PSPSP: A Tool for Automated Verification of Stateful Protocols in Isabelle/HOL Pre-Print

Andreas Viktor Hess [*] Sebastian Alexander Mödersheim [†]
Achim D. Brucker [‡] Anders Schlichtkrull [§]

April 24, 2024

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

[*]DTU Compute
[†]DTU Compute, `samo@dtu.dk`
[‡]University of Execter, `a.brucker@exeter.ac.uk`
[§]Aalborg University, `andsch@cs.aau.dk`

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. For instance, we have proved in Isabelle/HOL a compositionality result [24] for our protocol model: given a set of protocols for which we have proved security and that meet a number of requirement, then also their composition is correct. Since also the said requirements are proved in Isabelle, we arrive at a full security proof of the entire system checked by Isabelle. A result like this is beyond the scope of any standard verification tool. Note also that as part of the composition, some of the component protocols may be proved secure by different methods or even automatically.

Paulson [38] and Bella [5] developed a protocol model in Isabelle and performed several security proofs in this model, e.g., [39]. 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 [20, 11], as well as the tool Scyther-proof [31] for the backward search-based tool Scyther [18].

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 [33] 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. [23]. 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 [26], 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, 35]. 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.

Contributions Our main contribution is the formalization in Isabelle of the abstraction interpretation approach for stateful protocols as the PSPSP tool. In a nutshell, we have implemented in Isabelle the computation of the abstract fixed point—the proof idea so to speak—and how Isabelle can convince herself that this fixed point covers everything that can happen in the concrete protocol. The Isabelle security proof that one obtains 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 25,000 lines of Isabelle code (definitions, theorems and proofs).

A second contribution is the development and integration into Isabelle of a simple protocol specification language for stateful protocols that is based on a notion of atomic transactions: in a transaction, an entity may receive a message, consult its long-term database, make changes to the database and finally send out a reply. This language is more high-level than for instance multi-set rewriting while directly defining a state-transition system.

With respect to the conference version of this paper [27], we have made a number of improvements. First, we have improved the verification method itself. We have devised a novel method for checking the fixed-point coverage that significantly improves the runtime of many examples. We have also improved the check that fixed point is closed under decryption rules.

Second, we have connected the PSPSP tool to the compositional reasoning results of [23]: the transaction language now includes everything that is necessary to specify for protocol composition (most importantly, protocol interfaces and declassification for shared messages). The PSPSP tool now offers an automated check of the sufficient condition of compositionality. This allows for proving the security of a complex system in Isabelle entirely automatically, namely by using PSPSP to check the components and compositionality conditions, and then applying the composition theorem.

Third, related to that, one may of course prove only some components of a composition automatically with PSPSP and prove other components manually when they do not fall into the scope of what PSPSP can support. Also, one may integrate manual reasoning with automated PSPSP analysis: when the runtime of automated analysis are high, a human prover may have an idea to prove some aspect more easily avoiding, e.g., some lengthy enumerations. We give a case study of such an interactive proof that shows the potential of a methodology for combining automatic verification with human ingenuity.

Fourth, we have improved the user experience of the tool, e.g., by better error messages and support to understand attacks. A trace of derivation steps for the attack constant can be of great help: either this is a true attack and one can strengthen the protocol to prevent the attack, or it is a false positive induced by the over-approximation and this may give a hint how to refine the model of the protocol.

Fifth, while the transactions of a protocol specification immediately give rise to a state-transition systems there is a slight semantic gap between them namely that it is a necessity of the abstract interpretation approach that in the abstract transactions, every value occurs in a message (while in the transaction language specification a value may occur only in a set). This is without loss of generality, because one can make a transformation that for every newly generated value v generates a special message $\mathit{occurs}(v)$ and require that $\mathit{occurs}(v)$ holds for every non-fresh value v in a rule. The tool includes this transformation so the modelers do not have to make this encoding themselves. Now we have proved the soundness of this transformation. Another point is that the semantics of transactions is defined as symbolic traces ("lazy intruder") of unbounded length. This is particularly practical for relative soundness results like typing and compositionality. We have now also proved the equivalence with a more standard "ground" semantics.

Sixth, we have a major new case study from working with the Danish company Logos. In this case study we verify a protocol that the company is using for a travel card solution. The verification with PSPSP revealed a flaw in the protocol. After repairing the flaw, we were able to prove the security of the fixed protocol using PSPSP.

The complete formalization is available at the Archive of Formal Proofs as the entry titled *Automated Stateful Protocol Verification* [29]:

`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/pspsp.html`

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, Section 8 gives a short demonstration of PSPSP from the user's perspective (and discusses the application of the compositionality result [24]), Section 9 presents and reports on a case study where we apply PSPSP to a protocol by the Danish company Logos and finally Section 10 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 $f(t_1, \dots, t_n)$ 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.

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

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*, 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).

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 $x_1, \dots, x_n$ 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 and private symbols, respectively. By $\mathcal{C}_{pub}$ and $\mathcal{C}_{priv}$ we then denote the sets of public and private constants, respectively. 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 $[x_1 \mapsto t_1, \dots, x_n \mapsto t_n]$ to denote the substitution with domain $\{x_1, \dots, x_n\}$ and range $\{t_1, \dots, t_n\}$ 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 [23] 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:

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

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

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 $(\mathit{Compose})$ rules only.

The (Axiom) rule simply expresses that all messages directly known to the intruder are derivable, the $(\mathit{Compose})$ 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 $(\mathit{Decompose})$ 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* $\mathsf{inv}(k)$*. 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 [23] 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

[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 [23] on Ana such analysis rules would not lead to more expressive Ana functions.

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 [5, 7, 11, 38, 39] rely on such a typed model since it simplifies the protocol verification problem. There exist many typing results [16, 26, 1, 25, 22, 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 such result shows 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. [26] 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 $\mathsf{settype}$ and attack has type $\mathsf{attacktype}$. 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 [23] 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, [23] 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 called trac based on transactions with a bit of syntactic sugar, and this language is directly embedded into Isabelle. It is a simple text-based format directly accepted by our tool – see section 8. For now we introduce this language only at hand of a keyserver example adapted from [23] 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 $\mathsf{outOfBand}$ 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:

```
outOfBand($U$: user)
    new $PK$
    insert $PK$ ring($U$)
    insert $PK$ valid($U$)
    send $PK$.
```

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 ring(U) is a set of public keys controlled by U (and U has the corresponding private key of each) while 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 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 inv(PK) of the old key and sends this signature to the server by transmitting it over the network:

```
keyUpdateUser($U$: user, $PK$: value)
    $PK$ in ring($U$)
    new $NPK$
    delete $PK$ ring($U$)
    insert $NPK$ ring($U$)
    send sign(inv($PK$), pair($U$, $NPK$)).
```

The check PK in ring(U) represents here a non-deterministic choice of an element of ring(U). (Observe that a user can register any number of keys with the outOfBand transaction.) 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 revoked(U) containing the revoked keys of U:

```
keyUpdateServer($U$: user, $PK$: value, $NPK$: value)
    receive sign(inv($PK$), pair($U$, $NPK$))
    $PK$ in valid($U$)
    $NPK$ notin valid(_)
    $NPK$ notin revoked(_)
    delete $PK$ valid($U$)
    insert $PK$ revoked($U$)
    insert $NPK$ valid($U$)
    send inv($PK$).
```

As a last action, the old private key 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:

```
attackDef($U$: honest, $PK$: value)
    receive inv($PK$)
    $PK$ in valid($U$)
    attack.
```

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:

$$\begin{aligned}
S_r &::= \mathsf{receive}\ t_1, \ldots, t_n \cdot S_r \mid 0 \\
S_c &::= x\ \mathsf{in}\ s \cdot S_c \mid x\ \mathsf{notin}\ s \cdot S_c \mid x \not\doteq x' \cdot S_c \mid 0 \\
F &::= \mathsf{new}\ x \cdot F \mid 0 \\
S_u &::= \mathsf{insert}\ x\ s \cdot S_u \mid \mathsf{delete}\ x\ s \cdot S_u \mid 0 \\
S_s &::= \mathsf{send}\ u_1, \ldots, u_n \cdot S_s \mid 0
\end{aligned}$$

where $x, x' \in \mathcal{V}_{\mathsf{value}}$, $s \in \mathbb{S}$, $t_i \in \mathcal{T}(\mathbb{E} \cup \mathbb{F}, \mathcal{V}_{\mathsf{value}})$, $u_i \in \mathcal{T}(\mathbb{E} \cup \mathbb{F}, \mathcal{V}_{\mathsf{value}}) \cup \{\mathsf{attack}\}$, 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 $\mathsf{new}\ x$ occurs in T).

Protocols are defined as finite sets of such transactions $\mathcal{P} = \{T_1, \dots, T_n\}$. Their semantics is defined in terms of a ground transition system in which each configuration is of the form (M, D, C) where M is the intruder knowledge (the messages sent so far), D is a set 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 C keeps track of the constants that are no longer fresh. For a configuration and a transaction we can check if the transaction is executable from that configuration, and if so then there is a transition to the new configuration which results from executing the transaction. When executing a transaction, variables x occurring in $\mathsf{new}\ x$ actions will be instantiated with fresh values. This instantiation takes care of the $\mathsf{new}\ x$ actions which are then no longer needed. The instantiation also requires a slightly more flexible syntax compared to the transaction syntax, to allow for actions such as $\mathsf{insert}\ t\ s$ where $t \notin \mathcal{V}$. We introduce a syntax that accounts for this, called *constraints*:

$$\begin{aligned}\mathcal{A} ::= {} & \mathsf{send}\ t_1, \dots, t_n \cdot \mathcal{A} \mid \mathsf{receive}\ t_1, \dots, t_n \cdot \mathcal{A} \mid t \not\doteq t' \cdot \mathcal{A} \mid \mathsf{insert}\ t\ t' \cdot \mathcal{A} \mid \mathsf{delete}\ t\ t' \cdot \mathcal{A} \mid \\ & t\ \mathsf{in}\ t' \cdot \mathcal{A} \mid t\ \mathsf{notin}\ t' \cdot \mathcal{A} \mid 0\end{aligned}$$

where $t, t' \in \mathcal{T}(\Sigma, \mathcal{V})$ and where 0 is the empty constraint. Note also that in contrast to transactions, constraints are seen from the intruder's point of view, in the sense that the directions of transmitted messages are swapped (so $\mathsf{receive}$s become sends and vice-versa).

