Performing Security Proofs of Stateful Protocols

Andreas V. Hess* Sebastian Mödersheim*
Achim D. Brucker† Anders Schlichtkrull‡

*DTU Compute, Technical University of Denmark,
Lyngby, Denmark
{avhe,samo}@dtu.dk
†Department of Computer Science, University of Exeter,
Exeter, United Kingdom
a.brucker@exeter.ac.uk
‡Department of Computer Science, Aalborg University,
Copenhagen, Denmark
andsch@cs.aau.dk

Abstract

In protocol verification we observe a wide spectrum from fully automated methods to interactive theorem proving with proof assistants like Isabelle/HOL. The latter provide overwhelmingly high assurance of the correctness, which automated methods often cannot: due to their complexity, bugs in such automated verification tools are likely and thus the risk of erroneously verifying a flawed protocol is non-negligible. There are a few works that try to combine advantages from both ends of the spectrum: a high degree of automation and assurance. We present here a first step towards achieving this for a more challenging class of protocols, namely those that work with a mutable long-term state. To our knowledge this is the first approach that achieves fully automated verification of stateful protocols in an LCF-style theorem prover. The approach also includes a simple user-friendly transaction-based protocol specification language embedded into Isabelle, and can also leverage a number of existing results such as soundness of a typed model.

1 Introduction

There are at least three reasons why it is desirable to perform proofs of security in a proof assistant like Isabelle/HOL or Coq. First, it gives us an overwhelming assurance that the proof of security is actually a proof and not just the result of a bug in a complex verification tool. This is because the basic idea of an LCF-style theorem prover is to have an abstract datatype *theorem* so that

new theorems can only be constructed through functions that correspond to accepted proof rules; thus implementing just this datatype correctly prevents us from ever accepting a wrong proof as a theorem, no matter what complex machinery we build for automatically finding proofs. Second, a human may have an insight of how to easily prove a particular statement where a "stupid" verification algorithm may run into a complex check or even be infeasible. Third, the language of a proof assistant can formalize all accepted mathematics, so there is no narrow limit on what aspects of a system we can formalize, e.g., physical properties.

Paulson [32] and Bella [6] developed a protocol model in Isabelle and performed several security proofs in this model, e.g., [33]. That the proof of a single protocol (for which even some automated security proofs exist) is worth a publication, underlines how demanding it is to conduct proofs in a proof assistant. This raised the question of how one can automatically produce proofs that can be checked by a proof assistant and thus get the mentioned overwhelming assurance. The first works in this direction consider tools based on Horn-clause resolution like ProVerif [19, 11], as well as the tool Scyther-proof [26] for the backward-search based tool Scyther [17].

A drawback of these approaches so far is that they only apply to Alice-and-Bob style protocols where there is no relation between several sessions. When we consider, however, any system that maintains a mutable long-term state, e.g., a security token or a server that maintains a simple database, we hit the limits of tools like ProVerif and Scyther. To cope with the complexity, some extensions to ProVerif have been proposed [3, 14], but also a tool that went a completely different way: Tamarin [28] is actually inspired by Scyther-proof and has the flavor of a proof assistant environment itself, namely combining partial automation with interactively performing a proof, i.e., supplying the right lemmas to show. Interestingly, there is no connection to Isabelle or other LCF-style theorem provers, while one may intuitively expect that this should be easily possible. The reason seems to be that Tamarin combines several specialized automated methods, especially for term algebraic reasoning, that would be quite difficult to "translate" into Isabelle/HOL—at least the authors of this paper do not see an easy way to make such a connection. In fact, if it was possible for a large class of stateful protocols, the combination of overwhelming assurance of proofs and a high degree of automation would be extremely desirable.

The goal of this work is to achieve exactly this combination for a well-defined fragment of stateful protocols. We are here using as a foundation the Isabelle/HOL formalization and protocol model by Hess et al. [22]. One reason for this choice is that the proof technique we present in this paper works only in a restricted typed model. Fortunately, that formalization ships with a typing result [25], namely an Isabelle theorem that says: if a protocol is secure in this typed model, then it is also secure in the full model without the typing restriction—as long as the protocol in question satisfies a number of basic requirements. Thus we get fully automated Isabelle proofs for most protocols even without a typing restriction.

The automated proof technique we employ in this paper is based on the

set-based abstraction approach of [12, 30]. The basic idea is that we represent the long-term state of a protocol by a number of sets; the protocol rules specify how protocol participants shall insert elements into a set, remove them from a set, and check for membership or non-membership. (The intruder may also be given access to some sets.) Based on this, we perform an abstract interpretation approach that identifies those elements that have the same membership status in all sets and compute a *fixed point*, more precisely a representation of all messages that the intruder can ever know after any trace of the protocol (including the set membership status of elements that occur in these messages). One may wonder if considering just intruder-known messages limits the approach to secrecy goals, but thanks to sets, a wide range of trace-based properties can be expressed by reduction to the secrecy of a special constant `attack`. (We cannot, however, handle privacy-type properties in this way.)

We thus check if the fixed point contains the `attack` constant, and if so, we can abort the attempt to prove the protocol correct. This may happen also for a secure protocol as the abstraction entails an over-approximation. However, vice-versa, if `attack` is not in the fixed point, then the protocol should be secure—if the fixed point is indeed a sound representation of the messages the intruder can ever know. The proof we perform in Isabelle now is thus basically to show that the fixed point is closed under every protocol rule: given any trace where the intruder knows only messages covered by the fixed point, then every extension by one protocol step reveals only messages also covered by the fixed point.

The main contribution of this paper is the Isabelle implementation of a mechanism to both compute the abstract fixed point—the proof idea so to speak—and to then break it down into digestible pieces for Isabelle. This proof consists of two main parts: first, we have a number of protocol-independent theorems that we have proved in Isabelle once and for all, and second, for every protocol and fixed point, we have a number of checks that Isabelle can directly execute to establish the correctness of a given protocol. The entire protocol-independent formalization consists of more than 15,000 lines of Isabelle code (definitions, theorems and proofs). This figure includes also another minor contribution: we devise a simple protocol specification language into Isabelle that overcomes some drawbacks of low-level input languages like (multi-)set rewriting rules used in some other tools. The complete formalization is available at the Archive of Formal Proofs as the entry titled *Automated Stateful Protocol Verification* [23]:

`https://www.isa-afp.org/entries/Automated_Stateful_Protocol_Verification.html`

The latest development version and related works can be found at the following webpage:

`https://people.compute.dtu.dk/samo/composec.html`

This gives us thus a fully automated approach to generate proofs of highest assurance for a relevant class of stateful security protocols. This class is limited by the automated method behind it, most notably we can only insert atomic messages into the sets, but allows for full automation.

The rest of this paper is organized as follows: Section 2 introduces preliminaries, Section 3 defines the protocol model, Section 4 explains the set-based abstraction approach, Section 5 introduces the protocol checks with optimizations introduced in Section 6, Section 7 presents and reports on the results of a number of experiments applying our approach to a selection of protocols, and finally Section 8 is the conclusion where we also discuss related work.

2 Preliminaries

2.1 Terms and Substitutions

We model terms over a countable set Σ of *symbols* (also called *function symbols* or *operators*) and a countable set $\mathcal{V}$ of *variables* disjoint from Σ. Each symbol in Σ has an associated arity and we denote by Σ^n the symbols of Σ of arity n. A *term* built from $S \subseteq \Sigma$ and $X \subseteq \mathcal{V}$ is then either a variable $x \in X$ or a *composed term* of the form $f(t_1, \ldots, t_n)$ where each t_i is a term built from S and X, and $f \in S^n$. The set of terms built from S and X is denoted by $\mathcal{T}(S, X)$. Arbitrary terms t usually range over $\mathcal{T}(\Sigma, \mathcal{V})$, unless stated otherwise. By $subterms(t)$ we denote the set of subterms of t.

The set of *constants* $\mathcal{C}$ is defined as the symbols with arity zero: $\mathcal{C} \equiv \Sigma^0$. It contains the following distinct subsets:

- the countable set $\mathbb{V}$ of *concrete values* (or just *values*),
- the finite set $\mathbb{A}$ of *abstract values*,
- the finite set $\mathbb{E}$ of *enumeration constants*,
- the finite set $\mathbb{S}$ of *database constants*,[1]
- and a special constant attack.

The analyst, i.e., the author of a protocol specification may freely choose $\mathbb{E}$ and $\mathbb{S}$ as well as any number of function symbols $\mathbb{F}$ with their arities (disjoint from the above subsets).

Example 1 *Consider a protocol with two users* a *and* b*, and where each user* a *has its own keyring* $\mathsf{ring}(a)$*, and the server maintains databases of the currently valid keys* $\mathsf{valid}(a)$ *and revoked keys* $\mathsf{revoked}(a)$ *for* a*. For such a protocol we define* $\mathbb{E} = \{\mathsf{a}, \mathsf{b}\}$ *and* $\mathbb{S} = \{\mathsf{ring}(a), \mathsf{valid}(a), \mathsf{revoked}(a) \mid a \in \mathbb{E}\}$. □

We regard all elements of $\mathbb{S}$ as constants, despite the function notation, which is just to ease specification. This work is currently limited to finite enumerations and finite sets, as handling infinite domains would require substantial complications of the approach (e.g., a symbolic representation or a small system result).

[1] These databases are simply sets of messages and we therefore often refer to them simply as "sets" in this paper.

Arbitrary constants are usually denoted by a, b, c, d, whereas arbitrary variables are denoted by x, y, and z. By $\bar{x}$ we denote a finite list $x_1, \ldots, x_n$ of variables.

We furthermore partition Σ into the *public* symbols (those symbols that are available to the intruder) and the *private* symbols (those that are not). We denote by Σ_{pub} and Σ_{priv} the set of public respectively private symbols. By $\mathcal{C}_{pub}$ and $\mathcal{C}_{priv}$ we then denote the sets of public respectively private constants. The constant attack, the values $\mathbb{V}$, the abstract values $\mathbb{A}$, and the database constants $\mathbb{S}$ are all private.

The set of variables of a term t is denoted by $fv(t)$ and we say that t is *ground* iff $fv(t) = \emptyset$. Both definitions are extended to sets of terms as expected.

A *substitution* is a mapping from variables $\mathcal{V}$ to terms. The *substitution domain* (or just *domain*) $dom(\theta)$ of a substitution θ is defined as the set of those variables that are not mapped to themselves by θ: $dom(\theta) \equiv \{x \in \mathcal{V} \mid \theta(x) \neq x\}$. The *substitution range* (or just *range*) $ran(\theta)$ of θ is the image of the domain of θ under θ: $ran(\theta) \equiv \theta(dom(\theta))$. For finite substitutions we use the notation $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ to denote the substitution with domain $\{x_1, \ldots, x_n\}$ and range $\{t_1, \ldots, t_n\}$ that sends each x_i to t_i. Substitutions are extended to composed terms homomorphically as expected. A substitution δ is *injective* iff $\delta(x) = \delta(y)$ implies $x = y$ for all $x, y \in dom(\delta)$. An *interpretation* is a substitution $\mathcal{I}$ such that $dom(\mathcal{I}) = \mathcal{V}$ and $ran(\mathcal{I})$ is ground. A *variable renaming* ρ is an injective substitution such that $ran(\rho) \subseteq \mathcal{V}$. An *abstraction substitution* is a substitution δ such that $ran(\delta) \subseteq \mathbb{A}$.

2.2 The Intruder Model

We employ the intruder model from [22] which is in the style of Dolev and Yao: the intruder controls the communication medium and can encrypt and decrypt with known keys, but the intruder cannot break cryptography. More formally, we define that the intruder can derive a message t from a set of known messages M (the *intruder knowledge*, or just *knowledge*), written $M \vdash t$, as the least relation closed under the following rules:

$$\frac{}{M \vdash t}\ \begin{array}{c}(Axiom)\\ t \in M\end{array} \qquad \frac{M \vdash t_1 \quad \cdots \quad M \vdash t_n}{M \vdash f(t_1, \ldots, t_n)}\ \begin{array}{c}(Compose)\\ f \in \Sigma^n_{pub}\end{array}$$

$$\frac{M \vdash t \quad M \vdash k_1 \quad \cdots \quad M \vdash k_n}{M \vdash r}\ \begin{array}{c}(Decompose)\\ \mathsf{Ana}(t) = (K, R), r \in R,\\ K = \{k_1, \ldots, k_n\}\end{array}$$

where $\mathsf{Ana}(t) = (K, R)$ is a function that maps a term t to a pair of sets of terms K and R. We also define a restricted variant $\vdash_c$ of $\vdash$ as the least relation closed under the $(Axiom)$ and $(Compose)$ rules only.

