our semantics ensures that the intruder knowledge is closed under *split*.

The rules KNOWN and EXEC follow trivially from the definition of $\prec_{\mathit{tr}}$, *knows*, and *steps*.

The rules IRR and TRANS express that $\prec_{\mathit{tr}}$ is a strict partial order. Intuitively, they 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 expresses that if a thread *tid* 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 *tid*. Intuitively, this rule is sound because both the SEND and the RECV transitions successively execute role steps in the order specified by the roles.

The rule INPUT expresses that if a thread *tid* has executed a receive step $\mathsf{Recv}_l(\mathit{pt})$, then the instantiated pattern *pt* has been learned before $\mathsf{Recv}_l(\mathit{pt})$ was executed by the thread *tid*. Intuitively, this rule is sound because the RECV transition ensures that the intruder knows the received message.

The rule CHAIN expresses 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.
3) m is an encryption $\{\!|x|\!\}_k$ of a known message x with a known key k and the intruder built $\{\!|x|\!\}_k$ himself.
4) m is a pair (x, y) of two known messages x and y and the intruder built (x, y) himself.
5) There was some send step $\mathsf{Send}_l(\mathit{pt})$ executed by some thread *tid* such that the intruder learned the sent message $\mathit{inst}_{\sigma,\mathit{tid}}(\mathit{pt})$ and from this message he learned m using zero or more decryptions and projections.

The key insight for 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*. Anal-

ogously, 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 3. 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 equal to the message m and the intruder has learned m' after the events in E, or
2) m' is an encryption $\{|x|\}_k$ and the intruder has 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 has 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 earlier than the tuple (x, y).

We illustrate the usage of our inference system by giving a formal proof of session-key secrecy for the *C* role of the *CR* protocol from Example 1.

Example 4 (Proof of session-key secrecy)**.** We prove that $\forall q \in reachable(CR).\ \phi_{\text{sec}}(q)$ for ϕ_{sec} from Example 3.

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

We show that each possible way that the intruder could learn $k\sharp i$ leads to a contradiction. We determine the possible ways by applying the CHAIN rule to $k\sharp i \in knows(tr)$. This results in the following conclusion, whose disjuncts we have numbered.

(1) $(k\sharp i \in IK_0) \vee$
(2) $(\exists x.\quad k\sharp i = \mathsf{h}(x) \ \wedge x \prec_{tr} \mathsf{h}(x) \qquad) \vee$
(3) $(\exists x\ k.\ k\sharp i = \{|x|\}_k \wedge x \prec_{tr} \{|x|\}_k \wedge k \prec_{tr} \{|x|\}_k) \vee$
(4) $(\exists x\ y.\ k\sharp i = (x, y) \wedge x \prec_{tr} (x, y) \wedge y \prec_{tr} (x, y)) \vee$
(5) $(\exists R \in CR.\ \exists\, \mathsf{Send}_l(pt) \in R.\ \exists tid.\ role_{th}(tid) = R \wedge$
$chain_{tr}(\{(tid, \mathsf{Send}_l(pt))\},\ inst_{\sigma,tid}(pt),\ k\sharp i))$

Cases **(1-4)** are false due to the definition of IK_0 and syntactic inequality. This corresponds to the intuition that $k\sharp i$ is a *fresh* session key.

Case **(5)** unfolds after renaming *tid* to j as follows.

(5.1) $(\exists j \in TID.\ role_{th}(j) = S \wedge$
$chain_{tr}(\{(j, S_2)\},\ \mathsf{h}(\sigma(v, j)),\ k\sharp i))$
(5.2) $\vee\ (\exists j \in TID.\ role_{th}(j) = C \wedge$
$chain_{tr}(\{(j, C_1)\},\ \{|k\sharp j|\}_{\mathsf{pk}_{\sigma(s,j)}},\ k\sharp i))$

Unfolding the *chain* predicate in Case **(5.1)** results in a single conjunct containing $\mathsf{h}(\sigma(v, j)) = k\sharp i$, which is false.

Unfolding the *chain* predicate in Case **(5.2)** results in

(5.2.1) $((j, C_1) \prec_{tr} \{|k\sharp j|\}_{\mathsf{pk}_{\sigma(s,j)}} \wedge \{|k\sharp j|\}_{\mathsf{pk}_{\sigma(s,j)}} = k\sharp i)$
(5.2.2) $\vee\ ((j, C_1) \prec_{tr} \{|k\sharp j|\}_{\mathsf{pk}_{\sigma(s,j)}} \prec_{tr} k\sharp j \wedge$
$\mathsf{sk}_{\sigma(s,j)} \prec_{tr} k\sharp j \wedge k\sharp j = k\sharp i)$.

Case **(5.2.1)** is false due to $\{|k\sharp j|\}_{\mathsf{pk}_{\sigma(s,j)}} \neq k\sharp i$.
Case **(5.2.2)** implies that $k\sharp j = k\sharp i$, which implies $j = i$ due to the injectivity of the message constructor $\sharp$. From $\mathsf{sk}_{\sigma(s,j)} \prec_{tr} k\sharp i$ and $j = i$, we conclude $\mathsf{sk}_{\sigma(s,i)} \in knows(tr)$ using the KNOWN rule. Applying the CHAIN rule and removing trivial cases results in the conclusion $\mathsf{sk}_{\sigma(s,i)} \in IK_0$. This corresponds to the intuition that, for the *CR* protocol, the only long-term private keys known to the intruder are the ones he initially knows. By unfolding the definition of IK_0, we have $\sigma(s, i) \in Compr$, which contradicts our assumptions.