For the semantics of constraints we define a relation $\mathcal{I} \models^M_D \mathcal{A}$ where $\mathcal{A}$ is a constraint, M is the intruder knowledge, D is a set of pairs representing the current state of the databases, and $\mathcal{I}$ is an interpretation:

$$\begin{array}{lll}
\mathcal{I} \models^M_D 0 & \text{iff} & \mathit{true} \\
\mathcal{I} \models^M_D \mathsf{send}\ t_1, \dots, t_n \cdot \mathcal{A} & \text{iff} & M \vdash \mathcal{I}(t_i), \text{for all } i \in \{1, \dots, n\}, \text{and } \mathcal{I} \models^M_D \mathcal{A} \\
\mathcal{I} \models^M_D \mathsf{receive}\ t_1, \dots, t_n \cdot \mathcal{A} & \text{iff} & \mathcal{I} \models^{M \cup \{\mathcal{I}(t_1), \dots, \mathcal{I}(t_n)\}}_D \mathcal{A} \\
\mathcal{I} \models^M_D \mathsf{insert}\ t\ s \cdot \mathcal{A} & \text{iff} & \mathcal{I} \models^M_{D \cup \{\mathcal{I}((t,s))\}} \mathcal{A} \\
\mathcal{I} \models^M_D \mathsf{delete}\ t\ s \cdot \mathcal{A} & \text{iff} & \mathcal{I} \models^M_{D \setminus \{\mathcal{I}((t,s))\}} \mathcal{A} \\
\mathcal{I} \models^M_D t \not\doteq t' \cdot \mathcal{A} & \text{iff} & \mathcal{I}(t) \neq \mathcal{I}(t') \text{ and } \mathcal{I} \models^M_D \mathcal{A} \\
\mathcal{I} \models^M_D t\ \mathsf{in}\ s \cdot \mathcal{A} & \text{iff} & \mathcal{I}((t,s)) \in D \text{ and } \mathcal{I} \models^M_D \mathcal{A} \\
\mathcal{I} \models^M_D t\ \mathsf{notin}\ s \cdot \mathcal{A} & \text{iff} & \mathcal{I}((t,s)) \notin D \text{ and } \mathcal{I} \models^M_D \mathcal{A}
\end{array}$$

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 define what an intruder learns $\mathit{ik}(\mathcal{A})$ when a constraint $\mathcal{A}$ is executed: $\mathit{ik}(\mathcal{A}) = \{t_i | 1 \leq i \leq n, (\mathsf{receive}\ t_1, \dots, t_n) \in \mathcal{A}\}$. We also define how the databases look $db(\mathcal{A},\mathcal{I},D)$ after a constraint $\mathcal{A}$ is executed from a database D under interpretation $\mathcal{I}$:

$$\begin{aligned} db(0,\mathcal{I},D) &= D \\ db(\mathfrak{t}\cdot S,\mathcal{I},D) &= \begin{cases} db(S,\mathcal{I},D\cup\{(\mathcal{I}(t),\mathcal{I}(s))\}) & \text{if } \mathfrak{t} = \mathsf{insert}\ t\ s \\ db(S,\mathcal{I},D\setminus\{(\mathcal{I}(t),\mathcal{I}(s))\}) & \text{if } \mathfrak{t} = \mathsf{delete}\ t\ s \\ db(S,\mathcal{I},D) & \text{otherwise} \end{cases} \end{aligned}$$

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

$$(M,D,C) \Rightarrow_{\mathcal{P}} (M\cup\mathcal{I}(ik(\mathcal{A})), db(\mathcal{A},I,D), C\cup(subterms(\mathcal{A})\cap\mathcal{C}))$$

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 $fresh(T)$ to fresh values (i.e., $dom(\sigma) = fresh(T)$, $ran(\sigma)\subseteq\mathbb{V}$, and $ran(\sigma)\cap C=\emptyset$),
3. $\mathcal{A} = dual(\sigma(S_r\cdot S_c\cdot S_u\cdot S_s))$, and
4. $\mathcal{I}\models_D^M\mathcal{A}$.

A configuration (M,D,C) is said to be *ground reachable in* $\mathcal{P}$ iff $0 \Rightarrow_{\mathcal{P}}^{\star} (M,D,C)$ where $\Rightarrow_{\mathcal{P}}^{\star}$ denotes the transitive reflexive closure of $\Rightarrow_{\mathcal{P}}$. For any configuration (M,D,C) ground reachable in this transition system, M and D are ground because in each step the substitutions σ and $\mathcal{I}$ replace variables with ground terms in the elements added to these sets.

We now define a different semantics for protocols, namely one defined in terms of a symbolic transition system in which a single *constraint* is built up during transitions, essentially representing a "trace" of what has happened. We use this system as a basis for our formalization of both typing and compositionality because for these two aspects it is convenient to reason about the mentioned single constraint. For typing, it is convenient to reason about the many solutions it may have, and for compositionality it is convenient to split the constraint into parts that then constitute constraints of the individual protocols. We call the system symbolic because we allow the built constraint to contain variables—this is in contrast to the ground transition system which picks and applies a new interpretation $\mathcal{I}$ in each transition. The symbolic transition system is defined using 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. The transition

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

is 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 *symbolically reachable in* $\mathcal{P}$ iff $0 \Rightarrow^{\bullet\star}_{\mathcal{P}} \mathcal{A}$ where $\Rightarrow^{\bullet\star}_{\mathcal{P}}$ denotes the transitive reflexive closure of $\Rightarrow^{\bullet}_{\mathcal{P}}$. The protocol then has an *attack* iff there exists a symbolically reachable and satisfiable constraint where the intruder can produce the attack signal, i.e., there exists a symbolically 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*.

We show that the notions of reachability in the two systems correspond:

Theorem 1 [3]

$$\{(M, D) \mid (M, D, C) \text{ is ground reachable in } \mathcal{P}\} = \\ \{(\mathit{ik}(\mathcal{I}(\mathcal{A})), \mathit{db}(\mathcal{A}, \mathcal{I}, \emptyset)) \mid \mathcal{A} \text{ is symbolically reachable in } \mathcal{P} \text{ and } \mathcal{I} \models \mathcal{A}\}$$

For the remainder of the paper we will focus our attention on the symbolic transition system as justified by the above theorem and thus by *reachable* we will mean symbolically reachable.

3.3 Well-Formedness

We are going to employ the abstraction-based verification technique from [35] 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 be 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: $\mathit{fresh}(T) \cap (\mathit{fv}(S_r) \cup \mathit{fv}(S_c)) = \emptyset$

C3: $\mathit{fresh}(T) \subseteq \mathit{fv}(S_s) \cup \{x \mid \mathsf{insert}\ x\ s \in S_u\}$

[3]This theorem is called `protocol_model_equivalence` in the Isabelle formalization and can be found in the `Stateful_Protocol_Model.thy` theory file.

(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. As an easy workaround we define a special private unary function symbol occurs and then do a transformation. The transformation augments every rule containing action new x with the action send occurs(x), and also augments 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 using the following function:

Definition 1 *Let $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$ be a transaction, let $y_1 ... y_n$ be its fresh variables and let $\{x_1, ..., x_n\}$ be its (possibly empty) set of other free variables. We define a function add_occurs_sends that ensures that* occurs *messages are being sent:*

$$
\begin{aligned}
\mathit{add\_occurs\_sends}(0) &= \mathsf{send}(\mathsf{occurs}(y_1), ..., \mathsf{occurs}(y_n)) \\
\mathit{add\_occurs\_sends}(\mathsf{send}(t_1, ..., t_n) \cdot S) &= \mathsf{send}(\mathsf{occurs}(y_1), ..., \mathsf{occurs}(y_n), t_1, ..., t_n) \cdot S
\end{aligned}
$$

With this we define a function add_occurs_msgs that ensures that occurs *messages are being received and sent:*

$$\mathit{add\_occurs\_msgs}(T) = S'_r \cdot S_c \cdot F \cdot S_u \cdot S'_s$$

where

$$
\begin{aligned}
S'_r &= \textbf{if } \{x_1 ... x_n\} = \emptyset \textbf{ then } S_r \textbf{ else } \mathsf{receive}(\mathsf{occurs}(x_1), ..., \mathsf{occurs}(x_n)) \cdot S_r \\
S'_s &= \textbf{if } F = 0 \textbf{ then } S_s \textbf{ else } \mathit{add\_occurs\_sends}(S_s)
\end{aligned}
$$

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.

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.

If the user does not include an initial value producing transaction then our tool will automatically insert one. If the user does include one, then it is the design choice of the user to define exactly how it should look, as long as it lives up to the definition of being an initial value producing transaction. This is in our opinion more flexible than strictly enforcing a specific rule, since the user can adapt the rule 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:

```
intruderValues()
    new $PK$
    insert $PK$ intruderkeys
    send $PK$
    send inv($PK$).
```

Thus, we require (and automatically check) that each protocol specification includes a value-producing transaction:

Definition 2 *A transaction is an initial value-producing transaction for a protocol $\mathcal{P}$ if it is of the form* new $x \cdot S_u \cdot$ send $t_1 ... t_n$ *where $t_i = x$ for some i, no other variable than x occurs in a subterm in $t_1 ... t_n$ and where S_u is either* 0 *or* insert *x c for a set c such that no transaction in $\mathcal{P}$ deletes from nor does any check on c.*

It is clear that an initial value-producing transaction is applicable in every state and generates a fresh value.

Note that the occurs messages are only added during verification. We prove the transformation to be sound. Essential to this proof is the above realization that the intruder will only have occurs messages available for values (i.e., $\mathbb{V}$). This essentially means that if a run of $\mathcal{P}$ relies on running a transaction with its actual parameters being built from public constants (i.e. $\mathcal{C}_{pub}$), then in the transformed protocol the intruder cannot do the same, because he will not have in his knowledge the needed occurs messages for these. [4] In order to resolve this problem we prove the following lemma which transforms a run $\mathcal{A}$ of $\mathcal{P}$ relying possibly on public constants into one that relies only on values:

Lemma 1 *Let $\mathcal{P}$ be a protocol that includes a value-producing transaction and which has a well-typed attack $\mathcal{A} \cdot$* attack *with model $\mathcal{I}$. Then there exists strand $\mathcal{B}$ and interpretation $\mathcal{J}$ such that $\mathcal{B} \cdot$* attack *is a well-typed attack on $\mathcal{P}$ with model $\mathcal{J}$ and such that $\mathcal{J}$ maps all $\mathcal{B}$'s free variables to values.*

The proof is essentially by induction on how $\mathcal{A}$ was reached by $\Rightarrow^\bullet$. Thus, we have to consider an $\mathcal{A}$ being extended with a transaction $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$

[4]One could perhaps be tempted to simply remove the public constants from the development, however that is incompatible with the typing result of [26] which relies on an infinite set of these being available.

to $\mathcal{A} \cdot dual(\rho(\sigma(S_r \cdot S_c \cdot S_u \cdot S_s)))$ and then show that the corresponding $\mathcal{B}$ can be similarly be extended in a way that preserves the properties required by the lemma. The substitution σ picked some number n of public constants. In the extension of $\mathcal{B}$ we will apply first an initial value producing transaction n times to obtain n values, and then use T, but with a substitution σ' that uses these n values instead of the public constants. The formalized proof is tricky, as it requires us to keep track of which fresh values and variables have been used so far in the induction, and we also need to meticulously update our model $\mathcal{J}$ to ensure that it is indeed a model of $\mathcal{B}$. Therefore, in the formal proof, the induction is done in a central step where the property proved is strengthened to account for these aspects.