The $(Axiom)$ rule simply expresses that all messages directly known to the intruder are derivable, the $(Compose)$ rule closes the derivable terms under the application of public function symbols such as encryption or public constants

(when $f \in \Sigma^0_{pub} = \mathcal{C}_{pub}$). The (*Decompose*) rule represents decomposition operations: $\mathsf{Ana}(t) = (K, R)$ means that t is a term that can be analyzed, provided that the intruder knows all the "keys" in the set K, and he will then obtain the "results" in R. This gives us a general way to deal with typical constructor/destructor theories without needing to work with algebraic equations and rewriting. We may also write $\mathsf{Keys}(t)$ and $\mathsf{Result}(t)$ to denote the set of keys respectively results from analyzing t, i.e., $\mathsf{Ana}(t) = (\mathsf{Keys}(t), \mathsf{Result}(t))$.

Example 2 *To model asymmetric encryption and signatures we first fix two public* $\mathsf{crypt}, \mathsf{sign} \in \mathbb{F}^2$ *and one private* $\mathsf{inv} \in \mathbb{F}^1$ *function symbols. The term* $\mathsf{crypt}(k, m)$ *then denotes the message m encrypted with a public key k and* $\mathsf{sign}(\mathsf{inv}(k), m)$ *denotes m signed with the private key* $\mathsf{inv}(k)$ *of k. To obtain a message m encrypted with a public key k the intruder must produce the private key* $\mathsf{inv}(k)$ *of k. Formally, we define the analysis rule* $\mathsf{Ana}_{\mathsf{crypt}}(x_1, x_2) = (\{\mathsf{inv}(x_1)\}, \{x_2\})$. *For signatures we define the rule* $\mathsf{Ana}_{\mathsf{sign}}(x_1, x_2) = (\emptyset, \{x_2\})$ *modeling that the intruder can open any signature that he knows. We also model a transparent pairing function by fixing* $\mathsf{pair} \in \Sigma^2$ *and defining the rule* $\mathsf{Ana}_{\mathsf{pair}}(x_1, x_2) = (\emptyset, \{x_1, x_2\})$. □

Note that we have in this example used a simple notation for describing $\mathsf{Ana}(t)$ for an arbitrary term t: each rule $\mathsf{Ana}_f(x_1, \ldots, x_n) = (K, R)$ defines Ana for a constructor $f \in \mathbb{F}^n$. Here x_i are distinct variable symbols, and K and R are sets of terms such that $R \subseteq \{x_1, \ldots, x_n\}$ and $K \subseteq \mathcal{T}(\mathbb{F}, \{x_1, \ldots, x_n\})$. Note that for each constructor we have at most one analysis rule, and for all constructors without an analysis rule we just have $\mathsf{Ana}(t) = (\emptyset, \emptyset)$. (An example for the latter is a hash function: the intruder cannot obtain information from a hash value.)

The reason for this convention is that the formalization of [22] requires that the Ana function satisfies certain conditions, most notably that it is invariant under substitutions.[2] Without going into detail, our notation of the Ana rules allows for an automated proof that all these requirements are satisfied. Thus, this allows the user to specify an arbitrary constructor/destructor theory with these Ana rules without having to prove anything manually.

2.3 Typed Model

Our result is based on a typed model in which the intruder is restricted to only making "well-typed" choices. Many protocol verification methods [6, 7, 11, 32, 33] rely on such a typed model since it simplifies the protocol verification problem. There exist many typing results [16, 25, 1, 24, 21, 2] that show that a restriction to a typed model is sound for large classes of protocols. That is, it is without loss of attacks to restrict the verification to a typed model. Each

[2] One may wonder why we do not allow for analysis rules of the form $\mathsf{Ana}_f(t_1, \ldots, t_n) = (K, R)$, where the t_i are arbitrary terms instead of just variables. Because of the substitution invariance requirement from [22] on Ana such analysis rules would not lead to more expressive Ana functions.

such result show that if a protocol satisfies certain syntactic conditions and is secure in a typed model then the protocol is secure also in an untyped model. [25] is such a result that is part of the Isabelle formalization we employ. Since this result has itself been proved in Isabelle, it is sufficient to obtain the Isabelle proof of a protocol in the unrestricted model from the Isabelle proof in the typed model and that the protocol satisfies the requirements of the typing result. As a minor contribution of this paper that we just mention here is that we have automated the Isabelle proof of these requirements of the typing result for the protocol specification language we present. Thus, all that is left to do in the following section is the automated proof for the protocol in the typed model.

In a nutshell, the typing result requires that messages with different intended meaning cannot be confused for each other—a condition called *type-flaw resistance*. More formally, the typed model is parameterized over a *typing function* Γ and a finite set of *atomic types* $\mathfrak{T}_a$ satisfying the following:

- $\Gamma(x) \in \mathcal{T}(\Sigma \setminus \mathcal{C}, \mathfrak{T}_a)$ for $x \in \mathcal{V}$ (where $\mathfrak{T}_a$ here acts like a set of "variables")
- $\Gamma(c) \in \mathfrak{T}_a$ for $c \in \mathcal{C}$
- $\Gamma(f(t_1, \ldots, t_n)) = f(\Gamma(t_1), \ldots, \Gamma(t_n))$ for $f \in \Sigma \setminus \mathcal{C}$

A substitution θ is then said to be *well-typed* iff $\Gamma(\theta(x)) = \Gamma(x)$ for all variables x. In this paper we use $\mathfrak{T}_a = \{\mathsf{value}, \mathsf{enum}, \mathsf{settype}, \mathsf{attacktype}\}$, and the elements of $\mathbb{A} \cup \mathbb{V}$ have type value, the elements of $\mathbb{E}$ have type enum, the elements of $\mathbb{S}$ have type settype and attack has type attacktype. We furthermore assume that all variables that we use in protocol specifications have atomic types, and we denote by $\mathcal{V}_a$ the set of variables with atomic type a (e.g., $\mathcal{V}_{\mathsf{value}}$ is the set of value-typed variables). As an example, let $x, y \in \mathcal{V}_{\mathsf{value}}$ and $a \in \mathbb{E}$, then $\Gamma(\mathsf{sign}(\mathsf{inv}(x), \mathsf{pair}(a, y))) = \mathsf{sign}(\mathsf{inv}(\mathsf{value}), \mathsf{pair}(\mathsf{enum}, \mathsf{value}))$. Suppose an agent expects to receive a term of this type; then the typed model means the restriction that the intruder can only send messages of this type, i.e., he cannot send in place of x and y some terms of a different type. This restriction of the intruder to typed terms—which is without loss of generality when the requirements of the typing result hold—is drastically simplifying the task of proving the protocol correct.

3 Transactions

The Isabelle protocol model of [22] consists of a number of *transactions* specifying the behavior of the participants. A transaction consists of any combination of the following: input messages to receive, checks on the sets, modifications of the sets, and output messages to send. A transaction can only be executed atomically, i.e., it can only fire when input messages are present, such that the checks are satisfied, and then they produce all changes and the output messages in one state transition. Instead of defining a ground state transition system, [22] considers building symbolic traces as sequences of transactions with their

variables renamed apart, and with any instantiation of the variables that satisfies the checks and the intruder model in the sense that the intruder can produce every input message from previous output messages. (Transactions can also describe additional abilities of the intruder such as reading a set.) Security goals are formulated by transactions that check for a situation we consider as a successful attack, and then reveal the special constant attack to the intruder. Thus a protocol is safe if no symbolic constraint with the intruder finally sending attack has a satisfying interpretation. Note that the length of symbolic traces is finite but unbounded (i.e., an unbounded session model), and that the number of enumeration constants and databases currently supported is arbitrary but fixed in the specification.

For the convenience of an automated verification tool, we have defined a small language based on transactions with a bit of syntactic sugar, and this language is directly embedded into Isabelle. We introduce this language only at hand of a keyserver example adapted from [22] that we also use as a running example for the remainder of this paper.

3.1 A Keyserver Protocol

Before we proceed with the formal definitions we illustrate our protocol model through the keyserver example. Here users can register public keys at a trusted keyserver and these keys can later be revoked. Each user U has an associated keyring $\mathsf{ring}(U)$ with which it keeps track of its keys. (The elements of $\mathsf{ring}(U)$ are actually public keys; we implicitly assume that the user U knows the corresponding private key.)

First, we model a mechanism outOfBand by which a user U can register a new key PK at the keyserver out-of-band, e.g., by physically visiting the keyserver. The user U first constructs a fresh public key PK and inserts PK into its keyring $\mathsf{ring}(U)$. We model that the keyserver—in the same transaction—learns the key and adds it to its database of valid keys for user U, i.e., into a set $\mathsf{valid}(U)$. Finally, PK is published:

$$\begin{array}{l}
\mathsf{outOfBand}(U\colon \mathsf{user})\\
\ \mathsf{new}\ PK\\
\ \mathsf{insert}\ PK\ \mathsf{ring}(U)\\
\ \mathsf{insert}\ PK\ \mathsf{valid}(U)\\
\ \mathsf{send}\ PK.
\end{array}$$

Note that there is no built-in notion of set ownership, or who exactly is performing an action: we just specify with such transactions what can happen. The intuition is that $\mathsf{ring}(U)$ is a set of public keys controlled by U (and U has the corresponding private key of each) while $\mathsf{valid}(U)$ is controlled by the server (who is not even given a name here). Putting it into a single transaction models that this is something happening in collaboration between a user and a server.

Next, we model a key update mechanism that allows for registering a new key while simultaneously revoking an old one. Here we model this as two transactions, one for the user and one for the server, since here we model a scenario

where user and server communicate via an asynchronous network controlled by the intruder. To initiate the key revocation process the user U first picks and removes a key PK from its keyring to later revoke, then freshly generates a new key NPK and stores it in its keyring. (Again the corresponding private key $\mathsf{inv}(NPK)$ is known to U, but this is not explicitly described.) As a final step the user signs the new key with the private key $\mathsf{inv}(PK)$ of the old key and sends this signature to the server by transmitting it over the network:

$$
\begin{array}{l}
\mathsf{keyUpdateUser}(U\colon \mathsf{user}, PK\colon \mathsf{value}) \\
\;PK \text{ } \mathsf{in} \text{ } \mathsf{ring}(U) \\
\;\mathsf{new} \text{ } NPK \\
\;\mathsf{delete} \text{ } PK \text{ } \mathsf{ring}(U) \\
\;\mathsf{insert} \text{ } NPK \text{ } \mathsf{ring}(U) \\
\;\mathsf{send} \text{ } \mathsf{sign}(\mathsf{inv}(PK), \mathsf{pair}(U, NPK)).
\end{array}
$$

The check PK in $\mathsf{ring}(U)$ represents here a non-deterministic choice of an element of $\mathsf{ring}(U)$. (Note that a user can register any number of keys with the outOfBand transaction.) Note also that we declare PK as a variable of type value, because PK is not freshly generated; all freshly generated elements, like NPK here, are automatically of type value.

When the server receives the signed message, it checks that PK is indeed a valid key, that NPK has not been registered earlier, and then revokes PK and registers NPK. To keep track of revoked keys, the server maintains another database $\mathsf{revoked}(U)$ containing the revoked keys of U:

$$
\begin{array}{l}
\mathsf{keyUpdateServer}(U\colon \mathsf{user}, PK\colon \mathsf{value}, NPK\colon \mathsf{value}) \\
\;\mathsf{receive} \text{ } \mathsf{sign}(\mathsf{inv}(PK), \mathsf{pair}(U, NPK)) \\
\;PK \text{ } \mathsf{in} \text{ } \mathsf{valid}(U) \\
\;NPK \text{ } \mathsf{notin} \text{ } \mathsf{valid}(_) \\
\;NPK \text{ } \mathsf{notin} \text{ } \mathsf{revoked}(_) \\
\;\mathsf{delete} \text{ } PK \text{ } \mathsf{valid}(U) \\
\;\mathsf{insert} \text{ } PK \text{ } \mathsf{revoked}(U) \\
\;\mathsf{insert} \text{ } NPK \text{ } \mathsf{valid}(U) \\
\;\mathsf{send} \text{ } \mathsf{inv}(PK).
\end{array}
$$

As a last action, the old private key $\mathsf{inv}(PK)$ is revealed. This is of course not what one would do in a reasonable implementation, but it allows us to prove that the protocol is correct even if the intruder obtains all private keys to revoked public keys. (This could also be separated into a rule that just leaks private keys of revoked keys.)

Actions of the form x notin $s(_)$ for $s \in \Sigma^n$ are syntactic sugar for the sequence of actions x notin $s(a)$ for each $a \in \mathbb{E}$.

Finally, we define that there is an attack if the intruder learns a valid key of an honest user. This, again, can be modeled as a sequence of actions in which we check if the conditions for an attack holds, and, if so, transmit the constant attack that acts as a signal for goal violations. Let honest be a subset of user

that contains only the honest agents. Then we define:

$$
\begin{array}{l}
\mathsf{attackDef}(U\colon \mathsf{honest}, PK\colon \mathsf{value}) \\
\ \mathsf{receive}\ \mathsf{inv}(PK) \\
\ PK\ \mathsf{in}\ \mathsf{valid}(U) \\
\ \mathsf{attack}.
\end{array}
$$

The last action attack is just syntactic sugar for send attack.