Thus, we conclude that ϕ_{sec} holds for all reachable states of the *CR* protocol. ■

The previous example shows how to prove a secrecy property using decryption-chain reasoning. The general approach for secrecy proofs is to use the CHAIN rule both for the message m to be proven secret as well as for the keys that must be secret if m is not to be decrypted. 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 that sent m. Furthermore, if a message is authentic because it contains a secret n that only the partner knows, then the authentication proof depends on the secrecy proof of n. An example of such an authentication proof can be found in Appendix A. We can make such dependencies among proofs explicit and *reuse* previously proven security properties by instantiating their corresponding lemmas instead of reproving them each time they are needed. We will explain this in more detail in Section IV-B3.

B. Reasoning about Variable Contents

During the verification of most protocols, we will encounter expressions of the form

$$chain_{tr}(E,\ \sigma(v, j),\ m)\ ,$$

where $v \in Var$, $j \in TID$, $E \subseteq Event$, and $m \in Msg$. These expressions arise from an application of the CHAIN rule and unfolding the definition of the *chain* predicate as

$$\frac{v \in \mathit{AVar} \qquad \mathit{chain}_{tr}(E,\, \sigma(v, \mathit{tid}),\, m)}{E = \emptyset}\ \textsc{AgentVar} \qquad \frac{\begin{array}{cc}\mathit{wa\text{-}state}(tr, th, \sigma) & \exists e \in E.\ (\mathit{tid}, s) \preceq_{tr} e \\ v \in \mathit{FV}(s) \cap \mathit{MVar} & \mathit{chain}_{tr}(E,\, \sigma(v, \mathit{tid}),\, m)\end{array}}{(\forall e \in E.\ e \prec_{tr} \sigma(v, \mathit{tid})) \wedge \sigma(v, \mathit{tid}) = m}\ \textsc{Atomic}$$

Figure 4. Inference rules for reasoning about variable contents. They are derived from our semantics under the assumption $(tr, th, \sigma) \in \mathit{reachable}(P)$.

far as possible. Obviously, the core inference rules do not suffice for dealing with these expressions.

In this section, we present two additional inference rules, one for agent variables and one for message variables, that allow us to reduce these expressions to ones that can again be handled by the core inference rules.

1) Agent Variables: If v is an agent variable, then $\sigma(v, j)$ must be an agent name. Agent names are part of IK_0 and thus learned by the intruder at the beginning of every trace before any other event happens. Therefore, $\mathit{chain}_{tr}(E,\, \sigma(v, j),\, m)$ implies $E = \emptyset$ because otherwise we could derive $\sigma(v, j) \prec_{tr} \sigma(v, j)$, which is a contradiction. The rule AGENTVAR in Figure 4 captures this argument.

2) Message Variables: If v is a message variable, then we need a more elaborate argument than for agent variables because $\sigma(v, j)$ may be any message.

The argument used by many security protocol verification methods is to simply *assume* that variables are of a fixed type. If we were to do the same, then we could replace $\sigma(v, j)$ by an arbitrary message corresponding to the type of v, and finish unfolding the definition of the *chain* predicate.

The soundness such a typing assumption may in some cases be justified by results such as [18]–[20]. However, these results restrict both the protocols (tagging is required) and the properties (the soundness of the typing assumption is proven per property) that can be verified. Moreover, it is unclear how to extend these results to equational theories.

In our approach, we do not make any typing assumptions.[1] Instead, we prove for each protocol that it satisfies a typing invariant that specifies all possible instantiations for every message variable that occurs in an executed step. This approach allows us to construct machine-checked security proofs that are sound with respect to an untyped model without explicitly formalizing results like [18]–[20].

We focus on protocols where, in every execution, the message variables are instantiated with an atomic message or a message known to the intruder. We call such protocols *weakly atomic*. Many protocols in literature are weakly atomic. Note that weak atomicity does not rule out forwarding encrypted tickets as plaintext (e.g., Kerberos V uses such a construction) because these tickets will always be known to the intruder. An example of a protocol that is not weakly atomic is Kerberos IV, as it requires the use of an encrypted ticket inside an encryption. In general, protocols in literature that are not weakly atomic are protocols that require an agent to receive a composed message in a message variable inside an encryption.

In the rest of this section, we first define weak atomicity. Then, we show how to prove that a protocol is weakly atomic. Finally, we show how to exploit weak atomicity to reduce the expression $\mathit{chain}_{tr}(E,\, \sigma(v, j),\, m)$ to one that can be handled again using the core inference rules.