Theorem 2 [5] *Let $\mathcal{P}$ be a protocol that includes a value-producing transaction and which has a well-typed attack. Then the protocol $\{add\_occurs\_msgs\ T \mid T \in \mathcal{P}\}$ also has a well-typed attack.*

The proof has essentially three steps: The first step relies on lemma 1 by obtaining the attack $\mathcal{B}$ and model $\mathcal{J}$ described by that lemma's conclusion. The second step inserts in $\mathcal{B}$ the sending and receiving of appropriate occurs messages, thus turning it into an attack on $\{add\_occurs\_msgs\ T \mid T \in \mathcal{P}\}$. The third step proves that $\mathcal{J}$ is also a model of $\{add\_occurs\_msgs\ T \mid T \in \mathcal{P}\}$.

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 inserting x into a set and delete y from that very set. To steer clear of this, the paper [35] 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 $x \not\doteq y$ to the original rule. We do that until all rules have $x \not\doteq y$ 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 $PK \not\doteq NPK$ 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 [34, 12, 35]. Essentially the method computes a fixed point that over-approximates what can ever happen in any sequence of transactions.

[5]This theorem is called `add_occurs_msgs_soundness` in the Isabelle formalization and can be found in the `Stateful_Protocol_Verification.thy` theory file.

While it is relatively easy to formalize the computation of this fixed point in Isabelle, the main work consists in convincing Isabelle that every transaction is covered by the fixed point in the following sense. Given any trace that is represented by the fixed point and in which a transaction is executable, then also the resulting trace is covered by the fixed point. Thereby all traces are covered by the fixed point, and when the attack predicate is not contained in the fixed point, it is not reachable in any trace of the protocol. It is essential that the Isabelle proof does not rely on the correctness of the approach or the correct computation of the fixed point: Rather, the fixed point can be regarded as a mere proof idea, as a claimed upper bound on what can happen, and in the worst case, if this upper bound were wrong then the attempt to prove coverage of all transactions would fail. Thus, the coverage check is an approach to automatically "generate" a proof in Isabelle, and this is indeed the core contribution of this work.

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 $\models_D^M$, where here M denotes the intruder knowledge (all the messages received so far) and D denotes the state of the sets $\mathbb{S}$ (all values inserted into a set that were not deleted so far). We could thus characterize 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 trace is possible (after taking a transition of* outOfBand *with variables instantiated by* $[PK \mapsto \mathsf{pk}_1, U \mapsto \mathsf{a}]$ *followed by a transition of* keyUpdateUser *with variables instantiated by* $[PK \mapsto \mathsf{pk}_1, U \mapsto \mathsf{a}, NPK \mapsto \mathsf{pk}_2]$*):*

$$\begin{array}{l}
\mathsf{insert\ pk_1\ ring(a)} \\
\mathsf{insert\ pk_1\ valid(a)} \\
\mathsf{receive\ pk_1} \\
\hline
\mathsf{pk_1\ in\ \ ring(a)} \\
\mathsf{delete\ pk_1\ ring(a)} \\
\mathsf{insert\ pk_2\ ring(a)} \\
\mathsf{receive\ sign(inv(pk_1), pair(a, pk_2))}
\end{array}$$

Suppose we start in state $M_0 = \emptyset$ *and* $D_0 = \emptyset$*. After this trace we have*

$$\begin{array}{rcl}
M & = & \{\mathsf{pk_1}, \mathsf{sign(inv(pk_1), pair(a, pk_2))}\}, \text{ and} \\
D & = & \{(\mathsf{pk_1}, \mathsf{valid(a)}), (\mathsf{pk_2}, \mathsf{ring(a)})\}.
\end{array}$$

In general, D consists of pairs (v, s) where $v \in \mathbb{V}$ is a value and $s \in \mathbb{S}$ is a set. The idea of our abstract interpretation is that we stop distinguishing values that are members of the same sets. Let thus $\mathbb{A}$ be the powerset of $\mathbb{S}$ and define an abstraction function α_D from $\mathbb{V}$ to $\mathbb{A}$ that depends on the current state D:

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

and we extend it to terms and sets of terms as expected. Remember that $\mathbb{A}$ is included in Σ^0 so we can build *abstract terms* that include elements of $\mathbb{A}$ as *abstract constants*.

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

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 [34, 12, 35] 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 [26] 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 also limit 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 [23].) Thus, it is sound to limit the fixed point, including intruder deduction, to well-typed terms, which makes the fixed point finite.

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.

Example 5 *The value* pk_1 *in example 3 is created in the first transaction and has, after the first transaction the abstraction* $\{\mathsf{valid(a)}, \mathsf{ring(a)}\}$. *Since* *the second transaction deletes* pk_1 *from* $\mathsf{ring(a)}$*, it changes its abstract class to* $\{\mathsf{valid(a)}\}$*.*

As such transitions of abstract class play a crucial role in the approach, define the following notion:

Definition 3 (Term implication) *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. Suppose an abstract set of messages contains several occurrences of the same abstract value $a \in \mathbb{A}$, say $M = \{f(a), g(a, a)\}$. Due to the abstraction, we have lost the information of how many distinct constants are represented here, e.g., two corresponding concrete set of messages could be $M_0 = \{f(c_1), f(c_2), g(c_1, c_2), g(c_1, c_1)\}$ and $M_1 = \{f(c_2), g(c_2, c_2), g(c_2, c_1)\}$ where both c_1 and c_2 have the same set memberships a. If now value c_1 changes its set memberships to, say, $b \in \mathbb{A}$, then the abstraction of M_0 becomes $\{f(b), f(a), g(b, b), g(b, a)\}$ and the abtraction of M_1 becomes $\{f(a), g(a, a), g(a, b)\}$. Thus, in general, to include all possible terms that can be reached by a term implication $a \twoheadrightarrow b$, each occurrence of a can independently change to b. This means that all of the original terms with no a changed to b are also reached and hence we call it an implication. This is captured by the following definitions:

Definition 4 (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:*

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

Note that this relation is also reflexive since $a \;{}_a\!\twoheadrightarrow_b\; a$ *follows from* $a \in \mathbb{A} \subseteq \Sigma^0$ *and the third rule. 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 5 (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 6 (Fixed point) *A* protocol fixed-point candidate*, or* fixed point *for short,*[6] *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 [23] (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.

Our approach allows for an unbounded number of sessions. The only difference here between our work and, e.g., Tamarin [33] 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 [26], 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 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 insert $Nonce$ $\mathsf{privCh}(A, B)$.

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

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

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

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}:

$$\begin{tikzcd} \{\mathsf{ring(a)}\} \arrow[r, two heads] \arrow[dr, two heads] & \emptyset \arrow[r, two heads] & \{\mathsf{valid(a)}\} \arrow[r, two heads] & \{\mathsf{revoked(a)}\} \\ & \{\mathsf{ring(a)}, \mathsf{valid(a)}\} \arrow[ur, two heads] & & \end{tikzcd}$$

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

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

To compute this, we first consider the transaction $\mathsf{outOfBand}$ where a fresh key is inserted into both $\mathsf{ring(a)}$ and $\mathsf{valid(a)}$ and sent out. The abstraction of this key is thus the value $\{\mathsf{ring(a)}, \mathsf{valid(a)}\}$. This value is in the intruder knowledge in FP_{ks} but redundant due to other messages we derive later.[7] Note that this rule cannot produce anything else so we do not consider it for the remainder.

Next let us look at the transaction $\mathsf{keyUpdateUser}$. For $\mathsf{keyUpdateUser}$ we need to choose an abstract value for PK that satisfies the check PK in $\mathsf{ring(a)}$. 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 $\mathsf{ring(a)}$, 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 $\mathsf{ring(a)}$, 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 $\mathsf{keyUpdateUser}$ 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 $\mathsf{ring(a)}$.

[7] 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).

Applying the $\mathsf{keyUpdateServer}$ 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 $\mathsf{keyUpdateServer}$: 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 using 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.[8]

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 6, 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

[8]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.

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 7 (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. $cl_{TI}(FP) \vdash \delta(t_i)$ *for all* receive $t_1, \ldots, t_n$ *occurring in* S_r *and for all* $i \in \{1, \ldots, n\}$

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

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

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 8 (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:*

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

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

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 9 (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. $cl_{TI}(FP) \vdash \delta_u(t_i)$ for all $\mathsf{send}\ t_1, \ldots, t_n$ occurring in S_s and for all $i \in \{1, \ldots, n\}$

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 10 (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 3 [9] *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 6 *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 10.*

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

The only variables occurring in 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 7. We denote by Δ the set of those substitutions. Afterwards we show that all $\delta \in \Delta$ satisfy the post-conditions given in Definition 9.*

The variables PK and NPK are not declared as fresh in 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*

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

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

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

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

Now we can verify that conditions G1 and G2 hold for δ^1 and δ^2: We have that $\delta^i(x) \twoheadrightarrow \delta^i_u(x)$ 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^i_u. 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 11 gives a simple method to compute protocol fixed points based on this idea.

Definition 11 *Let $\mathcal{P}$ be a protocol and let f be the function defined as follows:*

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

where
$$\begin{aligned}\Delta^{T}_{FP,TI} &\equiv \{\delta \mid dom(\delta) = fv(T), \mathsf{pre}((FP, TI), T, \delta)\}\\ \widehat{FP}^{T}_{\delta} &\equiv \{\delta_u(t_i) \mid \mathsf{send}\ t_1, \ldots, t_i, \ldots, t_n \ \mathit{occurs\ in}\ T\}\\ \widehat{TI}^{T}_{\delta} &\equiv \{(\delta(x), \delta_u(x)) \mid x \in fv(T) \setminus \mathit{fresh}(T)\}\end{aligned}$$

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 do 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 also handle 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 5. 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:

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

This function indeed fulfills its purpose:

Lemma 2 *$cl_{TI}(M) \vdash_c t$ 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 it 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 12 (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 shorthand 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 3 *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 4 *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 7 *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.* ■

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 $cl_{TI}(M)$). The former corresponds to the "if"-branch of the following definition and the latter corresponds to final "else"-branch:

Lemma 5 *$cl_{TI}(M)$ 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 if** $\mathbb{A} \cap subterms(\mathsf{Keys}(t)) = \emptyset$ **then** true
**else if** $\forall s \in cl_{TI}(\mathsf{Keys}(t)).\ cl_{TI}(M) \nvdash_c s$ **then** true
**else** all $t' \in cl_{TI}(t)$ are analyzed in $cl_{TI}(M)$.
```

Lemma 5 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 5 fails. For instance, in Example 7 we need to extend $M = \{f(a), g(b)\}$ with b, resulting in the analyzed knowledge $M' = \{f(a), g(b), b\}$.

The two "else-if"-branches are an improvement we have made for this journal version of the paper. The idea is the following:

1. If $\mathsf{Keys}(t)$ are not derivable from the term-implication closed knowledge $cl_{TI}(M)$, and if there is no abstract value occurring in $\mathsf{Keys}(t)$, then $\mathsf{Keys}(s) = \mathsf{Keys}(t)$ for all implied terms s of t, and so t is analyzed in $cl_{TI}(M)$.

2. If none of the implied keys of t are derivable then t is also analyzed in $cl_{TI}(M)$.

These two special cases are useful to speed up the analyzed-fixed-point check when the fixed point contains terms that have lots of abstract values in them and that cannot be analyzed by the intruder (the last else-branch would in such cases take a lot of time to compute since the size of $cl_{TI}(t)$ grows exponentially with the number of occurrences of abstract values in t)—also, it is often the case that $\mathsf{Keys}(t)$ has fewer abstract values in it than $\mathsf{Result}(t)$, and so the size of $cl_{TI}(\mathsf{Keys}(t))$ is likely to be much smaller than $cl_{TI}(t)$, hence the second "else-if" condition is usually much faster to check than the last else-branch.

As an example, when modeling private channels one may use terms of the form $\mathsf{secch}(\mathsf{secchcr}(a, b), t)$, denoting that t is sent on a private channel from agent a to agent b, where the term t is derivable if the secret $\mathsf{secchcr}(a, b)$ is known, and where there would be an attack on the protocol if the intruder knew $\mathsf{secchcr}(a, b)$ for honest a and b. In a secure protocol the term $\mathsf{secch}(\mathsf{secchcr}(a, b), t)$, for honest a and b, would not be derivable by the intruder, and so it is sufficient to check that $cl_{TI}(M) \nvdash_c \mathsf{secchcr}(a, b)$ and $\mathbb{A} \cap \mathit{subterms}(\mathsf{secchcr}(a, b)) = \emptyset$ instead of checking that all elements of $cl_{TI}(\mathsf{secch}(\mathsf{secchcr}(a, b), t))$ are analyzed in $cl_{TI}(M)$, which may take a significant amount of time since t may contain a lot of abstract values.

6.3 A Further Improvement of the Coverage Check

We return to the second issue described in the beginning of this section: the issue of restricting how many abstraction substitutions need to be considered in order to conclude that a transaction is covered. The solution presented so far restricted the number of abstraction substitions considered by computing those that satisfy conditions F2 to F4 of Definition 7. We now show a way to take into consideration also condition F1 to further restrict the set of abstraction substitutions to consider. This will not be exactly the set of substitutions that satisfy F1 to F4, but rather do an over-approximation that allows us to also calculate the desired set of abstration substitutions in an efficient way.

F1 essentially says that for each received term t of a considered transaction, its abstraction $\delta(t)$ must be something that the intruder can actually produce from the fixed point. Thus, by inspecting t and comparing it to the terms available in the fixed point we will be able to see what the variables in t could be instantiated to if $\delta(t)$ were to be a term producible from the fixed point.

The following functions capture this idea:

Definition 13 *Let T be a transaction and let $t_1, \ldots, t_n$ be the terms in the* $\mathsf{receive}$*-steps of T. Assume that $n > 0$, i.e., that T has at least one* $\mathsf{receive}$*-step. Let $\mathsf{FP} = (FP, TI)$ be a fixed point and let OCC be the abstract values that occur in FP. Assume that FP is analyzed. Let furthermore x be a free variable of T. Then an over-approximation of the possible abstractions of x constrained by $t_1, \ldots, t_n$ is the set $\mathsf{rcvconstrs}_x(\{t_1, \ldots, t_n\})$ where $\mathsf{rcvconstrs}_x$ is defined as follows:*

$$\begin{array}{rcll}
\mathsf{rcvconstrs}_x(\{t_1,\ldots,t_n\}) & = & \bigcap_{i\in\{1,\ldots,n\}} \mathsf{rcvconstrs}_x(\{t_i\}) & \\
\mathsf{rcvconstrs}_x(\{y\}) & = & \{b \mid a \in FP \cap \mathbb{A}, (a,b) \in TI^*\} & \textit{if } x = y \\
\mathsf{rcvconstrs}_x(\{y\}) & = & OCC & \textit{if } x \neq y \\
\mathsf{rcvconstrs}_x(\{c\}) & = & \emptyset & \textit{if } c \in \mathcal{C}_{priv} \textit{ and } c \notin FP \\
\mathsf{rcvconstrs}_x(\{c\}) & = & OCC & \textit{if } c \in \mathcal{C}_{pub} \textit{ or } c \in FP \cap \mathcal{C}_{priv} \\
\mathsf{rcvconstrs}_x(\{f(t_1,\ldots,t_n)\}) & = & \theta_1 \cup \theta_2 & \textit{if } f \in \Sigma^n, n > 0, \\
 & & & \theta_1 = \bigcup_{\delta\in\Delta} \delta(x), \\
 & & & \theta_2 = \mathsf{rcvconstrs}_x(\{t_1,\ldots,t_n\}), \\
 & & & \Delta = \mathsf{match}(f(t_1,\ldots,t_n), FP)
\end{array}$$

and where match *is defined as follows:*

$$\begin{array}{rcll}
\mathsf{match}(t, M) & = & \{\delta \in \mathsf{match}(t,s) \mid s \in M\} & \\
\mathsf{match}(t, s) & = & \{\theta\} & \textit{if } \delta \in \mathsf{match}'(t,s), \\
 & & & \forall x.\ \delta'(x) = \bigcap_{a\in\delta(x)} \{b \mid (a,b) \in TI^*\}, \\
 & & & \forall x \in \mathit{fv}(t).\ \delta'(x) \neq \emptyset, \\
 & & & \forall x.\ \theta(x) = \textbf{if } x \in \mathit{fv}(t) \textbf{ then } \delta'(x) \textbf{ else } OCC \\
\mathsf{match}(t, s) & = & \emptyset & \textit{otherwise} \\
\mathsf{match}'(x, a) & = & \{\theta\} & \textit{if } x \in \mathcal{V}, a \in \mathbb{A}, \forall y.\ \theta(y) = \textbf{if } x = y \textbf{ then } \{a\} \textbf{ else } \emptyset \\
\mathsf{match}'(t, s) & = & \{\theta\} & \textit{if } t = f(t_1,\ldots,t_n), s = f(s_1,\ldots,s_n), \\
 & & & \forall i \in \{1,\ldots,n\}.\ \mathsf{match}'(t_i,s_i) \neq \emptyset, \\
 & & & \Delta = \{\delta \in \mathsf{match}'(t_i,s_i) \mid i \in \{1,\ldots,n\}\}, \\
 & & & \forall x.\ \theta(x) = \bigcup_{\delta\in\Delta} \delta(x) \\
\mathsf{match}'(t, s) & = & \emptyset & \textit{otherwise}
\end{array}$$

We explain now what the above functions will do. Consider first the match' function from a syntactical point of view. This function takes as input a received term t and tries to match it with a term s from the intruder knowledge. It will firstly check if there is a way to replace each variable *occurrence* in t with an abstract value such that t becomes syntactically equal to s. If there is no such replacement, then it will simply give $\emptyset$. However, if there is such a replacement match' will return a singleton set consisting of a map θ from variables into sets of abstract values. Each variable is mapped to the set of abstract values that have the property that some occurrence of the variable needs to be replaced with that abstract value in order for t to become syntactically equal to s.

Example 8 *As an example, we have that* $\mathsf{match}'(\mathsf{f}(X, X), \mathsf{f}(\{\mathsf{valid(a)}, \mathsf{ring(a)}\}, \{\mathsf{revoked(a)}, \mathsf{ring(a)}\}) = \{[X \mapsto \{\{\mathsf{valid(a)}, \mathsf{ring(a)}\}, \{\mathsf{revoked(a)}, \mathsf{ring(a)}\}\}]\}$. *It is perhaps surprising that* X *is mapped to a set of values and not as in more simple matching algorithms to a single value. The reason is that an abstract value in a term in FP does not only represent itself, but its whole closure under the term implication graph. Thus,* $\{\mathsf{valid(a)}, \mathsf{ring(a)}\}$ *and* $\{\mathsf{revoked(a)}, \mathsf{ring(a)}\}$ *might actually be able to represent the same value, or more precisely, there might be a set of abstract values that* $\{\mathsf{valid(a)}, \mathsf{ring(a)}\}$ *and* $\{\mathsf{revoked(a)}, \mathsf{ring(a)}\}$ *both represent under term implication.*

The example also points to a more semantic way to understand match'. Say that we are interested in the set of abstraction substitutions that when applied to t gives a term represented by the term implication closure of s. For each free variable x in t, such an abstraction substitution must necessarily return an abstract value that is part of the term implication closures of all abstract values in $\theta(x)$ for the $\theta \in \mathsf{match}'(t, s)$. Consider now the match function which captures exactly this semantic idea. When possible, it returns a singleton containing a map θ that expresses our idea more directly: The match function ensures that for each variable x, the abstraction substitutions we were interested in just above will necessarily return an abstract value in $\theta(x)$. Additionally, match can also be applied to a received term t and a whole intruder knowledge M. Consider here any abstraction substitution, that when applied to t gives a term represented by the term implication closure of some s in M. For such an abstraction substitution there must be a $\theta \in \mathsf{match}(t, M)$ such that for each variable x, the abstraction substitution returns an abstract value in $\theta(x)$. Let us consider an example:

Example 9 *Assume that the term implication graph is* $\{\mathsf{valid(a)}, \mathsf{ring(a)}\} \twoheadrightarrow \{\mathsf{revoked(a)}, \mathsf{ring(a)}\}$. *Consider then the following calculation by the* match *function:*

$$\begin{aligned}
&\mathsf{match}(\mathsf{f}(X, X), \mathsf{f}(\{\mathsf{valid(a)}, \mathsf{ring(a)}\}, \{\mathsf{revoked(a)}, \mathsf{ring(a)}\}))\\
&= \{[X \mapsto \{\{\mathsf{valid(a)}, \mathsf{ring(a)}\}, \{\mathsf{revoked(a)}, \mathsf{ring(a)}\}\} \cap \{\{\mathsf{revoked(a)}, \mathsf{ring(a)}\}\}]\}\\
&= \{[X \mapsto \{\{\mathsf{revoked(a)}, \mathsf{ring(a)}\}\}]\}
\end{aligned}$$