3.2 Protocol Model

The keyserver protocol that we just defined consists of *transactions* that we now formally define. To keep the formal definitions simple we omit the variable declarations and the syntactic sugar employed in our protocol specification language. Thus only value-typed variables remain in transactions since the enumeration variables are resolved as syntactic sugar. A transaction T is then of the form $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$ where the S_i are *strands* built from the following grammar:

$$
\begin{array}{lcl}
S_r & ::= & \mathsf{receive}\ t \cdot S_r \mid 0 \\
S_c & ::= & x\ \mathsf{in}\ s \cdot S_c \mid x\ \mathsf{notin}\ s \cdot S_c \mid x \not\doteq x' \cdot S_c \mid 0 \\
F & ::= & \mathsf{new}\ x \cdot F \mid 0 \\
S_u & ::= & \mathsf{insert}\ x\ s \cdot S_u \mid \mathsf{delete}\ x\ s \cdot S_u \mid 0 \\
S_s & ::= & \mathsf{send}\ u \cdot S_s \mid 0
\end{array}
$$

where $x, x' \in \mathcal{V}_{\mathsf{value}}$, $s \in \mathbb{S}$, $t \in \mathcal{T}(\mathbb{E} \cup \mathbb{F}, \mathcal{V}_{\mathsf{value}})$, $u \in \mathcal{T}(\mathbb{E} \cup \mathbb{F}, \mathcal{V}_{\mathsf{value}}) \cup \{\mathsf{attack}\}$, and where 0 denotes the empty strand.

The function *fv* is extended to transactions as expected, and for a transaction $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$ we define $\mathit{fresh}(T) \equiv \mathit{fv}(F)$ (i.e., $x \in \mathit{fresh}(T)$ iff new x occurs in T).

Protocols are defined as finite sets of such transactions $\mathcal{P} = \{T_1, \ldots, T_n\}$. Their semantics is defined in terms of a symbolic transition system in which *constraints* are built up during transitions. A constraint essentially records what transactions have been taken, but from the intruder's point of view in the sense that the directions of transmitted messages are swapped (so receives become sends and vice-versa). For this reason the syntax of constraints $\mathcal{A}$ is similar to the syntax for transactions, but without the new construct:[3]

$$
\begin{array}{rl}
\mathcal{A} ::= & \mathsf{send}\ t \cdot \mathcal{A} \mid \mathsf{receive}\ t \cdot \mathcal{A} \mid t \not\doteq t' \cdot \mathcal{A} \mid \mathsf{insert}\ t\ t' \cdot \mathcal{A} \mid \\
& \mathsf{delete}\ t\ t' \cdot \mathcal{A} \mid t\ \mathsf{in}\ t' \cdot \mathcal{A} \mid t\ \mathsf{notin}\ t' \cdot \mathcal{A} \mid 0
\end{array}
$$

where $t, t' \in \mathcal{T}(\Sigma, \mathcal{V})$ and where 0 is the empty constraint.

For the semantics of constraints we define a relation $\mathcal{I} \models_D^M \mathcal{A}$ where $\mathcal{A}$ is a constraint, M is the intruder knowledge (the messages sent so far), D is a set

[3] When building up constraints during transitions the variables x occurring in new x actions will be instantiated with fresh values. Hence, new x actions will never occur in constraints. For the same reason the constraint syntax needs to be slightly more flexible compared to the transaction syntax, so as to allow for actions such as insert t s where $t \notin \mathcal{V}$.

of pairs representing the current state of the databases (e.g., $(k, s) \in D$ iff k is an element of the database s), and $\mathcal{I}$ is an interpretation:

$$
\begin{array}{lll}
\mathcal{I} \models_D^M 0 & \text{iff} & \mathit{true} \\
\mathcal{I} \models_D^M \mathsf{send}\ t \cdot \mathcal{A} & \text{iff} & M \vdash \mathcal{I}(t) \text{ and } \mathcal{I} \models_D^M \mathcal{A} \\
\mathcal{I} \models_D^M \mathsf{receive}\ t \cdot \mathcal{A} & \text{iff} & \mathcal{I} \models_D^{M \cup \{\mathcal{I}(t)\}} \mathcal{A} \\
\mathcal{I} \models_D^M \mathsf{insert}\ t\ s \cdot \mathcal{A} & \text{iff} & \mathcal{I} \models_{D \cup \{\mathcal{I}((t,s))\}}^M \mathcal{A} \\
\mathcal{I} \models_D^M \mathsf{delete}\ t\ s \cdot \mathcal{A} & \text{iff} & \mathcal{I} \models_{D \setminus \{\mathcal{I}((t,s))\}}^M \mathcal{A} \\
\mathcal{I} \models_D^M t \not\doteq t' \cdot \mathcal{A} & \text{iff} & \mathcal{I}(t) \neq \mathcal{I}(t') \text{ and } \mathcal{I} \models_D^M \mathcal{A} \\
\mathcal{I} \models_D^M t\ \mathsf{in}\ s \cdot \mathcal{A} & \text{iff} & \mathcal{I}((t,s)) \in D \text{ and } \mathcal{I} \models_D^M \mathcal{A} \\
\mathcal{I} \models_D^M t\ \mathsf{notin}\ s \cdot \mathcal{A} & \text{iff} & \mathcal{I}((t,s)) \notin D \text{ and } \mathcal{I} \models_D^M \mathcal{A}
\end{array}
$$

We say that $\mathcal{I}$ is a *model of* $\mathcal{A}$, written $\mathcal{I} \models \mathcal{A}$, iff $\mathcal{I} \models_\emptyset^\emptyset \mathcal{A}$. We may also apply substitutions θ to constraints $\mathcal{A}$, written $\theta(\mathcal{A})$, by extending the definition of substitution application appropriately. The function *fv* is also extended to constraints.

We can then define a transition relation $\Rightarrow_\mathcal{P}^\bullet$ for protocols $\mathcal{P}$ in which states are constraints and the initial state is the empty constraint 0. First, we define the *dual* of a constraint $\mathcal{A}$, written $\mathit{dual}(\mathcal{A})$, as "swapping" the direction of the sent and received messages of $\mathcal{A}$: $\mathit{dual}(0) = 0$, $\mathit{dual}(\mathsf{receive}\ t \cdot \mathcal{A}) = \mathsf{send}\ t \cdot \mathit{dual}(\mathcal{A})$, $\mathit{dual}(\mathsf{send}\ t \cdot \mathcal{A}) = \mathsf{receive}\ t \cdot \mathit{dual}(\mathcal{A})$, and $\mathit{dual}(\mathfrak{a} \cdot \mathcal{A}) = \mathfrak{a} \cdot \mathit{dual}(\mathcal{A})$ otherwise. The transition

$$\mathcal{A} \Rightarrow_\mathcal{P}^\bullet \mathcal{A} \cdot \mathit{dual}(\rho(\sigma(S_r \cdot S_c \cdot S_u \cdot S_s)))$$

is then applicable for a transaction $T \in \mathcal{P}$ if the following conditions are met:

1. $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$ for some F,

2. σ is a substitution mapping $\mathit{fresh}(T)$ to fresh values (i.e., $\mathit{dom}(\sigma) = \mathit{fresh}(T)$, $\mathit{ran}(\sigma) \subseteq \mathbb{V}$, and the elements of $\mathit{ran}(\sigma)$ do not occur in $\mathcal{A}$), and

3. ρ is a variable renaming sending the variables of T to new variables that do not occur in $\mathcal{A}$ or $\mathcal{P}$ (that is, $\mathit{dom}(\rho) = \mathit{fv}(T)$ and $(\mathit{fv}(\mathcal{A}) \cup \mathit{fv}(\mathcal{P})) \cap \mathit{ran}(\rho) = \emptyset$).

A constraint $\mathcal{A}$ is said to be *reachable in* $\mathcal{P}$ iff $0 \Rightarrow_\mathcal{P}^{\bullet\star} \mathcal{A}$ where $\Rightarrow_\mathcal{P}^{\bullet\star}$ denotes the transitive reflexive closure of $\Rightarrow_\mathcal{P}^\bullet$. The protocol then has an *attack* iff there exists a reachable and satisfiable constraint where the intruder can produce the attack signal, i.e., there exists a reachable $\mathcal{A}$ in $\mathcal{P}$ and an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{A} \cdot \mathsf{send\ attack}$. If $\mathcal{P}$ does not have an attack then $\mathcal{P}$ is *secure*.

3.3 Well-Formedness

We are going to employ the abstraction-based verification technique from [30] in the following to automatically generate security proofs. The technique has a few more requirements in order to work and which we bundle in a notion of *well-formedness*.

First, when a transaction uses a variable when sending a message or performing a set update, then that variable must either be fresh or have occurred positively in a received message or check. Intuitively, transactions cannot produce a value "out of the blue", but the value either has to exist before the transaction (in some message or set) or is created by the transaction. Formally, let $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$, then we require:

C1: $\mathit{fv}(S_u) \cup \mathit{fv}(S_s) \subseteq \mathit{fv}(S_r) \cup \mathit{fv}(S_c) \cup \mathit{fresh}(T)$

C2: $\mathit{fresh}(T) \cap (\mathit{fv}(S_r) \cup \mathit{fv}(S_c)) = \emptyset$

(The second condition simply states that values that are freshly generated by a transaction T should not also occur in the received messages and the checks of T.)

The abstraction approach that we employ, furthermore, would not work if, e.g., an agent freshly creates a value and stores it in a set, but never sends it out as part of a message. This is because the abstraction discards the explicit representation of sets, and just keeps the abstracted messages. An easy workaround is to define a special private unary function symbol occurs and augment every rule containing action new x with the action send occurs(x); and we augment every transaction where variable x occurs but is not freshly generated with receive occurs(x). In order not to bother the user with this, our tool can make this transformation automatically.

This addition of occurs has, however, a subtle consequence. Suppose a specification contains no transaction that generates any fresh value, but, say, only an attack rule like this:

```
attackDef2(PK: value)
 receive PK
 attack.
```

This rule cannot fire after the occurs transformation, because it adds the requirement to receive occurs(PK) which nobody can produce. One would, however, naturally expect that said protocol is not secure. Thus, we require (and automatically check) that each protocol specification includes a value-producing transaction, i.e., a transaction that is applicable in every state and generates a fresh value.

One may wonder in the above example why the intruder is not able to provide the value, since he has an unlimited supply of constants of every type, including type value. However, for such a constant c he does not have occurs(c) (because it is not fresh and occurs is private) and thus cannot use it in any transaction. Thus, we have a model where the intruder by default cannot generate fresh values, and it is thus the design choice of the user to define a transaction that allows the intruder to generate fresh values, if this is desired. This is in our opinion more flexible than a fixed intruder rule, since the rule can be adapted to the context of a particular model. For instance, in the keyserver example where values represent public keys one may define the intruder rule that gives

also the corresponding private key to the intruder and inserts it into a dedicated set:

$$\begin{array}{l} \mathsf{intruderValues}() \\ \ \mathsf{new}\ PK \\ \ \mathsf{insert}\ PK\ \mathsf{intruderkeys} \\ \ \mathsf{send}\ PK \\ \ \mathsf{send}\ \mathsf{inv}(PK). \end{array}$$

Finally, a small technical difficulty arises when a transaction has two variables x, y that could be the same value, i.e., that allows for a model $\mathcal{I}$ with $\mathcal{I}(x) = \mathcal{I}(y)$. This is difficult to handle in the verification since the transaction may require to insert x into a set and delete y from that very set. To steer clear of this, the paper [30] simply defines the semantics to be injective on variables. For user-friendliness we do not want to follow this, and rather do the following: for any rule with variables x and y that are not part of a new construct, we generate a variant of the rule where we unify x and y, checking whether this gives a consistent transaction. If so, we add it to the rule system. Then we add the constraint $x \neq y$ to the original rule. We do that until all rules have $x \neq y$ for all pairs of variables that are not freshly generated. For instance, in the keyserver example, we have only one rule to look at: keyUpdateServer with variables PK and NPK. Since unifying PK and NPK gives an unsatisfiable rule, it is safe to add $PK \neq NPK$ to it.

4 Set-Based Abstraction

We now come to the core of our approach: for a given protocol, how to automatically verify and generate a security proof that Isabelle can accept. As explained earlier this is based on an abstract interpretation method called set-based abstraction [29, 12, 30]. Of course this approach is rather complicated and lengthy when explained in all detail, and even more so is the contribution of the present paper, i.e., the Isabelle machinery and proofs building on this idea. Thus, the following explanations can only summarize the approach, leaving out many details (especially ones published in existing works), and can give the reader only an idea. We therefore emphasize again that the final Isabelle-accepted proofs do not depend on the correctness of this machinery we sketch in the following: the machinery is just to provide a proof that Isabelle can check, and an error in this machinery (or an attack in the protocol) only leads to failure to prove it in Isabelle.