Formalizing Weak Atomicity: We generalize the notion of free variables to role steps such that $\mathit{FV}(s)$ denotes the free variables of the message pattern of the role step s.

$$\mathit{FV}(s) \stackrel{\text{def}}{=} \begin{cases} \mathit{FV}(pt) & \text{if } s = \mathsf{Send}_l(pt) \\ \mathit{FV}(pt) & \text{if } s = \mathsf{Recv}_l(pt) \end{cases}$$

A state (tr, th, σ) is *weakly atomic* iff every message variable v mentioned in some executed step (tid, s) is instantiated with either a fresh message or a message known to the intruder before (tid, s) was executed.

$$\begin{aligned} &\mathit{wa\text{-}state}(tr, th, \sigma) \stackrel{\text{def}}{=} \\ &\quad \forall (\mathit{tid}, s) \in \mathit{steps}(tr).\ \forall v \in \mathit{FV}(s). \\ &\qquad \sigma(v, \mathit{tid}) \in \mathit{Fresh} \sharp \mathit{TID} \vee \sigma(v, \mathit{tid}) \prec_{tr} (\mathit{tid}, s) \end{aligned}$$

The above definition differs in two ways from the informal definition of weak atomicity given above. These differences do not exclude any protocols, but simplify exploiting weak atomicity. The differences are the following. (1) We restrict $\sigma(v, \mathit{tid})$ not only to atomic messages, but to *fresh* messages or some message known to the intruder. This does not exclude any protocols, as the other two types of atomic messages, global constants and agent names, are always known to the intruder. (2) Instead of only requiring that $\sigma(v, \mathit{tid})$ is known to the intruder, we require that it is learned before any step of the thread tid mentioning v. This also does not exclude any protocols, as the first step (s, tid) that mentions v is always a receive step. Hence, everything that the intruder knows when (s, tid) is executed must have been learned before (s, tid) happened.

We define a protocol P to be *weakly atomic* iff all its reachable states are weakly atomic.

Proving Weak Atomicity: We prove that a protocol P is weakly atomic by proving for all message variables occurring in a role of P that, at the time of their first receipt, their content is either already known by the intruder or it is a fresh message.

Theorem 1 (Weak Atomicity)**.** *Let P be a protocol. If, for every $(tr, th, \sigma) \in \mathit{reachable}(P)$, $v \in \mathit{MVar}$, $\mathit{tid} \in \mathit{dom}(th)$,*

[1] We do restrict agent variables to contain agent names only. However, agent variables are used only to parametrize roles. Message variables, which are untyped, are used to *receive* agent names.

$R \in P$, $\mathsf{Recv}_l(pt) \in R$, and sequences of role steps done and todo, the assumptions

- $\mathit{inst}_{\sigma,tid}(pt) \in \mathit{knows}(tr)$
- $\mathit{th}(tid) = (R, \langle \mathsf{Recv}_l(pt) \rangle \hat{} \mathit{todo})$
- $R = \mathit{done} \hat{} \langle \mathsf{Recv}_l(pt) \rangle \hat{} \mathit{todo}$
- $v \in \mathit{FV}(pt) \setminus \bigcup_{s \in \mathit{done}} \mathit{FV}(s)$
- $\mathit{wa\text{-}state}(tr, th, \sigma)$

imply

$$\sigma(v, tid) \in \mathit{knows}(tr) \vee \sigma(v, tid) \in \mathit{Fresh} \sharp \mathit{TID} \ ,$$

then the protocol P is weakly atomic.

The proof obligations resulting from an application of the above theorem are proven using decryption-chain reasoning starting from the premise $\mathit{inst}_{\sigma,tid}(pt) \in \mathit{knows}(tr)$.

Exploiting Weak Atomicity: For a weakly atomic state (tr, th, σ), we use the following argument, captured by the ATOMIC rule in Figure 4, to reduce expressions of the form $\mathit{chain}_{tr}(E, \sigma(v, j), m)$ for $v \in \mathit{MVar}$ to expressions that can be handled again by the core inference rules.

First, note that such a *chain* expression must stem from the conclusion of an application of the CHAIN rule. Hence, there will always be an event $e \in E$ and an executed step $(j, s) \in \mathit{steps}(tr)$ with $v \in \mathit{FV}(s)$ such that $(j, s) \preceq_{tr} e$. Exploiting weak atomicity, we have that $\sigma(v, j)$ is either **(1)** a fresh message or **(2)** a message known to the intruder before (j, s) was executed.
In case **(1)**, we can just unfold $\mathit{chain}_{tr}(E, \sigma(v, j), m)$ to

$$(\forall e \in E.\ e \prec_{tr} \sigma(v, j)) \wedge \sigma(v, j) = m \ .$$

In case **(2)**, we have

$$\exists e \in E.\ \sigma(v, j) \prec_{tr} (j, s) \preceq_{tr} e \wedge \mathit{chain}_{tr}(E, \sigma(v, j), m) \ ,$$

which contradicts the irreflexivity of $\prec_{tr}$.

Discussion of Weak Atomicity: The rules AGENTVAR and ATOMIC together with Theorem 1 obviate the need for a typed protocol model. Together with the core inference rules, they give rise to a reasoning technique, which we call *decryption-chain reasoning*. Decryption-chain reasoning suffices for verifying many weakly atomic protocols; we give examples in our case studies in Section IV-B3.

