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

Andreas Viktor Hess [a], Sebastian Alexander Mödersheim [a], Achim D. Brucker [b,*] and Anders Schlichtkrull [c]

[a] *DTU Compute, Technical University of Denmark, Denmark*
E-mails: avhe@dtu.dk, samo@dtu.dk
[b] *Department of Computer Science, University of Exeter, United Kingdom*
E-mail: a.brucker@exeter.ac.uk
[c] *Department of Computer Science, Aalborg University Copenhagen, Denmark*
E-mail: andsch@cs.aau.dk

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

Keywords: Formal Verification, Theorem Proving, Security Protocols

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 Rocq. First, it gives us an overwhelming assurance that the proof of security is actually a proof and not just the result of a bug in a complex verification tool. This is because the basic idea of an LCF-style theorem prover is to have an abstract datatype *theorem* so that new theorems can only be constructed through functions that correspond to accepted proof rules; thus implementing just this datatype correctly prevents us from ever accepting a wrong proof as a theorem, no matter what complex machinery we build for automatically finding proofs. Second, a human may have an insight of how to easily prove a particular statement where a "stupid" verification algorithm may run into a complex check or even be infeasible. Third, the language of a proof assistant can formalize all accepted mathematics, so there is no narrow limit on what aspects of a system we can formalize. For instance, we have proved in Isabelle/HOL a compositionality result [1] for our protocol model: given a set of protocols for which we have proved security and that meet a number of requirements, then also their composition is correct.

[*] Corresponding author. E-mail: a.brucker@exeter.ac.uk.

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 [2] and Bella [3] developed a protocol model in Isabelle and performed several security proofs in this model, e.g., [4]. 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 generate machine-checkable proofs. [41] shows how one can verify Horn-clause based descriptions (e.g., in ProVerif [5] internally) by finding a (finite) model for it that can easily be machine-checked. Scyther-proof [6] produces Isabelle proofs for the backward search-based tool Scyther [7].

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 several tools for infinite-sessions like Scyther. For instance ProVerif is based on an abstract interpretation that basically abstracts away from states (and time). To overcome these limitations, several amendments to the abstraction were suggested, like set-based abstraction [8] which our approach is based on, StatVerif [9] and GSVerif [10], and many of these ideas are now incorporated into ProVerif.

Moreover, there is also a tool that went a completely different way: Tamarin [11] (which also is not limited to protocols without long-term state) is actually inspired by Scyther-proof and has the flavor of a proof assistant environment itself, namely combining partial automation with interactive proofs, 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. [12]. There are several reasons for this choice. First, the proof technique we present in this paper works only in a restricted typed model. Fortunately, that formalization ships with a typing result [13], 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.

Second, as described already in [1], building on this protocol model has allowed us to seamlessly integrate the PSPSP tool with the compositional reasoning results: a user can specify multiple protocols in PSPSP and give annotations for their composition. The PSPSP tool can then automatically check the requirements for the composition, automatically verify each component individually, and obtain a proof of the composed system entirely verified by Isabelle/HOL from beginning to end.

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 run-

time of automated analysis is high, a human prover may have an idea to prove some aspect more easily avoiding, e.g., some lengthy enumerations.

The automated proof technique we employ in PSPSP is based on the set-based abstraction approach of [14, 15]. 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 follow 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 is 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 abstract 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 the 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. The language additionally allows the specification of analysis rules which are rules that express how the intruder can extract knowledge from received messages built using cryptographic functions.

New contributions in this journal paper. The aforementioned contributions were also part of the conference version of the present paper [16]. In the present journal version we additionally have a number of new contributions and 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 for two of our examples. We have also improved the check that the fixed point is analyzed, i.e. covers also the knowledge that the intruder can gain by applying the analysis rules.

Second, we have improved the user experience of the tool, e.g., by better error messages and support for understanding 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. Our tool can now calculate such traces and present them to the user.

Third, we have two improvements on the semantic level of PSPSP. One improvement on the semantic level of PSPSP is that we have a soundness proof for a transformation that the tool is doing. The abstract interpretation approach requires that all values are sent out in the network as part of some message. However our syntax for transactions does not enforce this, because it is reasonable to have a transaction that inserts a value in a set/database without sending it out as part of some message. Our tool solves this by a transformation that for every newly generated value v in a transaction sends out a special message $\mathsf{occurs}(v)$ and modifies each transaction to receive $\mathsf{occurs}(v)$ for every non-fresh value v. 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 improvement concerns the definition of the semantics of transactions. While it is defined via symbolic traces of unbounded length, (which is particularly practical for relative soundness results like typing and compositionality), we now have also proved the equivalence with a more standard ground semantics.