Recall that in the previous section we formalized a protocol model by reachable constraints $\mathcal{A}$ (i.e., a sequence of transactions where variables have been named apart and the send/receive direction has been swapped in order to express it from the intruder's point of view) with their satisfying interpretations $\mathcal{I} \models \mathcal{A}$. Note that $\models$ is defined via a relation $\models_D^M$, representing the intruder knowledge—all the messages received so far—and the state of the sets $\mathbb{S}$—all values inserted into a set that were not deleted so far. We could thus character-

ize the "state" of the entire system after a number of instantiated transactions by these two items, M and D.

Example 3 *In our keyserver example the following instantiated constraint is possible (after taking a transition with* outOfBand *followed by one with* keyUpdateUser*):*

$$
\begin{array}{l}
\mathsf{insert}\ \mathsf{pk}_1\ \mathsf{ring}(\mathsf{a})\\
\mathsf{insert}\ \mathsf{pk}_1\ \mathsf{valid}(\mathsf{a})\\
\mathsf{receive}\ \mathsf{pk}_1\\
\mathsf{pk}_1\ \mathsf{in}\ \ \mathsf{ring}(\mathsf{a})\\
\mathsf{delete}\ \mathsf{pk}_1\ \mathsf{ring}(\mathsf{a})\\
\mathsf{insert}\ \mathsf{pk}_2\ \mathsf{ring}(\mathsf{a})\\
\mathsf{receive}\ \mathsf{sign}(\mathsf{inv}(\mathsf{pk}_1), \mathsf{pair}(\mathsf{a}, \mathsf{pk}_2))
\end{array}
$$

After this trace we have

$$
\begin{array}{rcl}
M & = & \{\mathsf{pk}_1, \mathsf{sign}(\mathsf{inv}(\mathsf{pk}_1), \mathsf{pair}(\mathsf{a}, \mathsf{pk}_2))\},\ \textit{and}\\
D & = & \{(\mathsf{pk}_1, \mathsf{valid}(\mathsf{a})), (\mathsf{pk}_2, \mathsf{ring}(\mathsf{a}))\}.
\end{array}
$$

The idea is that the abstract representation replaces each concrete value of $\mathbb{V}$ with a constant from $\mathbb{A}$ that represents the set memberships. For this reason, we choose $\mathbb{A}$ to be isomorphic to the power set of $\mathbb{S}$, i.e., for every subset of $\mathbb{S}$, we have a corresponding constant in $\mathbb{A}$. We write $\boxed{A}$ for the abstract value that corresponds to the subset A of $\mathbb{S}$, e.g., if $s_1, s_2 \in \mathbb{S}$ then $\boxed{\{s_1, s_2\}} \in \mathbb{A}$.

Thus, given a state D of the databases we define an abstraction function from $\mathbb{V}$ to $\mathbb{A}$ as follows:

$$\alpha_D(c) = \boxed{\{s \mid (c, s) \in D\}}$$

and we extend it to terms and sets of terms as expected.

Example 4 *In the previous example we have* $\alpha_D(\mathsf{pk}_1) = \boxed{\{\mathsf{valid}(\mathsf{a})\}}$ *and* $\alpha_D(\mathsf{pk}_2) = \boxed{\{\mathsf{ring}(\mathsf{a})\}}$. *Thus* $\alpha_D(M) = \{\boxed{\{\mathsf{valid}(\mathsf{a})\}}, \mathsf{sign}(\mathsf{inv}(\boxed{\{\mathsf{valid}(\mathsf{a})\}}), \mathsf{pair}(\mathsf{a}, \boxed{\{\mathsf{ring}(\mathsf{a})\}}))\}$.

The key idea is to compute the *fixed point* of all the abstract messages that the intruder can obtain in any model of any reachable constraint. Note that this fixed point is in general infinite, even if $\mathbb{S}$ is finite (and thus so is $\mathbb{A}$), because the intruder can compose arbitrarily complex messages and send them. This is why tools like [29, 12, 30] do not directly compute it but represent it by a set of Horn clauses and check using resolution whether attack is derivable.

However, remember that we can restrict ourselves to the typed model and use the typing result of [25] to infer the security proof without the typing restriction. All variables that occur in a constraint are of type value (the parameter variables of the transactions are de-sugared) and thus, in a typed model it holds that $\mathcal{I}(x) \in \mathbb{V}$ for every variable x and well-typed interpretation $\mathcal{I}$. While $\mathbb{V}$ is still countably infinite, the abstraction (in any state D) maps to the finite $\mathbb{A}$. Thus the fixed point is always finite in a typed model.

There is a subtle point here: even though we limit the variables to well-typed terms, and thus all messages that can ever be sent or received, the Dolev-Yao

closure is still infinite, i.e., for a (finite) set M of messages there are still infinitely many t such that $M \vdash t$. Only finitely many of these t can be sent by the intruder in the typed model, but one may wonder if the entire derivation relation $\vdash$ can be limited to "well-typed" terms without losing attacks. Indeed, we define *well-typed terms* as the set of terms that includes all well-typed instances of sent and received messages in transactions, and that is closed under subterms and Keys. We have now proved in Isabelle that for the intruder to derive any well-typed term, it is sound to also limit the intruder deduction to well-typed terms, so no ill-typed intermediate terms are needed during the derivation. (This is indeed very similar to some lemmas we have proved for parallel compositionality, namely for so-called homogeneous terms the deduction does not need to consider any inhomogeneous terms [22].) Thus, it is sufficient to limit the fixed point, including intruder deduction, to well-typed terms, and thus have a finite fixed point.

4.1 Term Implication

Let us now see in more detail how to compute the fixed point. An important aspect of the abstraction approach is that the global state is mutable, i.e., the set membership of concrete values can change over transitions, and so their abstraction changes. For this we have the notion of a term implication:

Definition 1 *A* term implication (a, b) *is a pair of abstract values* $a, b \in \mathbb{A}$ *and a* term implication graph TI *is a binary relation between abstract values, i.e.,* $TI \subseteq \mathbb{A} \times \mathbb{A}$*. Instead of* $(a, b) \in TI$ *we may also write* $a \twoheadrightarrow b$*.*

The reason we use the word "implication" is as follows. An abstract value $a \in \mathbb{A}$ in general represents a multitude of concrete values $c_1, c_2, \ldots \in \mathbb{V}$; when a concrete value c_1 changes its set memberships, then its abstraction changes, say to $b \in \mathbb{A}$; however, the other concrete values $c_2, \ldots \in \mathbb{V}$ that had the same abstraction before this change do not necessarily change their set memberships. Thus for an abstract message like $f(a)$ that contains the abstract value a, this a could either represent the concrete value c_1 that changes its set-membership, i.e., produce $f(b)$, or it could represent one of the other concrete values $c_2, \ldots$ and thus $f(a)$ would be the abstract message after the change. The abstraction must thus include both: every term implication $a \twoheadrightarrow b$ means that for every fact $f(a)$ we additionally have $f(b)$. Note that in case of several occurrences, e.g., $f(a, a)$, we have additionally $f(a, b)$, $f(b, a)$, and $f(b, b)$, since each occurrence of a could represent a different concrete value. This is captured by the following definitions:

Definition 2 (Term transformation) *Let* (a, b) *be a term implication. The* term transformation under (a, b) *is the least relation* ${}_a\!\twoheadrightarrow_b$ *closed under the following rules:*

$$\frac{}{x \;{}_a\!\twoheadrightarrow_b\; x}\; x \in \mathcal{V} \qquad \frac{}{a \;{}_a\!\twoheadrightarrow_b\; b}$$

$$\frac{t_1 \;{}_a\!\twoheadrightarrow_b\; s_1 \quad \cdots \quad t_n \;{}_a\!\twoheadrightarrow_b\; s_n}{f(t_1, \ldots, t_n) \;{}_a\!\twoheadrightarrow_b\; f(s_1, \ldots, s_n)}\; f \in \Sigma^n$$

Note that this relation is also reflexive since $a \;{}_a{\twoheadrightarrow}_b\; a$ *follows from* $a \in \Sigma^0$*. If* $t \;{}_a{\twoheadrightarrow}_b\; t'$ *then we say that* t' is implied by t under (a, b)*, or just* t' is implied by t *for short.*

Definition 3 (Term implication closure) *Let* TI *be a term implication graph and let* t *be a term. The* term implication closure of t under TI *is defined as the least set* $cl_{TI}(t)$ *closed under the following rules:*

$$\frac{}{t \in cl_{TI}(t)} \qquad \frac{t' \in cl_{TI}(t)}{t'' \in cl_{TI}(t)} \; \begin{array}{l}(a \twoheadrightarrow b) \in TI,\\ t' \;{}_a{\twoheadrightarrow}_b\; t''\end{array}$$

This definition is extended to sets of terms M *as expected. If* $t' \in cl_{TI}(t)$ *then we say that* t' is implied by t (under TI).

The idea is that the fixed point should ultimately be closed under the term implication graph. However, this closure is actually quite large in many practical examples, and thus we just record the messages that are ever received by the intruder together with the term implication graph, but without performing this closure explicitly:

Definition 4 *A* protocol fixed-point candidate*, or* fixed point *for short,*[4] *is a pair* (FP, TI) *such that*

1. FP *is a finite and ground set of terms over* $\mathcal{T}(\Sigma \setminus \mathbb{V}, \emptyset)$*.*
2. TI *is a term implication graph:* $TI \subseteq \mathbb{A} \times \mathbb{A}$*.*

4.2 Limitations

There are some limitations of our approach that we now mention. First, we inherit the free algebra term model from [22] (two terms are equal iff they are syntactically equal) and so we do not support algebraic properties such as needed for Diffie-Hellman. Secondly, we inherit the limitations of AIF's set-based abstraction approach:

- We require each protocol to have a fixed and finite number of enumeration constants and sets. This typically means that also the number of agents is fixed—at least if the protocol has to specify a number of sets for each agent.
- We require that the sets can only contain values. The reason is to allow these values to be abstracted by set membership.
- We cannot refer directly to particular constants of type value. This would not be very useful as every value with the same set-membership status are identified with the same abstract value under the set-based abstraction.

[4] Here "candidate" is to emphasize that this is just a proof idea that has yet to be verified by Isabelle.

Our approach allows for an unbounded number of sessions. The only difference here between our work and e.g. Tamarin [28] and ProVerif [9] is that we need, as mentioned, to fix the number of enumeration constants and sets, and thereby, in a typical specification, also fix the number of agents. However, there is no difference in the notion of unbounded sessions: We allow for an unbounded number of transitions, every set can contain an unbounded number of values, and the intruder can make an unbounded number of steps.

Because we use the typing result from [25], we also require that protocols have to satisfy the type-flaw resistance requirements of that result. These requirements are a generalization of the common tagging mechanisms which should in many applications not be a practical limitation. Note that this requirement is checked automatically.

Finally, we do not directly support private channels, but one can instead send messages under a private function. For instance, one can write in a transaction $\mathsf{send}\ \mathsf{privChan}(A, B, t)$ where A and B are of type enum and t is a message. Such communication is asynchronous. One can model synchronous communication only in a limited way here through sets, e.g., as $\mathsf{insert}\ \mathit{Nonce}\ \mathsf{privCh}(A, B)$.

4.3 Example of a Fixed-Point Computation

Consider again the keyserver protocol defined in Subsection 3.1; for simplicity we do this example for just one user a who is also honest: $\mathsf{user} = \mathsf{honest} = \{\mathsf{a}\}$. We show how the fixed point (or rather the candidate that we then check with Isabelle) is computed; to make it more readable, let us give the fixed point right away and then see how each element is derived: $\mathsf{FP}_{ks} \equiv (FP_{ks}, TI_{ks})$ where

$$\begin{aligned} FP_{ks} \equiv \{\ & \{\mathsf{ring}(\mathsf{a}), \mathsf{valid}(\mathsf{a})\}, \{\mathsf{ring}(\mathsf{a})\}, \mathsf{inv}(\{\mathsf{revoked}(\mathsf{a})\}), \\ & \mathsf{sign}(\mathsf{inv}(\{\mathsf{valid}(\mathsf{a})\}), \mathsf{pair}(\mathsf{a}, \{\mathsf{ring}(\mathsf{a})\})) \\ & \mathsf{sign}(\mathsf{inv}(\emptyset), \mathsf{pair}(\mathsf{a}, \{\mathsf{ring}(\mathsf{a})\}))\ \} \end{aligned}$$

and where the term implication graph TI_{ks} can be represented graphically as follows where each edge $a \twoheadrightarrow b$ corresponds to an element of TI_{ks}:

$$\begin{array}{ccccccc} \{\mathsf{ring}(\mathsf{a})\} & \longrightarrow\!\!\!\!\!\twoheadrightarrow & \emptyset & \longrightarrow\!\!\!\!\!\twoheadrightarrow & \{\mathsf{valid}(\mathsf{a})\} & \longrightarrow\!\!\!\!\!\twoheadrightarrow & \{\mathsf{revoked}(\mathsf{a})\} \\ & \searrow & & \nearrow & & & \\ & & \{\mathsf{ring}(\mathsf{a}), \mathsf{valid}(\mathsf{a})\} & & & & \end{array}$$

Note that we can actually reduce the representation of the fixed point a little bit as we do not need to include facts that can be obtained via term implication from others; with this optimization we obtain actually:

$$\begin{aligned} FP'_{ks} \quad \equiv \quad \{\ & \mathsf{sign}(\mathsf{inv}(\emptyset), \mathsf{pair}(\mathsf{a}, \{\mathsf{ring}(\mathsf{a})\})), \\ & \{\mathsf{ring}(\mathsf{a})\}, \mathsf{inv}(\{\mathsf{revoked}(\mathsf{a})\})\ \ \} \end{aligned}$$

To compute this, we first consider the transaction $\mathsf{outOfBand}$ where a fresh key is inserted into both $\mathsf{ring}(\mathsf{a})$ and $\mathsf{valid}(\mathsf{a})$ and sent out. The abstraction of this key is thus the value $\{\mathsf{ring}(\mathsf{a}), \mathsf{valid}(\mathsf{a})\}$. This value is in the intruder knowledge

in FP_{ks} but redundant due to other messages we derive later).[5] Note that this rule cannot produce anything else so we do not consider it for the remainder.