An example of an (artificial) weakly atomic protocol where decryption-chain reasoning fails is

$$\{\underbrace{\langle \mathsf{Send}_1(\{\!|n|\!\}_{\mathsf{k}_{a,b}}) \rangle}_{I}, \underbrace{\langle \mathsf{Recv}_1(\{\!|v|\!\}_{\mathsf{k}_{a,b}}), \mathsf{Send}_2(\{\!|v|\!\}_{\mathsf{k}_{a,b}}) \rangle}_{R}\} \ ,$$

for $n \in \mathit{Fresh}$, $v \in \mathit{MVar}$, and $a, b \in \mathit{AVar}$. Here, the contents of variable v in role R are obviously secret, provided both a and b are uncompromised. However, we cannot prove this using decryption-chain reasoning. Whenever, we determine, using the CHAIN rule, the possible sources of the message m that a thread executing the R role receives in its first step, we are left with a case stating that m was sent in the second step of some other thread also executing the R role. Put differently: decryption-chain reasoning fails to capture the argument that there must be a *first* occurrence of the role step R_1 with uncompromised a and b that must have received its corresponding message from a thread executing the I role.

Decryption-chain reasoning, as we defined it here, is tailored towards arguments relying on the first time the intruder knows a certain message. To formalize arguments relying on the first occurrence of a more general situation (e.g., the first execution of a certain role step), we must resort to induction over the reachable states and use decryption-chain reasoning in the individual induction steps.

However, even this will not work for all protocols due to the following reason. The undecidability proofs [21] for the secrecy problem with unbounded nonces also apply to weakly atomic protocols. Moreover, for a secrecy predicate, both counter-examples (attacks) as well as proofs based on decryption-chain reasoning are recursively enumerable. As decryption-chain reasoning is sound, it cannot be complete.

In order to verify protocols that are not weakly atomic, the construction underlying weak atomicity can be generalized by stating for each protocol variable what the message structure (the "type") of its content must be, if the intruder does *not* know the content. A generalization of Theorem 1 can then be proven that provides a specialized induction scheme for "type checking" a protocol's message variables.

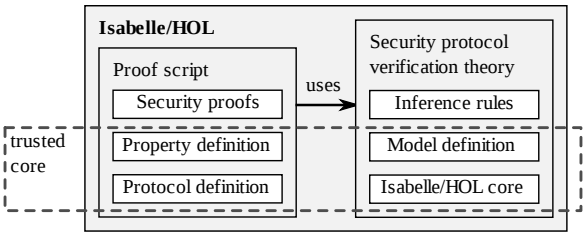

Figure 5. Machine-checked security proofs in Isabelle/HOL: elements and trusted core

IV. MACHINE-CHECKED SECURITY PROOFS

We formalized our protocol model and derived the inference rules in Isabelle/HOL [2]. This allows us to construct machine-checkable security proofs in the form of Isabelle proof scripts as depicted in Figure 5. If Isabelle successfully checks such a proof script, then this constitutes a security proof whose correctness only depends on the trusted core shown in Figure 5. In particular, no further trust assumptions are required: both the inference rules and concrete protocol proofs are machine-checked with respect to the trusted core.

We implemented two approaches for constructing such machine-checked proof scripts. The first approach consists of interactively constructing the proof script using Isabelle. The second approach uses an automatic proof-generation algorithm. We discuss both approaches below.

A. Interactive Proof Construction

In order to simplify the interactive construction of security proofs, we extended Isabelle's proof language [22] with a proof method called "sources", which automates single applications of the CHAIN rule. A call "sources m" applies the CHAIN rule to $m \in \mathit{knows}(\mathit{tr})$, converts the resulting conclusion into disjunctive normal form, and discharges all cases that trivially follow from the definition of IK_0 or syntactic inequality. The remaining cases are named and presented to the user for further input. As we can see in the following example and in the proofs of our case studies [23], this mechanization of decryption-chain reasoning allows for succinct, machine-checkable security proofs.

Example 5. The session-key secrecy proof given in Example 4 corresponds to the following proof script, which is checked by Isabelle in 0.5 seconds. Note that we have taken minor liberties in pretty-printing to improve readability.

```
1:  lemma (in CR-state) client-k-secrecy:
2:    assumes
3:      "role_th(tid) = C"
4:      "σ(AV(''s''), tid) ∉ Compr"
5:      "LN(''k'', tid) ∈ knows(tr)"
6:    shows "False"
7:  proof(sources "LN(''k'', tid)")
8:    case C_1-k thus "False"
9:    proof(sources "SK (σ(AV(''s''), tid))")
10:     case ik0 thus "False" by auto
11:   qed
12: qed
```

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 constructor "LN" marks an actual fresh message, while the constructor "AV" marks an agent variable.

Lines 7–12 give the proof, which has the same structure as the pen-and-paper proof from Example 4. Line 7 applies the CHAIN rule to determine the possible sources of $k\sharp\mathit{tid}$. Line 8 selects the non-trivial case named "C_1-k" corresponding to Case **(5.2.2)** of Example 4 and claims that it is contradictory. Line 9 applies the CHAIN rule again. Line 10 uses Isabelle's built-in tactic "auto" to show that the case "ik0" is contradictory because it contains contradictory assumptions about whether agents are compromised.

To asses the effectiveness of decryption-chain reasoning, we interactively constructed security proofs for several protocols. All of these proofs are available at [23]. These include security proofs for the Yahalom [24], Kerberos V [25], and TLS [3] protocols based on the models developed using the Inductive Approach [1]. Modeling the protocol took under an hour per protocol. Proving the security properties took 1.5 hours for Yahalom, 2 hours for Kerberos V, and 2.5 hours for TLS. These times represent roughly a two orders of magnitude improvement over the Inductive Approach, as we will see in Section V.