Notice here that indeed the term $\mathsf{f}(\{\mathsf{revoked(a)}, \mathsf{ring(a)}\}, \{\mathsf{revoked(a)}, \mathsf{ring(a)}\})$ *is represented by the term* $f(\{\mathsf{valid(a)}, \mathsf{ring(a)}\}, \{\mathsf{revoked(a)}, \mathsf{ring(a)}\})$ *when considering the term implication graph.*

Consider lastly the $\mathsf{rcvconstrs}$ function. This can take a variable x and singleton set of a received term t. It then essentially checks if t actually is a term that the intruder could potentially find in or build from the fixed point, and if that is the case then it collects a set of abstract values which x could have for that to happen. A simple case is when t is exactly the variable x because here we simply collect all abstract values that are directly in the fixed point when closed under the term implication graph. The most interesting case is when t is a composite term $f(t_1, \ldots, t_n)$. Here, we have essentially to consider two cases:

- the case that the intruder can directly "match" $f(t_1, \ldots, t_n)$ with a term in FP
- the case that the intruder can construct $f(t_1, \ldots, t_n)$ by composition from the intruder knowledge, thus having to recursively consider for $t_1, \ldots, t_n$ the two cases we are considering here.

We need not consider analysis, because we are assuming that FP is analyzed. The matching done in the former case is then where possible values for x is found. We also define $\mathsf{rcvconstrs}$ when applied to a non-singleton set M of received terms, representing a set of received terms. The variable x of course has to live up to the constraints generated by all the received terms relative to the fixed point, and thus we can simply take the intersection of constraints generated by the individual received terms. The following example illustrates the idea:

Example 10 *Consider a transaction receiving the set of terms* $\{\mathsf{c_{pub}}, X, \mathsf{f_{priv2}}(X, Y), \mathsf{f_{pub2}}(\mathsf{f_{priv1}}(X), Y)\}$ *where* $\mathsf{c_{pub}}$ *is a public constant,* $\mathsf{f_{priv1}}$ *and* $\mathsf{f_{priv2}}$ *are private functions and* $\mathsf{f_{pub2}}$ *is a public function.*

Assume that we have the following fixed point:

$$\begin{aligned} FP &= \{ \quad \mathsf{f_{priv2}}(\{\mathsf{valid(a)}\}, \{\mathsf{valid(a)}\}), \mathsf{f_{priv2}}(\{\mathsf{revoked(a)}\}, \{\mathsf{valid(a)}\}), \\ &\qquad \{\mathsf{revoked(a)}\}, \mathsf{f_{priv1}}(\{\mathsf{revoked(a)}\})\} \\ TI &= \emptyset \end{aligned}$$

Assume also for the sake of simplicity that there are no analysis rules. Then this fixed point is clearly analyzed because all messages that the intruder can derive from FP *using* $\vdash$ *can also be derived using* $\Vdash_c$. *Let us now apply* $\mathsf{rcvconstrs}$ *to get* $\mathsf{rcvconstrs}_X(\{\mathsf{c_{pub}}, X, \mathsf{f_{priv2}}(X, Y), \mathsf{f_{pub2}}(\mathsf{f_{priv1}}(X), Y)\}) = \{\{\mathsf{revoked(a)}\}\}$ *and* $\mathsf{rcvconstrs}_Y(\{\mathsf{c_{pub}}, X, \mathsf{f_{priv2}}(X, Y), \mathsf{f_{pub2}}(\mathsf{f_{priv1}}(X), Y)\}) = \{\{\mathsf{valid(a)}\}\}$. *Thus, in this case we get one possible abstraction substitution, namely* $[X \mapsto \{\mathsf{revoked(a)}\}, Y \mapsto \{\mathsf{valid(a)}\}]$.

Example 11 *Consider a transaction receiving the set of terms* $\{\mathsf{f_{priv2}}(X, Y)\}$ *where* $\mathsf{f_{priv2}}$ *is a private function. Consider the fixed point* $\mathsf{FP} = (\{\mathsf{f_{priv2}}(\{\mathsf{valid(a)}\}, \{\mathsf{revoked(a)}\}), \mathsf{f_{priv2}}(\{\mathsf{revoked(a}$
Assume also for the sake of simplicity that there are no analysis rules. Let us now apply $\mathsf{rcvconstrs}$ *to get* $\mathsf{rcvconstrs}_X(\{\mathsf{f_{priv2}}(X, Y)\}) = \{\{\mathsf{revoked(a)}\}, \{\mathsf{valid(a)}\}\}$ *and* $\mathsf{rcvconstrs}_Y(\{\mathsf{f_{priv2}}(X, Y)\}) = \{\{\mathsf{revoked(a)}\}, \{\mathsf{valid(a)}\}\}$. *Thus, in this case we get four possible abstraction substitution, namely* $[X \mapsto \{\mathsf{revoked(a)}\}, Y \mapsto \{\mathsf{valid(a)}\}]$, $[X \mapsto \{\mathsf{valid(a)}\}, Y \mapsto \{\mathsf{revoked(a)}\}]$, $[X \mapsto \{\mathsf{revoked(a)}\}, Y \mapsto \{\mathsf{revoked(a)}\}]$ *and* $[X \mapsto \{\mathsf{valid(a)}\}, Y \mapsto \{\mathsf{valid(a)}\}]$. *Notice that we see here that an overapproximation is happening because actually only the first two abstraction substitutions will generate terms represented by the fixed point.*

We explain also how the mentioned functions are implemented. The way that match' does the mentioned checks and calculates its θ is simply by recursion on the structure of t. In the recursive case each subterm of t will create a mapping, and these mappings are combined by pointwise union. The match function calculates its θ as follows: It uses match' to check if t can potentially match s and the resulting map is δ. For each variable x the match function calculates which abstract values $\delta(x)$ represents by taking the term implication closure of each $a \in \delta(x)$ and in order to find the abstract values that all these closures represent, their intersection is taken. This results in the mapping δ' which maps each variable x to the mentioned intersection. Then match checks that all free variables in t can actually represent some abstract value, because if not then no abstraction substitution can make t be represented by s. Lastly the resulting mapping θ is simply δ', but modified to return OCC for all variables not free in t, because for such variables the intruder may pick any abstract value, which the intruder might do to also make some other pair of terms match, but we know here at least that the intruder will not be able to use any other value than the ones that occur in FP. The function match can also be applied to a received term t and the full intruder knowledge M. This is done by applying match individually to all terms in M and collecting the resulting maps in a set. The way that $\mathsf{rcvconstrs}$ works when applied to a variable x and a singleton set of a term is to do a recursion on the term. In case the term is a different variable than x, a public constant or an intruder-known private constant then this is not constraining what x could be, and thus the whole OCC is returned because we then know only that the intruder at least must pick a variable occurring in FP. In case t is an intruder-*un*known private constant then the intruder has no way to give x an abstract value, and thus $\emptyset$ is returned. This also means that no abstraction substitution will make the transaction applicable. If t is equal to x then the intruder can pick any intruder-known abstract value for x. If t is a composite term $f(t_1, \ldots, t_n)$ then, as mentioned earlier, we need to take into account both of the possibilities that the intruder can directly match $f(t_1, \ldots, t_n)$ with a term in FP and that the intruder can construct $f(t_1, \ldots, t_n)$ by composition. In the former case we can use match to calculate what possible values x can have when t is matched with the intruder knowledge. This is in the definition collected as the set θ_1. In the latter case we can use recursion to see what possible abstract values x can get if it occurs in one or more of $t_1, \ldots, t_n$. The result is the set θ_2, and we finally return $\theta_1 \cup \theta_2$.

The following lemma captures the idea of match:

Lemma 6 *If $\theta(t) \in cl_{TI}(s)$ and $fv(s) = \emptyset$, and there are no abstract values occurring in t, and θ is an abstraction substitution, then $\mathsf{match}(t, s) \neq \emptyset$.*

This lemma essentially shows that if a term t (which is intended to be from a recive step) can be instantiated by some abstraction substitution θ to be in the term implication closure of s (which is intended to be in FP), then $\mathsf{match}(t, s)$ will return a non-emtpy set of functions.

Next, we have lemma capturing the idea of $\mathsf{rcvconstrs}$:

Lemma 7 *Let T be a transaction with at least one* receive*-step,* $\mathsf{FP} = (FP, TI)$ *be a fixed point where FP is analyzed, $x \in fv(T)$, δ be an abstraction substitution with domain $fv(T)$, and let $t_1, \ldots, t_n$ be the terms occurring in the* receive*-steps of T. If $FP \vdash_c \delta(\{t_1, \ldots, t_n\})$ then $\delta(x) \in \mathsf{rcvconstrs}_x(\{t_1, \ldots, t_n\})$.*

						Verification					
	Initialization		Fixed Point			Safe Check		NBE Check		Unsafe Check	
Protocol	Translation	Setup	Computation	|FP|	|TI|	Default	Receive	Default	Receive	Default	Receive
Keyserver_2_1	00:00:07	00:00:46	00:00:04	22	27	00:01:05	00:01:45	00:00:24	00:00:25	00:00:19	00:00:24
Keyserver_3_1	00:00:08	00:00:36	00:00:04	31	40	00:00:44	00:00:53	00:00:24	00:00:23	00:00:22	00:00:21
Keyserver_4_1	00:00:09	00:00:38	00:00:04	40	53	00:00:59	00:01:21	00:00:22	00:00:20	00:00:18	00:00:17
Keyserver2_2_1	00:00:08	00:00:35	00:00:04	9	4	00:01:22	00:02:13	00:00:20	00:00:22	00:00:20	00:00:18
Keyserver2_3_1	00:00:06	00:00:46	00:00:04	12	6	00:01:57	00:04:27	00:00:26	00:00:30	00:00:19	00:00:20
Keyserver2_4_1	00:00:07	00:00:37	00:00:04	15	8	00:04:02	00:10:49	00:00:35	00:00:42	00:00:19	00:00:18
KS_Comp_2_1	00:00:09	00:00:38	00:00:06	40	105	00:23:51	00:57:32	00:02:00	00:02:37	00:00:18	00:00:21
KS_Comp_3_1	00:00:09	00:00:36	00:00:08	56	153	01:36:50	03:55:23	00:06:30	00:08:47	00:00:20	00:00:23
KS_Comp_4_1	00:00:09	00:00:37	00:00:13	70	201	04:57:05	11:38:15	00:15:57	00:24:45	00:00:18	00:00:25
NSLclassic	00:00:06	00:00:35	00:00:04	69	6	00:05:26	00:05:49	00:00:25	00:00:26	00:00:19	00:00:22
NSPKclassic	00:00:06	00:00:35	00:00:04	43	6	attack	attack	attack	attack	attack	attack
PKCS_Model03	00:00:06	00:00:37	00:00:07	8	2	attack	attack	attack	attack	attack	attack
PKCS_Model07	00:00:10	00:00:34	00:00:29	15	5	attack	attack	attack	attack	attack	attack
PKCS_Model09	00:00:07	00:00:36	00:00:13	40	20	attack	attack	attack	attack	attack	attack
TLS12_auth_simp	00:00:09	00:00:38	00:00:07	48	20	11:59:01	04:21:27	08:22:19	00:02:47	00:00:27	00:00:19

Table 1: Runtime Measurements (Time Format: *hh:mm:ss*).