Next let us look at the transaction keyUpdateUser. For keyUpdateUser we need to choose an abstract value for PK that satisfies the check PK in ring(a). At this point in the fixed-point computation we have only $\{\mathsf{ring(a)}, \mathsf{valid(a)}\}$. Since the transaction removes the key PK from ring(a), we get the term implication $\{\mathsf{ring(a)}, \mathsf{valid(a)}\} \twoheadrightarrow \{\mathsf{valid(a)}\}$. A fresh value NPK is also generated and inserted into ring(a), and a signed message is sent out which gives us: $\mathsf{sign}(\mathsf{inv}(\{\mathsf{valid(a)}\}), \mathsf{pair}(\mathsf{a}, \{\mathsf{ring(a)}\}))$. Also, this one is a message that later becomes redundant with further messages. By analysis, the intruder also obtains $\{\mathsf{ring(a)}\}$.

The new value $\{\mathsf{ring(a)}\}$ allows for another application of the keyUpdateUser rule, namely with this key in the role of PK. This now gives the term implication $\{\mathsf{ring(a)}\} \twoheadrightarrow \emptyset$ and the message $\mathsf{sign}(\mathsf{inv}(\emptyset), \mathsf{pair}(\mathsf{a}, \{\mathsf{ring(a)}\}))$. After this, there are no further ways to apply this transaction rule, because we will not get to any other abstract value that contains ring(a).

Applying the keyUpdateServer transaction to the first signature we have obtained (i.e., with $PK = \{\mathsf{valid(a)}\}$ and $NPK = \{\mathsf{ring(a)}\}$), we get the term implications $\{\mathsf{valid(a)}\} \twoheadrightarrow \{\mathsf{revoked(a)}\}$ and $\{\mathsf{ring(a)}\} \twoheadrightarrow \{\mathsf{ring(a)}, \mathsf{valid(a)}\}$, and the intruder learns $\mathsf{inv}(\{\mathsf{revoked(a)}\})$. Applying it with the second signature (i.e., with $PK = \emptyset$ and NPK as before), we get additionally the term implication $\emptyset \twoheadrightarrow \{\mathsf{valid(a)}\}$. Note that we must also check if the intruder can generate a signature that works with keyUpdateServer: however, the only private keys he knows are those represented by $\mathsf{inv}(\{\mathsf{revoked(a)}\})$ and they are not accepted for this transaction. (In a model with dishonest agents, the intruder can of course produce signatures with keys registered to a dishonest agent name, but here we have just one honest user a.)

No other transaction can produce anything we do not have in FP_{ks} already—in particular we cannot apply the attack transaction and this concludes the fixed-point computation. Thus—according to our abstract interpretation analysis—the protocol is indeed secure. Next we try to convince Isabelle.

5 Checking Fixed-Point Coverage

A major contribution of this work is now to use the fixed point that was automatically computed by the abstract interpretation approach as a "proof idea" for conducting the security proof in Isabelle on the concrete protocol. Essentially, we prove that the fixed point indeed "covers" everything that can happen. We break this down into an induction proof: given any trace that is covered by the fixed point, if we extended it by any applicable transition, then the resulting trace is also covered by the fixed point. This induction step we break down into a number of *checks* that are directly executable within Isabelle us-

[5] In fact, the well-formedness conditions of the previous section require to also include occurs facts, but for illustration, we have simply omitted them (as the intruder knows every public key that occurs).

ing the built-in term rewriting proof method *code-simp*. We have also proved some protocol-independent Isabelle theorems that show that any protocol that passes said checks is indeed correct. Note that these checks are not only fully automated, but they are also terminating in all but a few degenerate cases.[6]

Term rewriting within Isabelle benefits from the exact same correctness guarantees as the rest of Isabelle since every rewriting step is verified by the system. This, of course, also means that evaluation through the built-in term rewriting proof methods can be several orders of magnitudes slower than executing a program since every application of a rewrite rule is checked by Isabelle's core inference system. Fortunately, Isabelle provides a means of evaluating terms through code generation using, e.g., the built-in proof method *eval*. This allows us to offer a trade-off to our users: one can use either *code-simp* for the overwhelming assurance of Isabelle proofs, or one can use *eval* to get the full speed of running code outside of Isabelle but with the disadvantage of having to trust in the correctness of the code generation infrastructure. For instance, one could use *eval* during development of a protocol specification and only use *code-simp* as a final step to get a fully Isabelle-verified proof of security.

5.1 Automatically Checking for Fixed-Point Coverage

Let us look at how we can automatically check if a fixed-point covers a protocol. We first explain how this works in general and thereafter give an example, in Example 5, of how it works using the keyserver example.

A transaction of the protocol after resolving all the sugar has only variables of type value. Thus, in a typed model and under the abstraction, we can instantiate the variables only with abstract values, i.e., elements from $\mathbb{A}$. We first define what it means that a transaction is applicable under such a substitution of the variables with respect to the fixed point computed by the abstract interpretation:

Definition 5 (Fixed-point coverage: pre-conditions) *Let* $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$ *be a transaction and let* $\mathsf{FP} = (FP, TI)$ *be a fixed point. Let further* δ *be an abstraction substitution mapping the variables of* T *to abstract values of* $\mathbb{A}$*. We say that* δ satisfies the pre-conditions (for T and FP)*, written* $\mathsf{pre}(\mathsf{FP}, \delta, T)$*, iff the following conditions are met:*

F1. $cl_{TI}(FP) \vdash \delta(t)$ *for all* $\mathsf{receive}\ t$ *occurring in* S_r

F2. $s \in \delta(x)$ *for all* x in s *occurring in* S_c

F3. $s \notin \delta(x)$ *for all* x notin s *occurring in* S_c

[6] It is technically possible to specify protocols for which the checks do not terminate. For instance, an analysis rule of the form $\mathsf{Ana}_f(x) = (\{f(f(x))\}, R)$, for some f, x and R, would lead to termination issues when automatically proving the conditions for the typing result which we rely on, because we here need to compute a set that contains the terms occurring in the protocol specification and is closed under keys needed for analysis, and such a set would in this case be infinite. However, this is an artificial example that normally does not occur since it is usually the case that keys cannot themselves be analyzed.

F4. $\delta(x) = \emptyset$ *for all* $x \in \mathit{fresh}(T)$

(We write here $s \in \delta(x)$ as a short-hand for $s \in A$ for the set A such that $\delta(x) = A$. $s \notin \delta(x)$ is defined similarly.) Here, F1 checks that the intruder can produce all input messages for the transaction under the given δ. Note that the intruder has control over the entire network, so he can use here any message honest agents have sent and also construct other messages from that knowledge (hence the $\vdash$). Moreover, we consider here the closure of the intruder knowledge FP under the term implication rules, since that represents all variants of the messages that are available to the intruder; we will later show as an optimization that we can check whether $cl_{TI}(FP) \vdash \delta(t)$ holds without first explicitly computing $cl_{TI}(FP)$. The next checks F2 and F3 are that all set membership conditions are satisfied, and F4 checks that all fresh variables represent values that are not member of any set.

Now for every δ under which the transaction T can be applied (according to FP), we compute what T can "produce" and that that is also covered by FP. What the transaction can produce are the outgoing messages and the changes in set memberships. The latter is captured by an updated abstraction substitution δ_u that is identical with δ except for those values that changed their set memberships during the transaction:

Definition 6 (Abstraction substitution update) *Let* $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$ *be a transaction and* δ *an abstraction substitution. We define the* update of δ w.r.t. T*, written* δ_u*, as follows:*

$$\delta_u(x) \equiv \mathsf{upd}(S_u, x, \delta(x)), \ \textit{where}$$

$$\begin{aligned} \mathsf{upd}(0, x, a) &= a \\ \mathsf{upd}(\mathsf{t}.S, x, a) &= \begin{cases} \mathsf{upd}(S, x, a \cup \{s\}) & \textit{if } \mathsf{t} = \mathsf{insert}(x, s) \\ \mathsf{upd}(S, x, a \setminus \{s\}) & \textit{if } \mathsf{t} = \mathsf{delete}(x, s) \\ \mathsf{upd}(S, x, a) & \textit{otherwise} \end{cases} \end{aligned}$$

Note that according to this definition, if a transaction contains insert and delete operations of the same value x for the same set, then "the last one counts". But there is a more subtle point: suppose the transaction includes the operations $\mathsf{insert}(x, s)$ and $\mathsf{delete}(y, s)$. The above definition would not necessarily formalize the updates of the set memberships if the transaction were applicable (in the concrete) under an interpretation $\mathcal{I}$ with $\mathcal{I}(x) = \mathcal{I}(y)$. Note that for this very reason the concrete semantics requires $\mathcal{I}$ to be injective, and, as explained earlier in Subsection 3.3, we automatically achieve this through appropriate syntactic sugar so as to not bother the user.

Based on this update, we can now define what it means for a transaction to be covered by a fixed point:

Definition 7 (Fixed-point coverage: post-conditions) *Let* $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$ *be a transaction and let* $\mathsf{FP} = (FP, TI)$ *be a fixed point. Let* δ *be an abstraction substitution and* δ_u *the update of* δ *w.r.t.* T*. We say that*

δ satisfies the post-conditions (for T and FP), *written* $\mathsf{post}(\mathsf{FP}, \delta, T)$, *iff the following conditions are met:*

G1. $(\delta(x) \twoheadrightarrow \delta_u(x)) \in TI^*$ *for all* $x \in fv(T) \setminus fresh(T)$

G2. $cl_{TI}(FP) \vdash \delta_u(t)$ *for all* send t *occurring in* S_s

Here G1 expresses that every update of a value must be a path in the term implication graph (it does not need to be a single edge). G2 means that the intruder learns every outgoing message $\delta_u(t)$ and thus it must be covered by the fixed point when closed under term implication.

We can now put it all together: for the pre-conditions we are restricting the coverage check to those abstraction substitutions that are actually possible in the fixed point. For the post-conditions we are then checking that the fixed point covers everything that the transaction produces under those same substitutions: fixed-point coverage is thus defined as follows:

Definition 8 (Fixed-point coverage) *Let* T *be a transaction and let* $\mathsf{FP} = (FP, TI)$ *be a fixed point. We say that* FP covers T *iff for all abstraction substitutions* δ *with domain* $fv(T)$*, if* $\mathsf{pre}(\mathsf{FP}, T, \delta)$ *then* $\mathsf{post}(\mathsf{FP}, T, \delta)$*. For a protocol* $\mathcal{P}$ *we say that* FP covers $\mathcal{P}$ *iff* FP *covers all transactions of* $\mathcal{P}$*.*

With this defined we can prove the following theorem:

Theorem 1 [7] *Let* $\mathcal{P}$ *be a protocol and let* FP *be a fixed point. If* attack *does not occur in* FP*, and if* $\mathcal{P}$ *is covered by* FP*, then* $\mathcal{P}$ *is secure.*

Example 5 *Consider the key update transaction* $\mathsf{keyUpdateServer}$ *from Subsection 3.1. We now show that the fixed point* FP_{ks} *defined in Example 4.3 covers this transaction, i.e., satisfies Definition 8.*

The only variables occurring in $\mathsf{keyUpdateServer}$ *are* PK *and* NPK*, so we can begin by finding the abstraction substitutions with domain* $\{PK, NPK\}$ *that satisfy the pre-conditions given in Definition 5. We denote by* Δ *the set of those substitutions. Afterwards we show that all* $\delta \in \Delta$ *satisfy the post-conditions given in Definition 7.*

The variables PK *and* NPK *are not declared as fresh in* $\mathsf{keyUpdateServer}$ *so condition F4 is vacuously satisfied. From F2 and F3 we know that* $\mathsf{valid(a)} \in \delta(PK)$ *and* $\mathsf{valid(a)}, \mathsf{revoked(a)} \notin \delta(NPK)$*, for all* $\delta \in \Delta$*. From F1 we know that* $cl_{TI_{ks}}(FP_{ks}) \vdash \delta(\mathsf{sign}(\mathsf{inv}(PK), \mathsf{pair}(\mathsf{a}, NPK)))$*. The intruder cannot compose the signature himself since he cannot derive a private key of the form* $\mathsf{inv}(b)$ *where* $b \in \mathbb{A}$ *and* $\mathsf{valid(a)} \in b$*. Hence the only signatures available to him—that also satisfy the constraints for* Δ *that we have deduced so far—are* $\mathsf{sign}(\mathsf{inv}(\{\mathsf{valid(a)}\}), \mathsf{pair}(\mathsf{a}, b))$ *for each* $b \in \{\{\mathsf{ring(a)}\}, \emptyset\}$*. The only surviving substitutions are*

$$\delta^1 = [PK \mapsto \{\mathsf{valid(a)}\}, NPK \mapsto \emptyset], \text{ and}$$
$$\delta^2 = [PK \mapsto \{\mathsf{valid(a)}\}, NPK \mapsto \{\mathsf{ring(a)}\}].$$

[7] This theorem is called `protocol_secure` in the Isabelle code and can be found in the `Stateful_Protocol_Verification.thy` theory file.