B. Automatic Proof Generation

We describe an algorithm for automatically generating machine-checkable proof scripts. The input of the algorithm is a protocol $P \in \mathit{Protocol}$ and a security property ϕ. If the algorithm succeeds in proving ϕ, then it outputs an Isabelle proof script containing the protocol specification of P and a lemma stating ϕ together with its proof.

We present our algorithm in three steps. First, we describe the basic proof generation algorithm. Second, we extend it to prove weak atomicity. Third, we describe two extensions that greatly increase its efficiency and the readability of the generated proofs. Afterwards, we present experimental results for case studies.

1) Basic Algorithm: The algorithm handles security properties that are representable as closed formulas of the form

$$\forall(\mathit{tr}, \mathit{th}, \sigma) \in \mathit{reachable}(P).\ \forall \mathit{tid}_1 \ldots \mathit{tid}_l.\ (\bigwedge\nolimits_{A\in\Gamma} A) \Rightarrow \exists \mathit{tid}_{l+1} \ldots \mathit{tid}_n.\ (\bigwedge\nolimits_{B\in\Delta} B)$$

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

$\mathit{tid} = \mathit{tid}'$	$m = m'$	$\mathit{role}_{\mathit{th}}(\mathit{tid}) = R$
$e \prec_{\mathit{tr}} e'$	$(\mathit{tid}, s) \in \mathit{steps}(\mathit{tr})$	$m \in \mathit{knows}(\mathit{tr})$
false	$\sigma(a, \mathit{tid}) \in \mathit{Compr}$	$\sigma(a, \mathit{tid}) \notin \mathit{Compr}$

We denote such a security property by a *judgment* $\Gamma \vdash_P \Delta$, where 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 6. The secrecy property ϕ_{sec} from Example 3 is represented by the judgment

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

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

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

The basic proof-generation algorithm PRFGEN is given in Figure 6. It takes a judgment $\Gamma \vdash_P \Delta$ and attempts to prove its validity as follows.

First, the equations in Γ are viewed as rewrite rules and normalized such that (i) left-hand sides are of the form tid, $\sigma(v, \mathit{tid})$, or $\mathit{role}_{\mathit{th}}(\mathit{tid})$, (ii) right-hand sides cannot be rewritten any further, and (iii) rules of the form $\mathit{tid} = \mathit{tid}'$ and $\sigma(v, \mathit{tid}) = \sigma(v', \mathit{tid}')$ are oriented according to a fixed term order. The normalized rules are then used to rewrite all other atoms in $\Gamma \vdash_P \Delta$. As a final step, equality atoms of the form $x = x$ are removed from Δ. Moreover, if normalization is not possible due to an equality equating syntactically

```
1: procedure PrfGen(Γ ⊢_P Δ)
2:     normalize Γ ⊢_P Δ with respect to equalities in Γ
3:     saturate Γ under all rules except Chain
4:     if Γ ⊢_P Δ is trivially valid then
5:         print "by auto"
6:     else
7:         select new (m ∈ knows(tr)) ∈ Γ
8:         print "proof(sources m)"
9:         J ← apply Chain to m ∈ knows(tr)
10:        for each J ∈ J do
11:            print "case nameOf(J) thus ?thesis"
12:            PrfGen(J)
13:        end for
14:        print "qed"
15:    end if
16: end procedure
```

Figure 6. The basic proof generation algorithm PRFGEN.

different message constructors, *false* is added to the premises Γ. Second, the premises Γ are saturated by extending them with all atoms derivable using inference rules other than the CHAIN rule. Third, the resulting judgment $\Gamma \vdash_P \Delta$ is checked for *trivial validity*; i.e., 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 $\tau(\Delta) \subseteq \Gamma$.

If this is the case, then Isabelle will be able to prove the validity of J using its built-in tactic "auto". Otherwise, a simple heuristic is used to select an atom $(m \in \mathit{knows}(tr)) \in \Gamma$ that has not yet been selected. If no such atom exists, proof-generation fails. Otherwise the CHAIN rule is applied to the selected atom, which 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 PRFGEN.

Although protocol security is undecidable [21], [26], the PRFGEN algorithm terminates for many practical protocols (e.g., all the case studies from Table I). Furthermore, similar to the algorithms underlying Athena [6] and Scyther [8], the failure to generate a proof may indicate an attack. The basic idea is that each case arising from an application of the CHAIN rule to an atom $m \in \mathit{knows}(tr)$ corresponds to an explanation of how the intruder could have learned m. Moreover, for most security properties, the only premises that are non-trivially satisfiable are assertions about the intruder knowledge. Hence, if the algorithm terminates but fails to generate a proof, then there is an explanation for all messages the intruder is required to know. Therefore, these explanations indicate how to construct an attack.

2) Weak-Atomicity: To allow the PRFGEN algorithm to prove that a protocol is weakly atomic, we also allow judgments of the form

$$\Gamma \vdash_P \sigma(v, \mathit{tid}) \in \mathit{knows}(tr) \vee \sigma(v, \mathit{tid}) \in \mathit{Fresh}\sharp\mathit{TID}\ ,$$

where Γ is a set of atoms as before. We also redefine trivial validity accordingly. Hence, we can represent the conclusion of an application of Theorem 1 as a set of judgments and prove their validity using PRFGEN.