This lemma essentially says that if there is an abstraction substitution that will instantiate a set of received terms to terms that the intruder can actually derive, then the application of that abstraction substitution to any variable will indeed be one of the abstract values that $\mathsf{rcvconstrs}_x$ calculates.

This section demonstrates how the Isabelle formalization allows for describing different strategies in order to prove to Isabelle that the computed fixed point covers the transactions. Indeed, as part of such arguments we show in Isabelle that the strategy is sound, like in Lemma 7. The strategies can often negotiate an efficient solution between different extremes. For instance, not considering the messages the intruder needs to produce (F1) leads to an unnecessarily large set of abstractions to consider, while computing the precise set of abstractions that satisfy (F1) would often waste a lot of time on an optimization that is just not worth it (or it may even be undecidable). While we here used our intuition and experience with examples, in general an extensive study of different variants on a larger benchmark suite could allow for further improvements.

7 Experimental Results

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 six columns, we report the run-time of three different strategies for the security proof: *Safe* employs symbolic evaluation using Isabelle's simplifier *code-simp*. For each of these strategies, we report the time our default version of the coverage check (*Default*) and the time for the improved coverage check (*Receive*). For the latter, please recall section 6.

In the *safe* 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. Ultimately, it is up to the user to decide which approach to use, preferably after consulting [21], 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 shared Linux server with an Intel Xeon E5-2640 CPU and 96GB 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. 20 minutes on the same machine.

The example Keyserver_h_d is our running keyserver example for h honest agents and d dishonest agents.[10] The example Keyserver_Composition_h_d with h honest agents and d dishonest agents is inspired by [23] 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 two honest agents 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 [35].

Finally, scenario 3 and 7 (PKCS#11_3 and PKCS#11_7), from the "PKCS#11" model that is distributed with AIF-ω [35] 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

[10]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.

	Initialization		Fixed Point (TLS)			Fixed Point (SSO)			Verification				
									TLS		SSO		
Check	Trans.	Setup	Comp.	|FP|	|TI|	Comp.	|FP|	|TI|	Default	Receive	Default	Receive	Composition
Safe Check	00:00:13	00:00:24	00:00:04	25	16	00:07:09	202	38	00:10:56	00:10:06	--:--:--	--:--:--	--:--:--
NBE Check	00:00:13	00:00:24	00:00:04	25	16	00:07:09	202	38	00:00:58	00:00:43	07:28:57	01:48:07	00:00:12
Unsafe Check	00:00:13	00:00:24	00:00:04	25	16	00:07:09	202	38	00:00:15	00:00:14	00:00:10	00:00:07	00:00:05

Table 2: Runtime Measurements (Time Format: *hh:mm:ss*) for the Composed Protocol TLS12_SSO.

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.

For all our examples, verification times for all examples, using the unsafe check (i.e., making full use of Isabelle's code generator), are below 30s. This makes this configuration ideal for interactive development, e.g., while refining a protocol specification. In contrast, the verification using only Isabelle's simplifier can take more than 12 hours for our example protocols. Thus, in most cases this configuration will be used in "batch-mode" after the protocol has been checked using the configuration employing the code generator. For the most protocols, NBE provides a good middle-ground, bringing the verification times down to under a minute for most examples, and below 30min for all examples except for TLS_auth_simp.

Furthermore, the improved coverage check introduced in section 6 significantly reduces the safe verification time for TLS_auth_simp in safe mode from 12 hours to less than 4 1/2 hours. For NBE, it reduces the runtime for TLS_auth_simp from over 8 hours to less than 3 minutes. For the configuration fully relying on code-generation (unsafe), the improvements are minor (from 27sec to 19sec). Note that this coverage check can also increase the verification times (e.g., for the composed keyserver examples). Further work is required to develop a heuristic helping users to decide, which check to try first. At this point in time, we recommend users of PSPSP to use the unsafe checking during interactive verification sessions and switch to safe using the regular coverage check for the final (batch-mode) verification. If this batch-mode check takes very long (e.g., more than a few hours), it is worthwhile to check if the improved coverage check (Default) is faster.

Table 2 shows the running times for an example of a protocol composition, implementing a TLS-based single sign-on. Note that the format of the table differs from Table 1: we report the running times for the three different verification checks (Safe, NBE, and Unsafe) on three separate lines. For each of these checks, we report the running time for the initialization and the fixed point computation (as well as its size) for each sub-protocol (i.e., TLS and SSO).

These aspects are the same for all three verification checks, as their implementation does not make use of the optimizations provided by the code generator. Next, we report the running times (as in Table 1 for both coverage checks) for individual verification of the two sub-protocols. The final column reports the running time for the proof that the composition of the two sub-protocols is secure. Here, "`--:--:--`" denotes that the check takes more than 12 hours. Notably, the security verification of the individual protocols takes significantly longer (for instance, several hours for SSO for the NBE check, the safe check does take more than 12 hours), while the proof that the composition of the two protocol is secure, only takes a few seconds. Furthermore, note TLS used in this composition case study focuses only on the core of TLS: the key exchange. In contrast, TLS_auth_simp in Table 1 also includes the password authentication and the transmission of some dummy data. This explains the difference in running times (and fixed-point sizes) between these two versions of TLS.

8 Isabelle/PSPSP

We implemented our approach on top of the Isabelle framework [40], resulting in a tool called Isabelle/PSPSP [29], which is now part of the Archive of Formal Proofs (AFP).[11] 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.

8.1 The Architecture of Isabelle/PSPSP

For our implementation of Isabelle/PSPSP, we make use of the fact that Isabelle is not only an interactive theorem prover; it also provides an extensible framework for developing formal method tools [43].

Figure 1 shows an overview of the Isabelle architecture, highlighting in green the additions provided by Isabelle/PSPSP. In particular:

- *Protocol Formalization:* PSPSP is built on, and re-uses, our stateful protocol formalization (and its typing results) formalized in Isabelle/HOL. This part is available as a stand-alone AFP entry [28], consisting of ca. $20,000$ lines of code. The formalization presented in this paper, formalizing the presented method for the automated verification of security protocols, adds another $25,000$ lines of code [29]. Note that these formalizations (proofs, definitions) are reusable, i.e., independent of any concrete security protocol.

- *Automated Proof Support (PSPSP Methods):* We developed several proof methods using, both, Isabelle's high-level proof development language Eisbach [30] and the Isabelle/ML interface. Isabelle/ML is Isabelle's pro-

[11]As part of the AFP, PSPSP will be maintained and, for instance, ported to the latest official release of Isabelle.

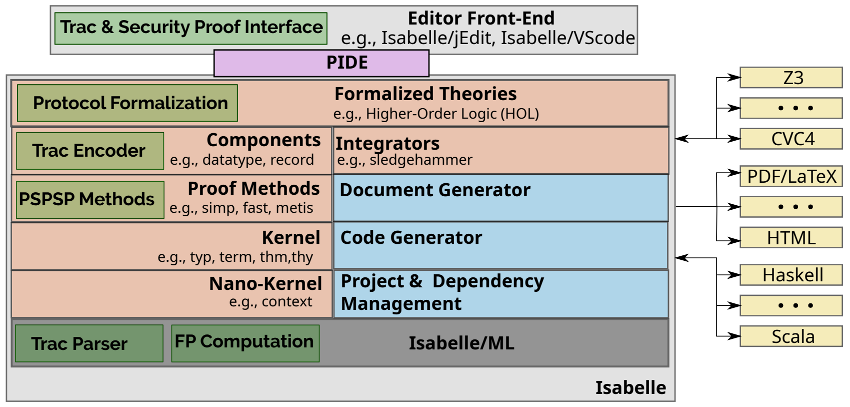


Figure 1: The system architecture of Isabelle and Isabelle/PSPSP.

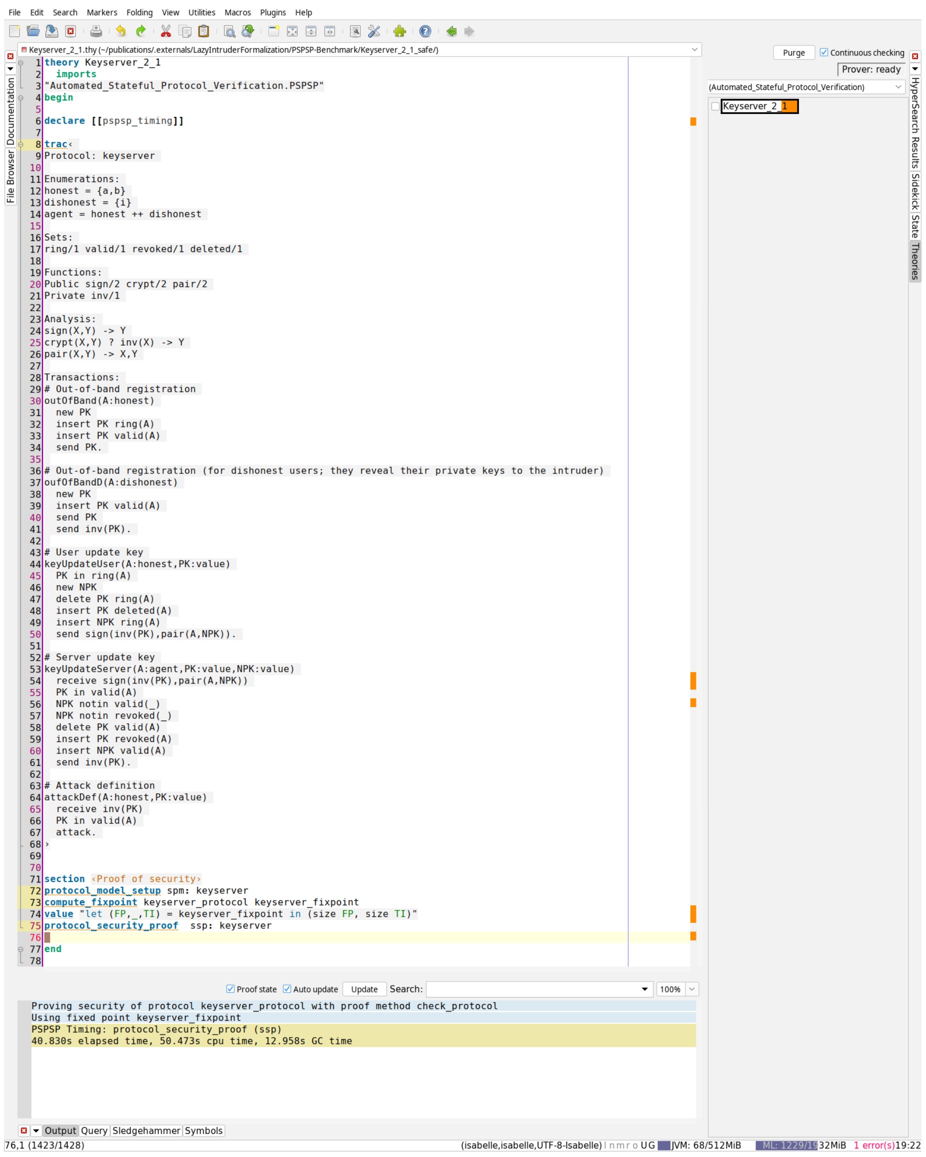
gramming API that allows one to extend Isabelle using the SML [37] programming language. We use this, in particular, for computing the abstraction of the fixed point that builds the back-bone of our automation.

- *Support for* `trac`*:* To improve the user-friendliness of PSPSP, we defined a trace-based specification language for security protocols, called `trac`. By supporting `trac` as input language, we allow users to use PSPSP without the need to understand all the details of our protocol formalization. Actually, users of PSPSP mostly need to understand `trac`, and our new Isabelle commands for verifying security protocols. Supporting `trac` requires a parser for `trac` (implemented in Isabelle/ML) and implementing an encoder (or datatype package) that translates `trac` into the corresponding HOL definitions. Furthermore, the `trac` datatype package also proves automatically a number of basic properties that are used within the actual security proof.

It is noteworthy all our additions have been implemented in a logically safe way, i.e., a bug in our implementation cannot result in an insecure protocol being successfully verified: any bug could only result in PSPSP not able to verify a secure protocol.

8.2 Isabelle/PSPSP – A Guided Tour

Figure 2 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 2 shows the specification, and the fully-automated verification of a toy keyserver protocol:


File Edit Search Markers Folding View Utilities Macros Plugins Help
Keyserver_2_1.thy (~/publications/.externals/LazyIntruderFormalization/PSPSP-Benchmark/Keyserver_2_1_safe/)
Purge
Continuous checking
Prover: ready
(Automated_Stateful_Protocol_Verification)
Keyserver_2_1
File Browser
Documentation
HyperSearch Results
Sidekick
State
Theories
Proof state
Auto update
Update
Search:
100%
Output
Query
Sledgehammer
Symbols