That is, $\Delta = \{\delta^1, \delta^2\}$.

Next, we compute the updated substitutions w.r.t. the transaction keyUpdateServer*:*

$$\delta_u^1 = [PK \mapsto \{\mathsf{revoked(a)}\}, NPK \mapsto \{\mathsf{valid(a)}\}], \text{ and}$$
$$\delta_u^2 = [PK \mapsto \{\mathsf{revoked(a)}\}, NPK \mapsto \{\mathsf{ring(a)}, \mathsf{valid(a)}\}].$$

Now we can verify that conditions G1 and G2 hold for δ^1 *and* δ^2*: We have that* $\delta^i(x) \twoheadrightarrow \delta_u^i(x)$ *is covered by* TI_{ks}*, for all* $i \in \{1,2\}$ *and all* $x \in \{PK, NPK\}$*. We also have that the outgoing message* $\mathsf{inv}(PK)$ *is in* $cl_{TI_{ks}}(FP_{ks})$ *under each* δ_u^i*. Thus* keyUpdateServer *is covered by* FP_{ks}*.*

We can, in a similar fashion, verify that the remaining transactions of the keyserver protocol are covered by the fixed point. Thus the keyserver protocol is covered by FP_{ks}*.* □

5.2 Automatic Fixed-Point Computation

An interesting consequence of the coverage check is that we can also use it to compute a fixed point for protocols $\mathcal{P}$. In a nutshell, we can update a given a fixed-point candidate FP_0 for $\mathcal{P}$ as follows: For each transaction of $\mathcal{P}$ we first compute the abstraction substitutions Δ that satisfy the pre-conditions F1 to F4. Secondly, we use the post-conditions G1 and G2 to compute the result of taking T under each $\delta \in \Delta$ and add those terms and term implications to FP_0. Starting from an empty initial iterand $(\emptyset, \emptyset)$ we can then iteratively compute a fixed point for $\mathcal{P}$. Definition 9 gives a simple method to compute protocol fixed points based on this idea.

Definition 9 *Let* $\mathcal{P}$ *be a protocol and let* f *be the function defined as follows:*

$$f((FP, TI)) \equiv (FP \cup \{t \in \widehat{FP}_\delta^T \mid T \in \mathcal{P}, \delta \in \Delta_{FP,TI}^T\},$$
$$TI \cup \{ab \in \widehat{TI}_\delta^T \mid T \in \mathcal{P}, \delta \in \Delta_{FP,TI}^T\})$$

where

$$\Delta_{FP,TI}^T \equiv \{\delta \mid dom(\delta) = fv(T), \mathsf{pre}((FP, TI), T, \delta)\}$$
$$\widehat{FP}_\delta^T \equiv \{\delta_u(t) \mid \mathsf{send}\ t \text{ occurs in } T\}$$
$$\widehat{TI}_\delta^T \equiv \{(\delta(x), \delta_u(x)) \mid x \in fv(T) \setminus fresh(T)\}$$

Then we can compute a fixed point for $\mathcal{P}$ *by computing a fixed point of* f*, e.g., by computing the least* $n \in \mathbb{N}$ *such that* $f^n((\emptyset,\emptyset)) = f^{n+1}((\emptyset,\emptyset))$*.*

We provide, as part of our Isabelle formalization, a function to compute such a fixed point (with some optimizations to avoid computing terms and term implications that are subsumed by the remaining fixed point), using the built-in code generation functionality of Isabelle.

6 Improving the Coverage Check

We now describe a number of improvements that are essential to an efficient check (small experiments show that without these, performance is quite poor

even in minimal examples). We emphasize again that even if we had introduced mistakes here, it would not affect the correctness of the entire approach, since in the worst case the proofs would be rejected by Isabelle.

There are two major issues that make the coverage check from the previous section quite inefficient when implemented directly. One concerns the fact that the fixed point should be considered closed under intruder deduction and term implication. Even though the typed model allows us to keep even the intruder deduction closure finite, explicitly computing the closure is not feasible even on rather modest examples. The second issue is about the abstraction substitutions δ of the check: recall that in the check we defined above, for a given transaction we consider *every* substitution δ of the variables with abstract values, which is of course exponential both in the number of variables and the number of sets.

Let us first deal with this second issue. We can indeed compute exactly those substitutions that satisfy conditions F2 to F4: every positive set-membership check x in s of the transaction requires that $s \in \delta(x)$, and similarly for the negative case. Moreover, $\delta(x)$ can be only an abstract value that actually occurs in the fixed point. Starting from these constraints often substantially cuts down the number of substitutions δ that we need to consider in the check, especially when we have more agents than in the example. This is because typically (at least in a good protocol) most values will not be members of many sets that belong to different agents (but rather just a few that deal with that particular value).

The first issue, i.e., avoiding computing the term implication closure $cl_{TI}(FP)$ when performing intruder deductions, is more difficult. The majority of this section is therefore dedicated to improving on conditions F1 and G2 so that we can avoid computing the entire closure $cl_{TI}(FP)$—only in a few corner cases, we need to compute the closure for a few terms of FP. A key to that is to saturate the intruder knowledge with terms that can be obtained by analysis and then work with composition only, i.e., $\vdash_c$.

6.1 Intruder Deduction Modulo Term Implications

Recall that $\vdash_c$ is the intruder deduction without analysis, i.e., only the (*Axiom*) and (*Compose*) rules. We first consider how we can handle in this restricted deduction relation the term implication graph TI efficiently, i.e., how to decide $cl_{TI}(M) \vdash_c t$ (for given TI, M and t) without computing $cl_{TI}(M)$. In a second step we then show how to handle also analysis, i.e., the full $\vdash$ relation.

In fact it boils down to checking the side condition of (*Axiom*), i.e., in our case, whether $t \in cl_{TI}(M)$, without having to compute $cl_{TI}(M)$ first. (The composition rule is then easier because it does not "directly look" at the knowledge.) For this, it is sufficient if we can check whether $t \in cl_{TI}(t')$ for any $t' \in M$, without having to compute $cl_{TI}(t')$.

Consider again Definition 3. We can use this to derive a recursive check function $t' \rightsquigarrow_{TI} t$ for the question $t \in cl_{TI}(t')$: it can only hold if either

- t and t' are the same variable,

- or t, t' are abstract values with a path from t' to t in TI,
- or $t = f(t_1, \ldots, t_n)$ and $t' = f(t'_1, \ldots, t'_n)$, where recursively $t'_i \rightsquigarrow_{TI} t_i$ holds for all $1 \leq i \leq n$.

With this we can now define a recursive function $\Vdash_c$ that checks for given M, TI, and t whether $cl_{TI}(M) \vdash_c t$ without computing $cl_{TI}(M)$, defined as follows:

$$\begin{aligned} M \Vdash_c^{TI} t \text{ iff } & (\exists t' \in M.\ t' \rightsquigarrow_{TI} t) \text{ or} \\ & t \text{ is of the form } t = f(t_1, \ldots, t_n) \text{ where} \\ & f \in \Sigma_{pub}^n \text{ and } M \Vdash_c^{TI} t_i \text{ for all } i \in \{1, \ldots, n\} \end{aligned}$$

This function indeed fulfills its purpose:

Lemma 1 $cl_{TI}(M) \vdash_c t$ *iff* $M \Vdash_c^{TI} t$

Next, we show how to reduce the intruder deduction problem $\vdash$ to the restricted variant $\vdash_c$.

6.2 Analyzed Intruder Knowledge

The idea is now that $\vdash_c$ is actually already sufficient, if we have an analyzed intruder knowledge: we define that a knowledge M is *analyzed* iff $M \vdash t$ implies $M \vdash_c t$ for all t. More in detail, we can consider a knowledge M that is saturated by adding all subterms of M that can be obtained by analysis. Then M is analyzed, i.e., we do not need any further analysis steps in the intruder deduction. This is intuitively the case because the intruder cannot learn anything from analyzing messages he has composed himself.

We now define formally what if means for a term t to be analyzed using the keys ($\mathsf{Keys}(t)$) and results ($\mathsf{Result}(t)$) from the analysis as defined in Subsection 2.2:

Definition 10 (Analyzed term) *Let M be a set of terms and let t be a term. We then say that t is* analyzed *in M iff $M \vdash_c \mathsf{Keys}(t)$ implies $M \vdash_c \mathsf{Result}(t)$ (where $M \vdash_c N$ for sets of terms M and N is a short-hand for $\forall t \in N.\ M \vdash_c t$).*

The following lemma then provides us with a decision procedure for determining if a knowledge is analyzed:

Lemma 2 *M is analyzed iff all $t \in M$ are analyzed in M.*

We now consider again an intruder knowledge given as the term implication closure of a set of messages, i.e., $cl_{TI}(M)$ instead of M. Efficiently checking whether an intruder knowledge's term implication closure is analyzed, without actually computing it, is challenging. The following lemma shows that if we can derive the results of analyzing a term t in the knowledge M then we can also derive the results of analyzing any implied term $t' \in cl_{TI}(t)$:

Lemma 3 *Let $t \in M$. If $cl_{TI}(M) \vdash_c \mathsf{Result}(t)$ then for all $t' \in cl_{TI}(t)$, $cl_{TI}(M) \vdash_c \mathsf{Result}(t')$.*

Therefore, if all $k \in \mathsf{Keys}(t)$ can be derived *and* t is analyzed in $cl_{TI}(M)$ then we can conclude that all implied terms $t' \in cl_{TI}(t)$ are analyzed in $cl_{TI}(M)$. If, however, some of the keys for t are not derivable then we are forced to check the implied terms as well as the following example shows:

Example 6 *Let $f, g \in \Sigma^1_{priv}$, $TI = \{a \twoheadrightarrow b\}$, and $M = \{f(a), g(b)\}$. Define the analysis rules $\mathsf{Ana}_f(x) = (\{g(x)\}, \{x\})$ and $\mathsf{Ana}_g(x) = (\emptyset, \emptyset)$. Then $cl_{TI}(M) = \{f(b)\} \cup M$. The term $f(a)$ is analyzed in $cl_{TI}(M)$ because the key $g(a)$ cannot be derived: $cl_{TI}(M) \nvdash_c g(a)$. However, $f(a) \,_a\!\!\twoheadrightarrow_b f(b)$ and $f(b)$ is not analyzed in $cl_{TI}(M)$: $\mathsf{Ana}(f(b)) = (\{g(b)\}, \{b\})$ but the key $g(b)$ is derivable from $cl_{TI}(M)$ in $\vdash_c$ whereas the result b is not. Thus $cl_{TI}(M)$ is not an analyzed knowledge.* $\square$

So in most cases we can efficiently check if $cl_{TI}(M)$ is analyzed, and in some cases we need to also compute the term implication closure $cl_{TI}(t)$ of problematic terms $t \in M$ (but not necessarily compute all of $cl_{TI}(M)$):

Lemma 4 *$cl_{TI}(M)$ is analyzed iff for all $t \in M$, the following holds*

> **if** $cl_{TI}(M) \vdash_c \mathsf{Keys}(t)$ **then** *t is analyzed in $cl_{TI}(M)$*
> **else** *all $t' \in cl_{TI}(t)$ are analyzed in $cl_{TI}(M)$.*

This lemma also provides us with the means to *extend* a knowledge M to one whose term implication closure is analyzed: The idea is to close M under the rule that extends it with the result $\mathsf{Result}(t)$ of those analyzable terms $t \in M$ for which the conditions on the right-hand side of the biconditional in Lemma 4 fails. For instance, in Example 6 we need to extend $M = \{f(a), g(b)\}$ with b, resulting in the analyzed knowledge $M' = \{f(a), g(b), b\}$.