3) Efficiency Improvements: The time required by Isabelle for checking a security proof based on decryption-chain reasoning is roughly proportional to the size of the protocol and the number of CHAIN rule applications. In order to reduce this time, we extended the PRFGEN algorithm to use a branch-and-bound strategy to search for the proof with the least number of CHAIN rule applications. The benefit of searching for small proofs is particularly pronounced for more complex protocols such as Paulson's model of TLS, where we measured an improvement in checking time by a factor 10. However, this comes at the cost of a factor 20 increase in generation time.

To reduce the generation time, we exploit the observation made in the discussion of Example 4: proofs of security properties may reuse already proven properties. In the proofs generated by PRFGEN, opportunities for reuse manifest themselves as subproofs that are equal up to the renaming of thread identifiers. We did not tackle the difficult problem of automatically extracting reusable security properties from a generated proof. Instead, we rely on a heuristic and, as a last resort, the user to suggest candidate reusable security properties. The heuristic is simple: given the protocol description, it generates secrecy properties for all long-term keys and all nonces sent and received encrypted. Additionally, for all nonces sent as plaintext, it generates a first origination property stating that these nonces must have been sent before the intruder learns them.

Technically, we extend the PRFGEN algorithm with the ability to reuse security properties by defining resolution for judgments. Assume we have proven a lemma $\Gamma_1 \vdash_P \Delta$ with no existentially quantified thread identifiers. When proving a judgment of the form $\Gamma_1 \cup \Gamma_2 \vdash_P \Pi$, we can reuse 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. Note that resolution is especially useful if $\Delta = \mathit{false}$ (e.g., secrecy properties have this form) because then no additional subproof apart from the existing proof of the lemma $\Gamma_1 \vdash_P \mathit{false}$ is needed.

	Protocol	generation	checking
1	Amended NS	2.42s	90s
2	Yahalom (Paulson's variant)	0.17s	28s
3	Andrew Secure RPC	0.03s	12s
4	Bilateral-Key Exchange	0.03s	14s
5	Denning-Sacco	0.39s	56s
6	NSL	0.03s	12s
7	TLS (simplified)	0.14s	15s
8	Wide Mouthed Frog	0.06s	15s
9	TLS (Paulson's Model)	0.33s	76s
10	Kerberos V	33.76s	146s
11	Kerberos V' (manual props.)	2.93s	135s

Table I
TIMES FOR AUTOMATIC PROOF GENERATION AND CHECKING.

In our experiments, we found that reusing security properties and generating smallest proofs significantly improves both proof generation as well as proof checking time. For example, for Paulson's TLS model, the generation time improved by a factor of 3.3 and the checking time improved by a factor of 47 compared to using the basic algorithm. Moreover, the smallest proofs tend to be really short; on average 2.3 applications of the CHAIN rule are required to prove a property of a protocol in our case studies. Hence, these proofs are well-suited for human inspection and understanding.

Experimental Results: Table I shows the experimental results we obtained using our implementation of the PRFGEN algorithm and its extensions. The source code used to generate these results can be downloaded from [23]. 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 protocols 1-8, we prove non-injective agreement [15] of all shared data for the initiator and responder where possible and secrecy for the session-key and payload, if they are present. For protocols 9-11, we additionally prove non-injective synchronization [16] (a strengthened variant of non-injective agreement).

Note that protocols 1-8 and the secrecy properties of protocol 10 were previously verified using other approaches for the automatic generation of machine-checked security proofs and we make comparisons in Section V. In contrast, protocol 9 and the authentication properties of protocols 10-11 have not been verified previously using other automatic approaches. They show how our method works for more complex protocols and properties.

The comparatively high generation time for Kerberos V is due to several unnecessary security properties being suggested by the heuristic for reuse. By manually selecting only the necessary properties (represented as Protocol 11 in the table), we reduced the generation time by an order of magnitude, while still proving the same properties of interest.

V. RELATED WORK

We discuss related work from four areas: interactive methods for machine-checked proofs, automatic proof methods, related proof methods where proofs are not machine-checked, and work related to our proof-generation algorithm.

Interactive Methods for Machine-Checked Proofs: The Inductive Approach is one of the most successful approaches for interactively constructing machine-checked security proofs. It was initially developed by Paulson [1] and later extended by Bella [25] and Blanqui [27]. It defines protocols indirectly as an inductively-defined set of traces denoting their execution 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 applying 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 [1] and the analysis of TLS took six weeks [3]. 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 interactive proof construction were developed using the PVS theorem prover. The first approach was developed by Evans and Schneider [28]. It is based on a formalization of rank-functions [29]. 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 [30]. It is based a formalization of a variant of strand spaces [31] and authentication tests [32]. They do not give any proof construction times.

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 [13], [33]. 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, 2, 10, and 11 from Table I were also analyzed by Goubault-Larrecq. The times reported in [33] are in the same range as ours. Similar to our method, proof checking takes about a minute. The approach can be used directly with equational theories, but currently

cannot handle the strong authentication properties considered in our work. It also requires trusting the soundness of the (non-trivial) abstractions required to model security protocols using Horn-clauses. In contrast, our method uses a single stateful execution model, from which we formally derive all verification rules.