Fourth, 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* [17]:

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 [1]), 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 $f(t_1, \ldots, t_n)$ where each t_i is a term built from S and X, and $f \in S^n$. The set of terms built from S and X is denoted by $\mathcal{T}(S, X)$. Arbitrary terms t usually range over $\mathcal{T}(\Sigma, \mathcal{V})$, unless stated otherwise. By $\mathit{subterms}(t)$ 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 disjoint 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.

Terms represent the messages sent during the run of a protocol and the concrete values are atomic parts of these messages that are generated during the execution of the protocol such as public keys, nonces, shared keys and other data. The abstract values are then abstractions of the concrete values in the abstract interpretation approach that we are using. We elaborate on these in Section 4 and choose there a specific set for $\mathbb{A}$ that suits the concrete abstract interpretation approach we are using.

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) which represent constructors and cryptographic functions used to build the messages.

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 $x_1, \ldots, x_n$ of variables.

We furthermore partition Σ into the *public* symbols (those symbols that are available to the intruder) and the *private* symbols (those that are not). We denote by Σ_{pub} and Σ_{priv} the set of public 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 $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ to denote the substitution with domain $\{x_1, \ldots, x_n\}$ and range $\{t_1, \ldots, t_n\}$ that sends each x_i to t_i. Substitutions are extended to composed terms homomorphically as expected. A substitution θ should be applied to a term t only when $fv(t) \subseteq dom(\theta)$. 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.

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

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 [12] 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:

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

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

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

The $(Axiom)$ rule simply expresses that all messages directly known to the intruder are derivable, the $(Compose)$ rule closes the derivable terms under the application of public function symbols such as encryption or public constants (when $f \in \Sigma^0_{pub} = \mathcal{C}_{pub}$). The $(Decompose)$ rule represents decomposition operations: $\mathsf{Ana}(t) = (K, R)$ means that t is a term that can be analyzed, provided that the intruder knows all the "keys" in the set K, and he will then obtain the "results" in R. This gives us a general way to deal with typical constructor/destructor theories without needing to work with algebraic equations and rewriting. We may also write $\mathsf{Keys}(t)$ and $\mathsf{Result}(t)$ to denote the set of keys respectively results from analyzing t, i.e., $\mathsf{Ana}(t) = (\mathsf{Keys}(t), \mathsf{Result}(t))$.

Example 2. *To model asymmetric encryption and signatures we first fix two public* $\mathsf{crypt}, \mathsf{sign} \in \mathbb{F}^2$ *and one private* $\mathsf{inv} \in \mathbb{F}^1$ *function symbols. The term* $\mathsf{crypt}(k, m)$ *then denotes the message m encrypted with a public key k and* $\mathsf{sign}(\mathsf{inv}(k), m)$ *denotes m signed with the private key* $\mathsf{inv}(k)$ *of k. To obtain a message m encrypted with a public key k the intruder must produce* $\mathsf{inv}(k)$*. 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 f we have at most one analysis rule. For those constructors f that have such a rule we define for each term $f(t_1, \ldots, t_n)$ the function Ana as $\mathsf{Ana}(f(t_1, \ldots, t_n)) \equiv \mathsf{Ana}_f(t_1, \ldots, t_n)$ and for all constructors g without an analysis rule we just have $\mathsf{Ana}(g(t_1, \ldots, t_n)) \equiv (\emptyset, \emptyset)$. (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 [12] requires that the Ana function satisfies certain conditions, most notably that it is invariant under substitutions.[2] Without going into detail, our notation of the Ana rules allows for an automated proof that all these requirements are satisfied. Thus, this allows the user to specify an arbitrary constructor/destructor theory with these Ana rules without having to prove anything manually.

2.3. Typed Model

PSPSP works in a typed model. That means every variable has an intended type and the intruder can send only well-typed messages. Many protocol verification methods [2–4, 18, 19] rely on such a typed model since it simplifies the protocol verification problem. There exist many typing results [13, 20–24] that show that a restriction to a typed model is *sound* for large classes of protocols: for a protocol falling in such a class it is without loss of attacks to restrict the intruder to well-typed messages. [13] is such a result that is part of the Isabelle formalization we employ. Since this result has itself been proved in Isabelle, to obtain the Isabelle proof that a protocol is secure in the unrestricted model, it is sufficient to have a proof that the protocol is secure in the typed model (e.g., automatically using PSPSP) and that the protocol is satisfies *type-flaw resistance*, i.e., the requirements of the typing result. We have automated the Isabelle proof of type-flaw resistance for the protocol specification language we present. Note that PSPSP only assumes a typed model, and the typing result here is just one way to discharge this assumption; there may be other ways to prove it, or one may leave it as an assumption. As PSPSP is not conceptually tied to the typing result, we give here only a brief summary of the result of [13].