7 Proof of Concept

Table 1 shows the fixed-point sizes of various example protocols together with measurements of the elapsed real time it takes to generate and verify the Isabelle specifications. First, we report the time for translating the protocol specifications into Isabelle/HOL (Translation), the time for showing that the given protocol is an instance of the formal protocol model (Setup), and the time for computing the fixed-point and its size. In the last three columns, we report the run-time of three different strategies for the security proof: *Safe* employs symbolic evaluation using Isabelle's simplifier *code-simp*. In this configuration, all proof steps are checked by Isabelle's LCF-style kernel. *NBE* employs normalization by evaluation, a technique that uses a partially symbolic evaluation approach that, to a limited extend, relies on Isabelle's code generator. Finally, *Unsafe* is an approach that directly employs the code generator and internally uses the proof method *eval*. In general, the configurations *NBE* and *Unsafe* require the user to trust the code generator. While Isabelle's code generator is thoroughly tested, it is not formally verified. We mainly provide these configurations to provide faster alternatives during interactive protocol explorations.

Protocol	Initialization		Fixed-Point			Verification		
	Translation	Setup	Computation	|FP|	|TI|	Safe	NBE	Unsafe
Keyserver_2_1	00:00:04	00:00:09	00:00:04	22	27	00:00:45	00:00:10	00:00:07
Keyserver_3_1	00:00:05	00:00:10	00:00:04	31	40	00:01:56	00:00:14	00:00:07
Keyserver_4_1	00:00:05	00:00:09	00:00:04	40	53	00:04:41	00:00:24	00:00:07
Keyserver2_2_1	00:00:05	00:00:10	00:00:04	9	4	00:00:28	00:00:09	00:00:07
Keyserver2_3_1	00:00:05	00:00:10	00:00:04	12	6	00:00:41	00:00:09	00:00:07
Keyserver2_4_1	00:00:05	00:00:10	00:00:04	15	8	00:00:58	00:00:10	00:00:07
Keyserver_Composition_2_1	00:00:07	00:00:10	00:00:04	40	105	00:37:09	00:02:12	00:00:08
Keyserver_Composition_3_1	00:00:07	00:00:11	00:00:05	56	153	02:58:28	00:08:26	00:00:09
Keyserver_Composition_4_1	00:00:07	00:00:10	00:00:08	70	201	10:02:24	00:22:45	00:00:09
TLS12simp	00:00:08	00:00:10	00:00:21	48	20	timeout	08:57:43	00:00:13
NSLclassic	00:00:05	00:00:10	00:00:06	43	6	00:05:27	00:00:13	00:00:08
NSPKclassic	00:00:05	00:00:10	00:00:06	69	6	attack	attack	attack
PKCS_Model03	00:00:05	00:00:10	00:00:05	8	2	00:00:14	00:00:09	00:00:07
PKCS_Model07	00:00:11	00:00:11	00:00:07	15	5	01:06:11	00:01:36	00:00:11

Table 1: Runtime Measurements (Time Format: *hh:mm:ss*). Experiments that took longer than 18 hours are marked with "timeout".

Ultimately, it is up to the user to decide which approach to use, preferably after consulting [20], which discusses the software stack that needs to be trusted in each of these configurations in more detail.

All experiments have been conducted on a Linux server with an Intel Xeon E5-2680 CPU and 256GB main memory. Our implementation provides an option to measure the time required for executing individual "top-level" commands (e.g., `protocol_security_proof`). We only report the times that are specific to the individual protocols using a "pre-compiled" session that contains our generic protocol translator as well as the protocol-independent formalizations and proofs. Compiling this session takes ca. 4 minutes and 30 seconds.

The example Keyserver_h_d is our running keyserver example for h honest agents and d dishonest agents.[8] The example Keyserver_Composition_h_d with h honest agents and d dishonest agents is inspired by [22] where another keyserver protocol—named Keyserver2_h_d here—runs in parallel on the same network and where databases are shared between the protocols.

We made further experiments where our focus is not the precise modeling and verification of particular protocols, but rather to experiment with our method on more complex examples and get an understanding of how our method scales.

With TLS12simp we have looked at one practical protocol, TLS 1.2, with

[8] We verify here a generalized version of the keyserver example (as compared to the running example): we include dishonest agents who can participate in the protocol. This also requires that agents maintain a set of deleted keys, because otherwise the abstraction $\emptyset$ leads to false attacks.

two honest agent and one dishonest agent, albeit with some simplifications, in particular modeling only one variant of the flow and simplifying the hashing.

NSLclassic and NSPKclassic are based on the NSL and Needham–Schroeder protocol specifications shipped with AIF-ω [30].

Finally, scenario 3 and 7 (PKCS#11_3 and PKCS#11_7), from the "PKCS#11" model that is distributed with AIF-ω [30] are examples of another flavor of stateful protocols, namely security tokens that can store keys and perform encryption and decryption and with which the intruder can interact through an API. Generally modeling such tokens and their APIs works quite well with the set-based abstraction. We report only two scenarios as they are the only ones that do not lead to an attack. In fact there is a third one (scenario #9) that is marked as correct in the AIF-ω distribution, but that is actually due to a mistake that our attempt to verify it in Isabelle has revealed. We discuss this example in more detail in the appendix. This illustrates our main point that there can be surprises when one tries to verify in Isabelle the results of automated tools.

8 Conclusion and Related Work

The research into automated verification of security protocols resulted in a large number of tools (e.g., [9, 10, 15, 4, 17]). The implementation of these tools usually focuses on efficiency, often resulting in very involved verification algorithms. The question of the correctness of the *implementation* is not easy to answer and this is in fact one motivation for research in using LCF-style theorem provers for verifying protocols (e.g., [32, 24, 13, 5, 6, 7]). While these works provide a high level of assurance into the correctness of the verification result, they are usually interactive, i.e., the verification requires a lot of expertise and time.

This trade-off between the trustworthiness of verification tools and the degree of automation inspired research of combining both approaches [19, 11, 26]. Goubault-Larrecq [19] considers a setting where the protocol and goal are given as a set S of Horn clauses; the tool output is a set S_∞ of Horn clauses that are in some sense saturated and such that the protocol has an attack iff a contradiction is derivable. His tool is able to generate proof scripts that can be checked by Coq [8] from S_∞. Meier [26] developed Scyther-proof [27], an extension to the backward-search used by Scyther [17], which is able to generate proof scripts that can be checked by Isabelle/HOL [31]. Brucker and Mödersheim [11] integrate an external automated tool, OFMC [4], into Isabelle/HOL. OFMC generates a witness for the correctness of the protocol that is used within an automated proof tactic of Isabelle.

Our work generalizes on these existing approaches for automatically obtaining proofs in an LCF-style theorem prover, first and foremost by the support for stateful protocols and thus a significantly larger range of protocols—moving away from simple isolated sessions to distributed systems with databases, or devices that have a long-term storage.

We achieve this by employing the abstraction-based verification technique of AIF [29], but with an important modification. The method of AIF produces

a set of Horn clauses that is then analyzed with ProVerif [9] (or SPASS[36]), and the same holds true also for several similar methods for stateful protocol verification, namely StatVerif [3], Set-π [12], AIF-ω [30] and GSVerif [14]. Note that definite Horn clauses in first-order predicate logic always have a trivial model (interpret all predicates as true for all arguments), and we are actually interested in the free model (free algebra for the functions and least model of the predicates). This is achieved in ProVerif (and SPASS) by checking whether the Horn clauses *imply* a given attack predicate. If they do, then the attack predicate is true also in the free model. If they do not, i.e., if the Horn clauses are *consistent* with the negation of the attack predicate, then the attack predicate is not true in all models, and in particular not in the free model since it is the *least* model. Thus, in a positive verification, the result from ProVerif is a consistent saturated set of Horn clauses. As first remarked by Goubault-Larrecq [19], this is not a very promising basis for a proof, as one does not get a derivation of a formula (the way SPASS for instance is often used in combination with Isabelle) but rather a failure to conclude a proof goal. The only chance to verify the resulting saturated set of Horn clauses, is to recompute the saturation and compare. Therefore [19] uses a different idea: showing that the Horn clauses and the negation of the attack predicate are consistent by trying to find some *finite* model and, if found, then using this finite model to generate a proof in Coq that the Horn clauses are consistent with the negation of the attack predicate.

The limitation of [19] is that it checks the protocol proofs only on the Horn clause level, i.e., after a non-trivial abstraction has been applied. In order to obtain Isabelle proofs for the original unabstracted stateful protocols, we use therefore another approach: rather than Horn clauses, we directly generate a fixed point of abstract facts that occur in any reachable state. This would in fact normally not terminate on most protocols due to the intruder deduction; however, we employ here the typing result we have formalized in Isabelle [25] to ensure that the fixed point is always finite and our method is in fact guaranteed to terminate. This fixed point, if it does not contain the attack predicate, is the core of a correctness proof for the given protocol, namely as an invariant that the fixed point covers everything that can happen and we essentially have to check that this invariant indeed holds for every transition rule of the protocol.

An interesting difference to previous approaches is that we do not rely on an external tool for the generation of the proof witness, but that it is implemented within Isabelle itself. The reason is more of a practical than a principle matter: Computing the fixed point in Isabelle is actually not difficult and—thanks to Isabelle's code generation—without much of a performance penalty; however, the fact that we do not rely on an external tool for the generation of the proof witness reduces the chances of synchronization and update problems (e.g., with new Isabelle versions). In fact, this work is part of the Archive of Formal Proofs[9], a collection of Isabelle proofs that are kept up to date with each new version of Isabelle. This means that for each protocol that works in today's version it is highly likely that the proof works in future versions, because the proofs of

[9] See `https://www.isa-afp.org`.

all theorems of our (protocol-independent) Isabelle theory will be updated, and the fixed point and the checks about it do not have to change. Thus we will also automatically benefit from all advances of Isabelle.

Another difference to previous approaches is that we do not directly generate proof scripts that Isabelle has to then check. Rather, we have a fixed (protocol-independent) set of theorems that imply that any protocol is secure if we have computed a fixed-point representation that gives an upper bound of what (supposedly) can happen and this representation passes a number of checks. These checks can either be done by generated code or entirely within Isabelle's simplifier. Especially with the generated code we have a substantial performance advantage, while using Isabelle's simplifier gives the highest level of assurance since we only rely on the correctness of the Isabelle kernel. We note that also the generated code is correct "by construction" and thus extremely unlikely to compute wrong results. Many small practical advantages arise from the integration: We do not have an overhead of parsing of proof scripts (which can be substantial for a larger fixed point). By using the internal API of Isabelle, we avoid the need for the Isabelle front-end parser to parse and type-check the fixed point (as we can directly generate a typed fixed-point on the level of the abstract syntax tree). Parsing and type-checking (on the concrete syntax level) of large generated theories (as, e.g., ones containing the generated fixed point) is, in fact, slow in Isabelle [11].

Another point is that there exist a number of protocol verification methods and results that use slightly different models. Here we actually seamlessly integrate a verification method into a rich Isabelle theory of protocols without any semantic gaps: We provide here a method that is integrated into a large framework of Isabelle theories for protocols (approximately 25,000 lines of code), in particular a typing and compositionality result. This allows for instance to prove manually (in the typed model) the correctness of a protocol, use our automated method to prove the correctness of a different protocol, and then compose the proof to obtain the correctness of the composition in an untyped model. This seamless integration of results without semantic gaps between tools we consider as an important benefit of this approach. Even though many protocol models are not substantially different from each other, bridging over the small differences can be very hard to do, especially in a theorem prover that prevents one from glossing over details. Our deep integration into the existing formalization of security protocols in Isabelle ensures that the same protocol model (same semantics) is used—which would otherwise require additional work (e.g., to ensure that the semantics of the protocol specified in a tool such as Scyther-proof is faithfully represented in the generated Isabelle theory).

It is in general desirable to have proofs that are not only machine-checked but also human-readable. A reason is that, for instance, mistakes in the specification itself (e.g., a mistake in a sent message so that it cannot be received by anybody) may lead to trivial security proofs which a human may notice when trying to understand the proof. Here Scyther-proof has the benefit that it produces very readable Isar-style proofs; in our case, there is, however, something that is also accessible: the fixed point that was computed is actually a high-level proof

idea that is often quite readable as well (see for instance our running example). Moreover, the entire set of protocol-independent theorems are hand-written Isar-style proofs.

Furthermore, our work shares a lot of conceptual similarities with Tamarin [28] in the sense that we also provide a protocol verification environment that allow for the seamless transition between a fully automated verification and an interactive verification approach. As the interactive verification component of our tool is based on Isabelle/HOL, the user can make use of all available Isabelle features, including its generic proof automation tools such as sledgehammer [34]. In contrast, Tamarin is a domain-specific theorem prover whose implementation is not based on a generic, widely reviewed, interactive theorem prover. As its design is, in our understanding, not based on an LCF approach, the risks of bugs in Tamarin resulting in wrong verification results is potentially higher compared to a tool following an LCF approach. As Tamarin shares ideas with Scyther, generating Isabelle proofs from Tamarin using a similar approach as Scyther-proof should in principle be possible. While this would be a very interesting extension—enabling automated or semi-automated support for a very large class of protocols—it does not seem immediate to achieve.