A recent approach by Brucker and Mödersheim for automatic generation of machine-checkable proofs is described in [12]. They use the OFMC model checker [34] 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 reference model of P formalized in the style of the Inductive Approach.

Protocols 3-8 from Table I were also investigated by Brucker and Mödersheim. Comparing timings, we see that the times for proof generation are similar. However, our times for proof checking are orders of magnitude faster (ranging from 10 times for NSL and to about 1700 times for the simplified TLS variant). Currently, their approach assumes a typed model and a non-standard intruder who can only send messages matching patterns of the protocol.

Related Proof Methods: The Protocol Composition Logic (PCL) proposed by Datta et. al. [35] is a methodology based on giving a set of axioms and inference rules, which allow for the manual construction of security proofs from simple invariants. These manually-specified invariants can be checked automatically using Prolog. However, no method has been proposed yet for machine-checking full PCL proofs nor the soundness of the axioms and inference rules.

Related Algorithms: Our proof-generation algorithm can be viewed as a significant extension of the Scyther algorithm [8], which in turn is a descendant of the Athena algorithm [6]. Our algorithm extends the Scyther algorithm by allowing a larger range of security properties, automatic generation of weak-atomicity proofs, and enabling the algorithm to reuse already proven properties.

VI. Conclusions

We formally derived a verification theory for security protocols from an operational protocol semantics. Based on this theory, we developed an algorithm for the automatic generation of machine-checked security proofs. Using our tool-supported methodology, machine-checked security proofs can be built either interactively or automatically. In both cases, the resulting construction time is faster than competing approaches.

An interesting application area for our work is the certification of security protocols, which is currently being pursued in some countries, such as Japan [36] and France [13]. Our tool-supported methodology allows non-experts to provide machine-checked protocol security proofs. Moreover, as the underlying foundations were formally derived from a straightforward operational semantics, we have strong correctness guarantees for the resulting verification.

As future work, we plan to extend our approach with a richer protocol execution model, e.g., with non-linear role scripts and support for equational theories. We would also like to integrate additional reasoning techniques, such as abstraction-based overapproximations as used by ProVerif [7] and OFMC [34], authentication tests [37], and generalized typing invariants (for verifying protocols that are not weakly atomic). Having a machine-checked formalization simplifies deriving additional inference rules to improve the performance of the proof-generation algorithm. Conversely, our implementation enables us to efficiently construct and examine proof scripts for large sets of protocols.

References

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

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

[3] L. C. Paulson, "Inductive analysis of the internet protocol TLS," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 3, pp. 332–351, 1999.

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

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

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

[7] B. Blanchet, "Automatic verification of correspondences for security protocols," *Journal of Computer Security*, vol. 17, no. 4, pp. 363–434, 2009.

[8] 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*. New York, NY, USA: ACM, 2008, pp. 119–128.

[9] S. Berezin, "Model checking and theorem proving: a unified framework," Ph.D. dissertation, Carnegie Mellon University, Jan. 2002.

[10] ——, "Extensions to Athena: Constraint satisfiability problem and new pruning theorems based on type system extensions for messages," 2001, unpublished manuscript. [Online]. Available: http://www.sergeyberezin.com/papers/athena-extensions.ps

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

[12] A. Brucker and S. Mödersheim, "Integrating automated and interactive protocol verification," in *Workshop on Formal Aspects in Security and Trust (FAST 2009)*, P. Degano and J. Guttman, Eds., 2009.

[13] J. Goubault-Larrecq, "Towards producing formally checkable security proofs, automatically," in *CSF*. IEEE Computer Society, 2008, pp. 224–238.

[14] F. Baader and T. Nipkow, *Term rewriting and all that*. New York, NY, USA: Cambridge University Press, 1998.

[15] G. Lowe, "A hierarchy of authentication specifications," in *CSFW*. IEEE Computer Society, 1997, pp. 31–44.

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

[17] M. J. C. Gordon, "Mechanizing programming logics in higher order logic," pp. 387–439, 1989.

[18] J. Heather, G. Lowe, and S. Schneider, "How to prevent type flaw attacks on security protocols," in *CSFW*, 2000, pp. 255–268.

[19] 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.*, vol. 21, no. 1, pp. 59–84, 2005.

[20] M. Arapinis and M. Duflot, "Bounding messages for free in security protocols," in *FSTTCS*, ser. Lecture Notes in Computer Science, V. Arvind and S. Prasad, Eds., vol. 4855. Springer, 2007, pp. 376–387.

[21] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov, "Undecidability of bounded security protocols," in *Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP, Trento, Italy*, N. Heintze and E. Clarke, Eds., 1999.

[22] M. Wenzel, L. C. Paulson, and T. Nipkow, "The Isabelle framework," in *TPHOLs*, ser. Lecture Notes in Computer Science, O. A. Mohamed, C. Muñoz, and S. Tahar, Eds., vol. 5170. Springer, 2008, pp. 33–38.

[23] "Source code of the Isabelle/HOL formalization and the automatic proof generation tool presented in this paper," April 2010. [Online]. Available: https://www.infsec.ethz.ch/research/Software

[24] L. C. Paulson, "Relations between secrets: Two formal analyses of the Yahalom protocol," *Journal of Computer Security*, vol. 9, no. 3, pp. 197–216, 2001.