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.

For instance, we may have a transaction with a step $\mathsf{receive}(X)$ where $\Gamma(X) = h(\mathsf{Nonce})$. Note that in general the recipient would be unable to check if the received message is really of this form (e.g., the intruder could send them any value); the typing result shows that (if the protocol satisfies type-flaw resistance), if there is an attack, then there is a well-typed attack, and it is thus sound to replace the above step by $\mathsf{receive}(h(N))$ for a new variable N of type Nonce (and replace all occurrences of X with $h(N)$ in the rule). This does not mean that the recipient would learn N, but only that the term can only be of the form $h(N)$ for some N of type Nonce.

For such a typing result, one considers a set of sub-message patterns (*SMP*) that contains the messages of the protocol description and is closed under well-typed instances, subterms, and key terms that can occur during analysis steps. For instance with the operators in Example 2, if $\mathsf{crypt}(k, m) \in \mathit{SMP}$, then also $k, m, \mathsf{inv}(k) \in \mathit{SMP}$. The main requirement of type-flaw resistance is: if two terms $s, t \in \mathit{SMP} \setminus \mathcal{V}$ have a unifier, then $\Gamma(s) = \Gamma(t)$.

For example, if *SMP* contains messages $s = \mathsf{scrypt}(X, \mathsf{pair}(A, Y))$ and $t = \mathsf{scrypt}(X', Z)$ for $\Gamma(X) = \Gamma(X') = \Gamma(Y) = \Gamma(Z) = \mathsf{value}$ and $\Gamma(A) = \mathsf{enum}$, then s, t have a unifier, but $\Gamma(s) \neq \Gamma(t)$, so this

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

violates type-flaw resistance. There is a cheap way to make basically any protocol type-flaw resistant: instead of pairs consider *formats*, i.e., public functions that represent a particular way of structuring clear-text messages (e.g. XML), one format for each type of message. In the example it can a be binary format f_1 and a unary format f_2, and using them for the different types of encrypted messages: $s = \mathsf{scrypt}(X, f_1(A, Y))$ and $t = \mathsf{scrypt}(X', f_2(Z))$. Now s and t are no longer unifiable, since they use different formats, and it is just following prudent engineering principles that one should not encrypt raw data like nonces, but add a few bits identifying what type of data it is.

As mentioned, we have now automated this check for type-flaw resistance; of course, *SMP* is infinite, so we cannot directly compute it, but it is rather straightforward to compute the property symbolically. (One can avoid concrete constants and does not need to consider variants of terms that are equal modulo renaming of variables.)

The typing result for type-flaw resistant protocols is proved in [13] by considering a symbolic search technique called *the lazy intruder*. The key idea is that the only way that the lazy intruder instantiates variables is by a substitution σ that is the most general unifier of two non-variable terms. For a type-flaw resistant protocol, the considered terms are always within SMP and the unifier between two non-variable terms of SMP is thus necessarily well-typed (thus, applying the unifier to other terms in the search keeps them also in SMP). Because the technique is sound and complete, it will find an attack if there is one, and this attack is well-typed for a type-flaw-resistant protocol.

In fact, like [13], we consider here stateful protocols where a message m can be inserted, deleted, and checked for containment/non-containment in a set s. The requirement here is the following: if we have in the protocol a set operation with message m and set s and another set operation with message m' and set s' and if the pairs (m, s) and (m', s') have a unifier, then they must be of the same type. For instance, we cannot put messages of different types into the same set.

In this paper we use as atomic types only $\mathfrak{T}_a = \{\mathsf{value}, \mathsf{enum}, \mathsf{settype}, \mathsf{attacktype}\}$, and the elements of $\mathbb{A} \cup \mathbb{V}$ have type value, the elements of $\mathbb{E}$ have type enum, the elements of $\mathbb{S}$ have type settype and attack has type attacktype.

As a consequence of the typing result, when we have a variable X of a composed type $\Gamma(X) = f(\tau_1, \ldots, \tau_n)$, it is sound to replace it by the term $f(X_1, \ldots, X_n)$ where the X_i are fresh variables of type τ_i. Repeatedly applying this, it is thus not a restriction to have variables of only atomic types.