Finally, another approach that, like Tamarin, is very much related to performing actual proofs of security protocols automatically and semi-automatically is CPSA [18, 35]. Also here it might be possible to make a connection to a theorem prover of Isabelle; however, the approach is even further away from our approach than Tamarin, because CPSA does not necessarily assume a closed world of transactions. Rather, it performs an enrich-by-need analysis obtaining all ways to complete a particular scenario and thereby yielding the strongest security goals a given system would satisfy (even in the presence of other transactions). We believe it is even more challenging to integrate this kind of reasoning into a theorem prover like Isabelle, but achievable. We like to investigate this as future work as it could give interesting ways for an analyst to interact with the proving process and inject proof ideas.

Acknowledgments

We thank Joshua Guttman and the anonymous reviewers for helpful feedback on this paper. This work was supported by the Sapere-Aude project "Composec: Secure Composition of Distributed Systems", grant 4184-00334B of the Danish Council for Independent Research, by the EU H2020 project no. 700321 "LIGHTest: Lightweight Infrastructure for Global Heterogeneous Trust management in support of an open Ecosystem of Trust schemes" (lightest.eu) and by the "CyberSec4Europe" European Union's Horizon 2020 research and innovation programme under grant agreement No 830929.

References

[1] Almousa, O., Mödersheim, S., Modesti, P., Viganò, L.: Typing and compositionality for security protocols: A generalization to the geometric fragment. In: European Symposium on Research in Computer Security. pp. 209–229 (2015)

[2] Arapinis, M., Duflot, M.: Bounding messages for free in security protocols - extension to various security properties. Information and Computation **239**, 182–215 (2014)

[3] Arapinis, M., Phillips, J., Ritter, E., Ryan, M.D.: Statverif: Verification of stateful processes. Journal of Computer Security **22**(5), 743–821 (2014)

[4] Basin, D.A., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. International Journal of Information Security **4**(3), 181–208 (2005)

[5] Bella, G., Butin, D., Gray, D.: Holistic analysis of mix protocols. In: Information Assurance and Security. pp. 338–343 (2011)

[6] Bella, G.: Formal Correctness of Security Protocols. Information Security and Cryptography, Springer (2007)

[7] Bella, G., Massacci, F., Paulson, L.C.: Verifying the SET purchase protocols. Journal of Automated Reasoning **36**(1-2), 5–37 (2006)

[8] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)

[9] Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Computer Security Foundations Workshop. pp. 82–96 (2001)

[10] Boichut, Y., Héam, P.C., Kouchnarenko, O., Oehl, F.: Improvements on the Genet and Klay technique to automatically verify security protocols. In: Automated Verification of Infinite-State Systems. pp. 1–11 (2004)

[11] Brucker, A.D., Mödersheim, S.: Integrating automated and interactive protocol verification. In: Formal Aspects in Security and Trust. pp. 248–262 (2009)

[12] Bruni, A., Mödersheim, S., Nielson, F., Nielson, H.R.: Set-π: Set membership π-calculus. In: Computer Security Foundations Symposium. pp. 185–198 (2015)

[13] Butin, D.F.: Inductive analysis of security protocols in Isabelle/HOL with applications to electronic voting. Ph.D. thesis, Dublin City University (2012)

[14] Cheval, V., Cortier, V., Turuani, M.: A little more conversation, a little less action, a lot more satisfaction: Global states in ProVerif. In: Computer Security Foundations Symposium. pp. 344–358 (2018)

[15] Chevalier, Y., Vigneron, L.: Automated Unbounded Verification of Security Protocols. In: Computer Aided Verification. pp. 325–337 (2002)

[16] Chrétien, R., Cortier, V., Dallon, A., Delaune, S.: Typing messages for free in security protocols. ACM Transactions on Computational Logic **21**(1), 1:1–1:52 (2020)

[17] Cremers, C.: Scyther: Semantics and verification of security protocols. Ph.D. thesis, Eindhoven University of Technology (2006)

[18] Doghmi, S.F., Guttman, J.D., Thayer, F.J.: Searching for shapes in cryptographic protocols. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 523–537 (2007)

[19] Goubault-Larrecq, J.: Towards producing formally checkable security proofs, automatically. In: Computer Security Foundations Symposium. pp. 224–238 (2008)

[20] Haftmann, F., Bulwahn, L.: Code generation from Isabelle/HOL theories (2020), `http://isabelle.in.tum.de/doc/codegen.pdf`

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

[22] Hess, A., Mödersheim, S., Brucker, A.: Stateful protocol composition. In: European Symposium on Research in Computer Security. pp. 427–446 (2018)

[23] Hess, A.V., Mödersheim, S., Brucker, A.D., Schlichtkrull, A.: Automated stateful protocol verification. Archive of Formal Proofs (Apr 2020), `http://isa-afp.org/entries/Automated_Stateful_Protocol_Verification.html`, Formal proof development

[24] Hess, A.V., Mödersheim, S.: Formalizing and proving a typing result for security protocols in Isabelle/HOL. In: Computer Security Foundations Symposium. pp. 451–463 (2017)

[25] Hess, A.V., Mödersheim, S.: A typing result for stateful protocols. In: Computer Security Foundations Symposium. pp. 374–388 (2018)

[26] Meier, S., Cremers, C., Basin, D.A.: Efficient construction of machine-checked symbolic protocol security proofs. Journal of Computer Security **21**(1), 41–87 (2013)

[27] Meier, S., Cremers, C.J.F., Basin, D.A.: Strong invariants for the efficient construction of machine-checked protocol security proofs. In: Computer Security Foundations Symposium. pp. 231–245 (2010)

[28] Meier, S., Schmidt, B., Cremers, C., Basin, D.A.: The TAMARIN prover for the symbolic analysis of security protocols. In: Computer Aided Verification. pp. 696–701 (2013)

[29] Mödersheim, S.: Abstraction by set-membership: verifying security protocols and web services with databases. In: Computer and Communications Security. pp. 351–360 (2010)

[30] Mödersheim, S., Bruni, A.: AIF-ω: Set-based protocol abstraction with countable families. In: Principles of Security and Trust. pp. 233–253 (2016)

[31] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science, Springer (2002)

[32] Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security **6**(1-2), 85–128 (1998)

[33] Paulson, L.C.: Inductive analysis of the internet protocol TLS. ACM Transactions on Information and System Security **2**(3), 332–351 (1999)

[34] Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: International Workshop on the Implementation of Logics. pp. 1–11 (2010)

[35] Rowe, P.D., Guttman, J.D., Liskov, M.D.: Measuring protocol strength with security goals. International Journal of Information Security **15**(6), 575–596 (2016)

[36] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Conference on Automated Deduction. pp. 140–145 (2009)

A problem with the AIF-ω specification `09-lost_key_att_countered.aifom`

When we tried to model this specification from the AIF-ω distribution, which is classified as secure by the AIF-ω tool, we failed to prove it secure with our approach in Isabelle, and in fact, our fixed-point generation was generating the `attack` constant. Going back to the AIF-ω verification we noticed that there was a problem with the public functions, in this case symmetric encryption and hashing. They were declared as public in the AIF-ω specification, but the intruder seemed unable to make use of them and get to the attack we had obtained.

In fact the problem was that AIF-ω does not generate intruder rules for the function symbols that are declared as public, so unless the user explicitly

states rules like "if the intruder knows x then he also knows $h(x)$)", the function symbol is like a private one that the intruder cannot apply himself. When we add appropriate rules for all public function symbols to the specification, also AIF-ω finds the attack.

One could argue that this is a problem of the specification (the modeler was in fact aware of this behavior), however, it can be considered a bug of AIF-ω, since the keyword "public" for a function symbol at least suggests that the composition rule would be automatically included. In this sense, our Isabelle verification has revealed a mistake, in particular one that has led to an erroneous "verification" of a flawed protocol by an automated tool. In fact, the attack is not a false positive (i.e., the original specification also has an attack).

Isabelle/PSPSP

We implemented our approach on top of Isabelle/HOL. This includes a formalization of the protocol model in Isabelle/HOL, a data type package that provides a domain specific language (called `trac`) for specifying security protocols, and fully automated proof support.

Figure 1 shows the Isabelle IDE (called Isabelle/jEdit). The upper part of the window is the input area that works similar to a programming IDE, i.e., supporting auto completion, syntax highlighting, and automated proof generation and interactive proof development. The lower part shows the current output (response) with respect to the cursor position. In more detail, Figure 1 shows the specification, and both the fully-automated and the interactive verification of a toy keyserver protocol:

- The protocol is specified using a domain-specific language that, e.g., could also be used by a security protocol model checker (line 9–59). Our implementation automatically translates this specification into a family of formal HOL definitions. Moreover, basic properties of these definitions are also already proven automatically (i.e., without any user interaction): for this simple example, already 350 theorems are automatically generated.
- Next (line 62) our implementation automatically shows that the protocol satisfies the requirement of our model (Technically, this is done by instantiating several Isabelle locales, resulting in another 1750 theorems "for free.").
- In line 65, we compute the fixed point. We can use Isabelle's value-command (line 75) to inspect its size.

After these steps, all definitions and auxiliary lemmas for the security proof are available. We can now have two options:

1. we can do a fully automated proof (line 71). This top-level command proofs automatically a lemma showing the security of the defined protocol.

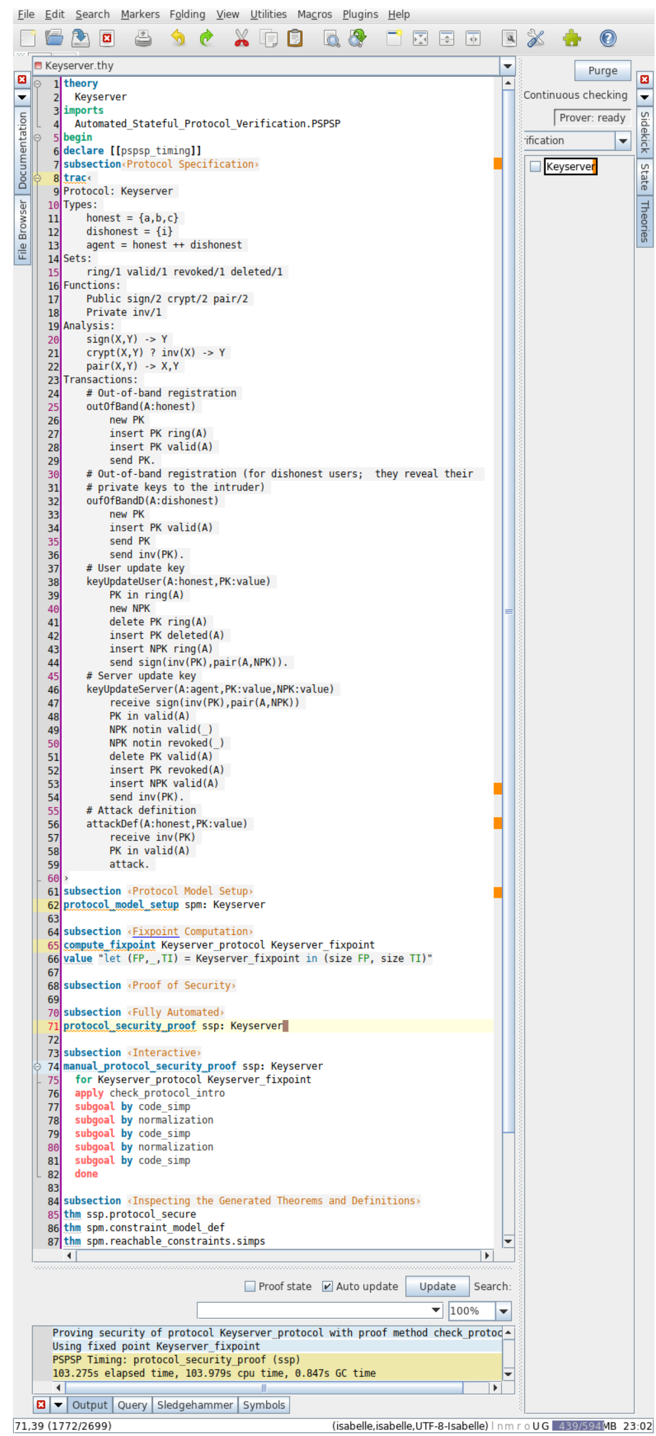

Figure 1: Using Isabelle/PSPSP for verifying a toy keyserver protocol.

2. we can interactively (manually) proof the security of the defined protocol. For this, we provide a top-level command (manual_protocol_security_proof, line 74) that generates a proof obligation that can be discharged by an interactively developed proof script (line 76–82).

Finally, we can inspect the theorem using the thm command (line 85 and following).

Summarizing, our implementation allows a non Isabelle-expert to specify security protocols and to verify them automatically. If the fully automated proof attempt fails, a seamless switch to an interactive proof attempt (requiring at least some knowledge of Isabelle/HOL) is possible.