```
theory Keyserver_2_1
  imports
"Automated_Stateful_Protocol_Verification.PSPSP"
begin

declare [[pspsp_timing]]

trac‹
Protocol: keyserver

Enumerations:
honest = {a,b}
dishonest = {i}
agent = honest ++ dishonest

Sets:
ring/1 valid/1 revoked/1 deleted/1

Functions:
Public sign/2 crypt/2 pair/2
Private inv/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
pair(X,Y) -> X,Y

Transactions:
# Out-of-band registration
outOfBand(A:honest)
  new PK
  insert PK ring(A)
  insert PK valid(A)
  send PK.

# Out-of-band registration (for dishonest users; they reveal their private keys to the intruder)
oufOfBandD(A:dishonest)
  new PK
  insert PK valid(A)
  send PK
  send inv(PK).

# User update key
keyUpdateUser(A:honest,PK:value)
  PK in ring(A)
  new NPK
  delete PK ring(A)
  insert PK deleted(A)
  insert NPK ring(A)
  send sign(inv(PK),pair(A,NPK)).

# Server update key
keyUpdateServer(A:agent,PK:value,NPK:value)
  receive sign(inv(PK),pair(A,NPK))
  PK in valid(A)
  NPK notin valid(_)
  NPK notin revoked(_)
  delete PK valid(A)
  insert PK revoked(A)
  insert NPK valid(A)
  send inv(PK).

# Attack definition
attackDef(A:honest,PK:value)
  receive inv(PK)
  PK in valid(A)
  attack.
›


section ‹Proof of security›
protocol_model_setup spm: keyserver
compute_fixpoint keyserver_protocol keyserver_fixpoint
value "let (FP,_,TI) = keyserver_fixpoint in (size FP, size TI)"
protocol_security_proof  ssp: keyserver

end
```

```
Proving security of protocol keyserver_protocol with proof method check_protocol
Using fixed point keyserver_fixpoint
PSPSP Timing: protocol_security_proof (ssp)
40.830s elapsed time, 50.473s cpu time, 12.958s GC time
```

```
76,1 (1423/1428)                    (isabelle,isabelle,UTF-8-Isabelle) I n m r o U G  JVM: 68/512MiB  ML: 1229/1932MiB  1 error(s) 19:22
```

Figure 2: Using Isabelle/PSPSP for verifying a toy keyserver protocol.

- The protocol is specified using the domain-specific language trac that, e.g., could also be used by a security protocol model checker (line 9–67). 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 over 350 theorems are automatically generated.

- Next (line 72) 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 73, we compute the fixed point. We can use Isabelle's value-command (line 74) to inspect its size.

After these steps, all definitions and auxiliary lemmas for the security proof are available. We can now perform a fully automated proof (line 75). This top-level command proves automatically a theorem showing the security of the defined protocol. This successful proof took ca. 41s (see lower part of the Isabelle/jEdit window.)

8.3 Compositionality

PSPSP is part of a larger Isabelle infrastructure for security protocols that allows also for compositionality [24], i.e., for a result of a form: if two or more protocols are secure in isolation and satisfy certain requirements, then also their composition is secure, i.e., when they run in parallel sharing the same network and even some sets. Especially the support for shared sets allows us to consider also complex interactions between two protocols, for instance where one protocol negotiates keys and another protocol uses them.

The compositionality framework uses the same specification language (trac) as PSPSP. One can thus specify a set of component protocols, use PSPSP to prove the security of each of them in isolation, and use the compositionality framework to check that they fulfill the requirements for compositionality and then obtain an Isabelle proof that the composed system is secure.

As an example, we have modeled in [24] a composition of TLS 1.2 and SAML Single-Sign-On (SSO): TLS establishes a shared key where the client is not authenticated, but can do so with the help of an identity provider, using a password between client and identity provider. The TLS protocol would store any exchanged keys that a client A has negotiated with server B in a set $\mathsf{clientKeys}(A, B)$ on the client side, and in the set $\mathsf{serverKeys}(B)$ on the server side. The latter set of keys is only parameterized over the agent name B, since A is not authenticated. Each of these sets are, of course, from the local point of view of each agent, and it is part of the verification that, for instance, the intruder does not find out a key between two honest agents A and B. Finally, the SSO protocol can just retrieve and use these keys, both to authenticate the connection between client and server, as well as between client and identity provider (where the password is transmitted). Note that the run time for the TLS component of the composition is here greater than the runtime of the TLS_auth_simp, because it is purely the key-exchange while all authentication and data-transmission is “outsourced” to the SSO protocol. Thus, in general, verification can be improved by using compositional reasoning, if one can split a complex system into smaller components.

The compositionality framework supports strictly more protocols than PSPSP, most importantly it allows for composed messages in sets. In these cases, one cannot use PSPSP to verify the respective components, but of course one can also consider compositions where a subset of the components is proved by PSPSP and the others are verified manually.

9 Case Study: Logos

We had the opportunity to formally verify with PSPSP a protocol by the Danish company Logos. This protocol is in the area of reader terminals for a travel card solution, namely to establish a secure connection between a terminal and the Logos server. The particular challenge here is that this should work after a terminal has been in storage for years and all public keys of the servers have been updated in the meantime; we do *not* want to exclude that an intruder could have obtained old private keys—after all that is the reason for updating them regularly. Such a protocol, even though its messages are fairly simple, is obviously a challenge for verification tools (without bounding the number of sessions) as it requires mutable long-term states at its core. With some simplifications, we have managed to make a model in PSPSP and found a security flaw, and then verified the protocol with PSPSP under a minor modification. Logos has applied this modification and has thus an Isabelle-verified product, one might say, although this should of course be taken with a grain of salt, given that we only verified a simplified model of the protocol (and in a black-box model of cryptography).

In order to model the Logos protocol in PSPSP, we need to make some simplifications and restrictions that are best explained by the enumeration constants:

$$\begin{aligned}\mathsf{hw\_id} &= \{\mathsf{t1}, \mathsf{t2}\}\\ \mathsf{epoch} &= \{\mathsf{e1}, \mathsf{e2}\}\end{aligned}$$

Here we have two terminals (hardware-ids) $\mathsf{t1}$ and $\mathsf{t2}$. A restriction to finitely many terminals is necessary. The server should maintain state for each terminal, namely a set $\mathsf{keys}(T)$ for T: $\mathsf{hw\_id}$. Note that this does not bound the number of sessions each terminal can perform. Second we have two *epochs* $\mathsf{e1}$ and $\mathsf{e2}$. We have introduced these epochs since PSPSP has no explicit notion of time that would allow us, for instance, to reason about whether one event occurred before the other. This model of epochs means that we split the whole timeline into two distinct epochs, without a bound on how many transactions can happen in each epoch. In fact, our model will work in *almost* the exact same way in both epochs (we explain the exceptions at each transaction below). This allows us to make a model that is almost oblivious about time except for one fact: that everything in e1 happens before everything in e2.

First, we define that the intruder can generate public-private key pairs. Note that when no epochs are mentioned, the rule can uniformly be applied in all epochs:

```
intruder_key_gen()
    new $PK$
    insert $PK$ intruderkeys
    send $PK$, inv($PK$).
```

For simplicity, we model here only a single public key of the Logos server. The server can at any time, and in every epoch, generate a new server key and insert it into the set of keys it currently has. This is an over-approximation, i.e., the server in practice would never have more than one key at the same time, but of course the abstraction used in PSPSP implies that everything that can happen, can happen arbitrarily often.

```
server_keys_gen($E$: epoch)
    new $PKL$
    insert $PKL$ server_keys($E$)
    send $PKL$.
```

Note that this can happen in every epoch, but we have parameterized the server keys set over the epoch, so we can distinguish e1-keys from e2-keys. Like in the keyserver examples before, the knowledge of the corresponding private key inv(PKL) is implicit for honest agents like the Logos server.

The fact that all keys could be replaced by new keys we can simply over-approximate by the following "revocation" rule:

```
server_keys_revoke($E$: epoch, $PKL$: value)
    $PKL$ in server_keys($E$)
    delete $PKL$ server_keys($E$).
```

When a batch of new terminals is manufactured, they initially have all the same public/private key pair, and they are just distinguished by their hardware-id. We assume here that batch generation is only in epoch e1, because with e2 we want to model only what happens long after manufacture. Before terminals can be manufactured, we first need to create a public/private key pair, and like for the Logos server, we allow for arbitrary such events and revocation at any point as an over-approximation of several manufacturing batches that can happen in e1:

```
batch_keys_gen()
    new $PK$
    insert $PK$ batch_keys(e1).
```

```
batch_keys_revoke($PK$: value)
    $PK$ in batch_keys(e1)
    delete $PK$ batch_keys(e1).
```

Again, the corresponding private key inv(PK) is implicit, since only the manufactured terminals would have it and are assumed to be honest.

When manufactured, the terminals have also the current public key of the Logos server. They are required to run a bootstrap protocol with the server in order to establish an individual key, and this has to happen within 30 days after manufacture, and we assume here that this is still within epoch e1. (Failure to run it is modeled here by the fact that batch and/or server key can be revoked, and thus the terminal cannot do any transactions.) In the bootstrap protocol, the terminal creates a fresh public key pair BKP, the bootstrap key, and inv(BKP) the corresponding private key. For now even the public key is kept secret between terminal and server:

```
bootstrap_endpoint_terminal($T$: hw_id, $PKL$: value, $PKBatch$: value)
    $PKL$ in server_keys(e1)
    $PKBatch$ in batch_keys(e1)
    new $BKP$
    insert $BKP$ bkp($T$)
    send crypt($PKL$, sign(inv($PKBatch$), pair($T$, $BKP$))).
```

Here we have drastically simplified the bilaterally authenticated TLS session between terminal and server, by encryption of the message with the public key PKL of the server and signing by the private key of the entire batch of terminals inv($PKBatch$). The terminal remembers its bootstrap key BKP (and implicitly the private key) in the set bkp(T). The server receives the public key BKP in the following transaction and stores it in keys(T):