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

[26] H. Comon and V. Cortier, "Tree automata with one memory set constraints and cryptographic protocols," *Theor. Comput. Sci.*, vol. 331, no. 1, pp. 143–214, 2005.

[27] F. Blanqui, "An Isabelle formalization of protocol-independent secrecy with an application to e-commerce," *CoRR*, vol. abs/cs/0610069, 2006.

[28] N. Evans and S. A. Schneider, "Verifying security protocols with PVS: widening the rank function approach," *J. Log. Algebr. Program.*, vol. 64, no. 2, pp. 253–284, 2005.

[29] S. Schneider, "Verifying authentication protocols with CSP," in *CSFW*, 1997, pp. 3–17.

[30] B. Jacobs and I. Hasuo, "Semantics and logic for security protocols," *J. Comput. Secur.*, no. 6, pp. 909–944, 2009.

[31] F. J. Thayer, J. C. Herzog, and J. D. Guttman, "Strand spaces: Proving security protocols correct," *Journal of Computer Security*, vol. 7, no. 1, 1999.

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

[33] J. Goubault-Larrecq, "Finite models for formal security proofs," *Journal of Computer Security*, 2010, to appear.

[34] S. Mödersheim and L. Viganò, "The Open-source Fixed-point Model Checker for symbolic analysis of security protocols," in *FOSAD*, ser. Lecture Notes in Computer Science, A. Aldini, G. Barthe, and R. Gorrieri, Eds., vol. 5705. Springer, 2009, pp. 166–194.

[35] "Protocol Composition Logic (PCL)," http://crypto.stanford.edu/protocols/.

[36] S. Matsuo, K. Miyazaki, A. Otsuka, and D. Basin, "How to evaluate the security of real-life cryptographic protocols? the cases of ISO/IEC 29128 and CRYPTREC," in *Financial Cryptography*, 2010, to appear.

[37] S. F. Doghmi, J. D. Guttman, and F. J. Thayer, "Searching for shapes in cryptographic protocols," in *TACAS*, ser. Lecture Notes in Computer Science, O. Grumberg and M. Huth, Eds., vol. 4424. Springer, 2007, pp. 523–537.

APPENDIX

A. Example of an Authentication Proof

The following proof demonstrates how to prove an authentication property using decryption-chain reasoning. Furthermore, it also shows an example of reusing a previously proven secrecy property. As we explain in the discussion of Example 4, reusing security properties plays an important role in the speedup (compared to existing approaches) we achieved both for automatic as well as interactive proof construction of machine-checked security proofs.

Example 7 (Proof of non-injective synchronization)**.** We prove that $\forall q \in \mathit{reachable}(\mathit{CR}).\ \phi_{\text{auth}}(q)$, where ϕ_{auth} is defined in Example 3.

Proof: We must show that for every state $(tr, th, \sigma) \in \mathit{reachable}(\mathit{CR})$ 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.

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

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 rule INPUT and $\mathsf{h}(k\sharp i) \in \mathit{knows}(tr)$ using rule KNOWN. Applying the CHAIN rule and removing trivial cases yields

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

Case **(1)** is where the intruder builds the received message by himself. Using rule KNOWN, we have that $k\sharp i \in \mathit{knows}(tr)$, which contradicts the secrecy property we proved in Example 4.

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

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

Case **(2.1)** states that the intruder fakes the message, which again contradicts the secrecy property proven in Example 4 due to $k\sharp i \prec_{tr} \{\!|k\sharp i|\!\}_{\mathsf{pk}_{\sigma(s,j)}}$ and rule KNOWN.

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

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

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

As for the secrecy proof in Example 4, our mechanization of decryption-chain reasoning allows for a succinct representation of the above proof highlighting its core.

Example 8. The statement and the proof of the non-injective synchronization property ϕ_{auth} from Example 3 are formalized in Isabelle/HOL by the following proof script.

```
1:  lemma (in CR-state) client-nisynch:
2:    assumes
3:      "(tid, C2) ∈ steps(tr)"
4:      "roleth(tid) = C"
5:      "σ(AV(''s''), tid) ∉ Compr"
6:    shows
7:      "∃ j. roleth(j) = S ∧
8:            σ(AV(''s''), tid) = σ(AV(''s''), j)  ∧
9:            LN(''k'', tid)    = σ(MV(''v''), j) ∧
10:           St(tid, C1) ≺ St(j, S1)             ∧
11:           St(j, S2)   ≺ St(tid, C2)"
12:   proof(sources "inst_σ,tid(C2-pt)")
13:     case fake thus ?thesis
14:       by (auto dest!: client-k-secrecy[OF known])
15:   next
16:     case (S2-hash j) thus ?thesis
17:     proof(sources "inst_σ,j(S1-pt)")
18:       case fake thus ?thesis
19:         by (auto dest!: client-k-secrecy[OF known])
20:     next
21:       case (C1-enc i) thus ?thesis by auto
22:     qed
23:   qed
```

Lines 2–11 are a direct translation of the security predicate 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 CHAIN rule to the message received in the second step of the client role C. We denote this message by the instantiation 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 7. We discharge this case by calling Isabelle's built-in tactic "auto" instructing it 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 we were looking for. In Line 17, as in the pen-and-paper proof, we apply the CHAIN rule 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 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.