```
bootstrap_endpoint_server($T$: hw_id, $PKL$: value, $BKP$: value, $PKBatch$: value)
    receive crypt($PKL$, sign(inv($PKBatch$), pair($T$, $BKP$)))
    $PKL$ in server_keys(e1)
    $PKBatch$ in batch_keys(e1)
    insert $BKP$ keys($T$).
```

Note again that this bootstrap protocol so far can only be done in epoch e1 as demanded by the parameters of the server and batch key sets, but it can happen for arbitrary many batches and updates of the server keys in e1. Now we come to the truststore protocol which a terminal may execute after many years in storage, so this is possible in both epochs. To initiate, the terminal says its unique hardware ID and a fresh nonce N:

```
truststore_endpoint_terminal($E$: epoch, $T$: hw_id)
    new $N$
    insert $N$ nonce($E$)
    send $T, N$.
```

For simplicity, we omit that this is also done via TLS: since in general neither can verify the certificate of the other, this is not much better than plaintext, and so we do not bother with modeling TLS here. Also, the N is inserted into an epoch-specific set, as the terminal would not accept an answer in epoch e2 with a nonce from epoch e1.

The server now receives the request, looks up the BKP of the claimed T, and constructs a so-called truststore message, telling the terminal all current public key certificates. We have simplified this to the server just telling its own current public key. To this end, the server generates a new shared key SK which is inserted into a set of current shared keys sk_keys, encrypts it with the BKP of the terminal and then authenticates the trust-store by MACing it (and the nonce) with the SK:

```
truststore_endpoint_server_epoch1($T$: hw_id, $BKP$: value, $PK$: value, $N$: value)
    receive $T, N$
    $BKP$ in keys($T$)
    $PK$ in server_keys(e1)
    $N$ notin nonce(e2)
    new $SK$
    insert $SK$ sk_keys($T$, e1)
    insert $PK$ witness($T$, e1)
    send crypt($BKP$, sk($SK$)), $PK$, h($SK, N, PK$).
```

This is the version for the case it is executed in e1. Here we actually make use of the epochs with the requirement N notin nonce(e2). This is not a check that the server can perform, it simply models that this transaction cannot happen in e1 with a nonce that was only created in e2. When this transaction is happening however in e2, we cannot exclude that the nonce was generated in e1 (e.g., the intruder replaying an old request):

```
truststore_endpoint_server_epoch2($T$: hw_id, $BKP$: value, $PK$: value, $N$: value)
    receive $T, N$
    $BKP$ in keys($T$)
    $PK$ in server_keys(e2)
    new $SK$
    insert $SK$ sk_keys($T$, e2)
    insert $PK$ witness($T$, e2)
    send crypt($BKP$, sk($SK$)), $PK$, h($SK, N, PK$).
```

To understand this, consider that actually this approach does not have a direct notion of time built in, and so we cannot directly talk about what has happened before or after. With the epochs we have artificially introduced a minimal notion of time: we distinguish things that have happened in e1 and that is before everything that happens in e2. Thus, we simply can exclude, when this transaction happens in e1 that it can use a nonce of epoch e2—it is simply some information we encode here into the reasoning of the tool.

To continue the bootstrapping protocol, the terminal receives the truststore message from the server:

```
truststore_endpoint_terminal′($T$: hw_id, $BKP$: value, $SK$: value, $PK$: value, $N$: value)
    receive crypt($BKP$, sk($SK$)), $PK$, h($SK, N, PK$)
    $BKP$ in bkp($T$).
```

Actually, we do not model here further the store of the terminal, but we have authentication goals as discussed below.

We formulate several goals by specifying again what would be an attack. First, for secrecy goals: the BKP (both public and private key) are secrets, further the SK and the private key of the terminal batch:

```
secrecy_bkp($T$: hw_id, $BKP$: value)
    receive $BKP$
    $BKP$ in keys($T$)
    attack.
```

```
secrecy_bkp′($T$: hw_id, $BKP$: value)
    receive inv($BKP$)
    $BKP$ in keys($T$)
    attack.
```

```
secrecy_sk($E$: epoch, $T$: hw_id, $SK$: value)
    receive $SK$
    $SK$ in sk_keys($T, E$)
    attack.
```

```
secrecy_batch_key($E$: epoch, $PKBatch$: value)
    receive inv($PKBatch$)
    $PKBatch$ in batch_keys($E$)
    attack.
```

For the receiving of the truststore, we have that it is an attack (non-injective agreement) if the terminal accepts a truststore that was not sent like this from logos (in any epoch):

```
noninjauth_server_keys($T$: hw_id, $PK$: value, $SK$: value, $BKP$: value, $N$: value)
    receive crypt($BKP$, sk($SK$)), $PK$, h($SK, N, PK$)
    $BKP$ in bkp($T$)
    $PK$ notin witness($T$, e1)
    $PK$ notin witness($T$, e2)
    attack.
```

The injective aspect now needs the formulation with epochs: it is an attack if the terminal accepts a message in epoch e2 that the server has indeed said in epoch e1, but not in epoch e2. (It would be ok, if in both epochs it is said by the server.)

```
replay_server_keys($T$: hw_id, $PK$: value, $SK$: value, $BKP$: value, $N$: value)
    receive crypt($BKP$, sk($SK$)), $PK$, h($SK, N, PK$)
    $BKP$ in bkp($T$)
    $N$ in nonce(e2)
    $PK$ in witness($T$, e1)
    $PK$ notin witness($T$, e2)
    attack.
```

This goal we actually found violated in the first version of the Logos protocol: in this first version the terminal did not include the nonce `N` in the truststore protocol (and neither did the server's reply of course). This allows for the following replay attack: a terminal gets manufactured, runs bootstrap and sometime later the truststore protocol with the intruder in the middle, recording the truststore message from the server. The terminal for some reason goes into storage and then later in epoch e2 it runs the truststore protocol again, where the intruder just acts as the logos server and replies with the old truststore messages from `e1`, so the terminal is made to accept the old Logos keys that are long revoked and possibly compromised by the intruder.

We note that this attack is not easy for the intruder to mount, but also not unrealistic, and invalidating exactly what the protocol should achieve, the reliable update of keys. The fix with the nonce is not very expensive and in this version all goals of the protocol are satisfied, including the subsequent server-certification of new keys generated by the terminal which we do not show here.

The PSPSP tool can verify this specification in about a minute.

10 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, 18]). 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., [38, 25, 13, 6, 5, 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 [20, 11, 31]. Goubault-Larrecq [20] 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 [31] developed Scyther-proof [32], an extension to the backward-search used by Scyther [18], which is able to generate proof scripts that can be checked by Isabelle/HOL [36]. 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 [34], but with an important modification. The method of AIF produces a set of Horn clauses that is then analyzed with ProVerif [9] (or SPASS[42]), and the same holds true also for several similar methods for stateful protocol verification, namely StatVerif [3], Set-π [12], AIF-ω [35] 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 [20], 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 [20] 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 [20] 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 [26] 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[12], 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 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 set of (protocol-independent) theorems that imply that any protocol is secure if we

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

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 [33] which can also be regarded as a kind of theorem prover. In fact, it was inspired by the mentioned work of Meier [32] that generates Isabelle proofs from the

Scyther tool, but Tamarin is not based on Isabelle and has rather a specialized proving environment. This in principle shares two very nice features of Isballe: that there are less limitations in what can be modeled, and that a user can supply proof ideas, but it also shares the disadvantage of Isabelle: that most of the interesting proofs are not automatic. There is work on improving this situation, i.e. finding more proofs automatically for Tamarin [17]. In contrast, PSPSP is a complete decision procedure for the class of protocols it supports. Another main difference is here that PSPSP is entirely formalized in Isabelle, and, as explained, does not rely on the correctness of any external tools. Also the core of Isabelle is so well studied and used in so many projects that it can be considered more trust-worthy than the specialized prover of Tamarin. On the other hand, Tamarin can support algebraic properties which we cannot at this point.

Finally, another approach that, like Tamarin, is very much related to performing actual proofs of security protocols automatically and semi-automatically is CPSA [19, 41]. 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.

Acknowledgement. For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission. This work was partially supported by Industriens Fond through the project "Sb3D: Security-by-Design in Digital Denmark".

References

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

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

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

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

[5] G. Bella. *Formal Correctness of Security Protocols.* Information Security and Cryptography. Springer, 2007.

[6] G. Bella, D. Butin, and D. Gray. Holistic analysis of mix protocols. In *Information Assurance and Security*, pages 338–343, 2011.

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

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

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

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

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

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

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

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

[15] Y. Chevalier and L. Vigneron. Automated Unbounded Verification of Security Protocols. In *Computer Aided Verification*, pages 325–337, 2002.

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

[17] V. Cortier, S. Delaune, J. Dreier, and E. Klein. Automatic generation of sources lemmas in TAMARIN: towards automatic proofs of security protocols. *Journal of Computer Security*, 30(4):573–598, Aug. 2022.

[18] C. Cremers. *Scyther: Semantics and verification of security protocols*. PhD thesis, Eindhoven University of Technology, 2006.

[19] S. F. Doghmi, J. D. Guttman, and F. J. Thayer. Searching for shapes in cryptographic protocols. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 523–537, 2007.

[20] J. Goubault-Larrecq. Towards producing formally checkable security proofs, automatically. In *Computer Security Foundations Symposium*, pages 224–238, 2008.

[21] F. Haftmann and L. Bulwahn. Code generation from Isabelle/HOL theories, 2020.

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

[23] A. Hess, S. Mödersheim, and A. Brucker. Stateful protocol composition. In *European Symposium on Research in Computer Security*, pages 427–446, 2018.

[24] A. Hess, S. Mödersheim, and A. Brucker. Stateful Protocol Composition in Isabelle/HOL. *ACM Transactions on Privacy and Security*, 26(3):1–36, 2023.

[25] A. V. Hess and S. Mödersheim. Formalizing and proving a typing result for security protocols in Isabelle/HOL. In *Computer Security Foundations Symposium*, pages 451–463, 2017.

[26] A. V. Hess and S. Mödersheim. A typing result for stateful protocols. In *Computer Security Foundations Symposium*, pages 374–388, 2018.

[27] A. V. Hess, S. Mödersheim, A. D. Brucker, and A. Schlichtkrull. Performing security proofs of stateful protocols. In *34th IEEE Computer Security Foundations Symposium (CSF)*, volume 1, pages 143–158. IEEE, 2021.

[28] A. V. Hess, S. Mödersheim, and A. D. Brucker. Stateful protocol composition and typing. *Archive of Formal Proofs*, April 2020. `https://isa-afp.org/entries/Stateful_Protocol_Composition_and_Typing.html`, Formal proof development.

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

[30] D. Matichuk, T. Murray, and M. Wenzel. Eisbach: A proof method language for isabelle. *Journal of Automated Reasoning*, 56(3):261–282, Mar. 2016.

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

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

[33] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification*, pages 696–701, 2013.

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

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

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

[37] L. C. Paulson. *ML for the working programmer (2nd ed.)*. Cambridge University Press, USA, 1996.

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

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

[40] L. C. Paulson, T. Nipkow, and M. Wenzel. From LCF to isabelle/hol. *Formal Aspects Comput.*, 31(6):675–698, 2019.

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

[42] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *Conference on Automated Deduction*, pages 140–145, 2009.

[43] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, number 4732, pages 352–367. 2007.

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).