3. Transactions

The Isabelle protocol model of [12] consists of a number of *transactions* (also called *rules*) 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, [12] 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. To begin with we introduce this language using a keyserver example adapted from [12] 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. The protocol specifies $\mathbb{E} = \{\mathsf{a}, \mathsf{b}, \mathsf{i}\}$ and uses the specification of $\mathbb{S}$ from Example 1 and of $\mathbb{F}$ from Example 2. We consider users a and b as honest users and i as dishonest. In the specification we will use sets $\mathsf{user} = \mathbb{E}$, $\mathsf{honest} = \{\mathsf{a}, \mathsf{b}\}$ and $\mathsf{dishonest} = \{\mathsf{i}\}$ as types in the parameters of the transaction we specify. In the keyserver example, users can register public keys at a trusted keyserver and these keys can later be revoked. Each user U has an associated keyring $\mathsf{ring}(U)$ with which it keeps track of its keys. (The elements of $\mathsf{ring}(U)$ are actually public keys; we implicitly assume that the user U knows the corresponding private key.)

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

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 $\mathsf{ring}(U)$ is a set of public keys controlled by U (and U has the corresponding private key of each) while $\mathsf{valid}(U)$ is controlled by the server (who is not even given a name here). Putting it into a single transaction models that this is something happening in collaboration between a user and a server.

Next, we model a key update mechanism that allows for registering a new key while simultaneously revoking an old one. We model this as two transactions, one for the user and one for the server, since here we model a scenario where user and server communicate via an asynchronous network controlled by the intruder. To initiate the key revocation process the user U first picks and removes a key PK from its keyring to later revoke, then freshly generates a new key NPK and stores it in its keyring. (Again the corresponding private key $\mathsf{inv}(NPK)$ is known to U, but this is not explicitly described.) As a final step the user signs the new key with the private key $\mathsf{inv}(PK)$ of the old key and sends this signature to the

server by transmitting it over the network:

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

$$
\begin{array}{lcl}
S_r & ::= & \mathsf{receive}\ t_1, \ldots, t_n \cdot S_r \mid 0 \\
S_c & ::= & x\ \mathsf{in}\ s \cdot S_c \mid x\ \mathsf{notin}\ s \cdot S_c \mid x \neq x' \cdot S_c \mid 0 \\
F & ::= & \mathsf{new}\ x \cdot F \mid 0 \\
S_u & ::= & \mathsf{insert}\ x\ s \cdot S_u \mid \mathsf{delete}\ x\ s \cdot S_u \mid 0 \\
S_s & ::= & \mathsf{send}\ u_1, \ldots, u_n \cdot S_s \mid 0
\end{array}
$$

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. We remark that only values can be inserted in the sets which is why we require $x, x' \in \mathcal{V}_{\mathsf{value}}$. This is needed for the abstract interpretation approach we present in Section 4.

The function *fv* is extended to transactions by letting it collect all variables occurring in the strands of the transaction, 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 $\mathcal{P} = \{T_1, \ldots, T_n\}$. 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 have been used already and thus cannot be considered 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. To ensure that these values are indeed fresh they must not be in C already. 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* or *traces*:

$$
\begin{array}{lcl}
\mathcal{A} & ::= & \mathsf{send}\ t_1, \ldots, t_n \cdot \mathcal{A} \mid \mathsf{receive}\ t_1, \ldots, t_n \cdot \mathcal{A} \mid t \neq t' \cdot \mathcal{A} \mid \mathsf{insert}\ t\ t' \cdot \mathcal{A} \mid \mathsf{delete}\ t\ t' \cdot \mathcal{A} \mid \\
 & & t\ \mathsf{in}\ t' \cdot \mathcal{A} \mid t\ \mathsf{notin}\ t' \cdot \mathcal{A} \mid 0
\end{array}
$$

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

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

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 when a constraint $\mathcal{A}$ is executed: $ik(\mathcal{A}) = \{t_i | 1 \leqslant i \leqslant n, \mathsf{receive}\ t_1, \ldots, t_n \in \mathcal{A}\}$. We also define the database $db(\mathcal{A}, \mathcal{I}, D)$, which is the update of the database D after the execution of the constraint $\mathcal{A}$ from database D under interpretation $\mathcal{I}$:

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

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

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

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 an injective 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) $\mathcal{I} \models^M_D \mathcal{A}$.

A configuration (M, D, C) is said to be *ground reachable in* $\mathcal{P}$ iff $(\emptyset, \emptyset, \emptyset) \Rightarrow^\star_\mathcal{P} (M, D, C)$ where $\Rightarrow^\star_\mathcal{P}$ 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 $\Rightarrow^{\bullet}_{\mathcal{P}}$ for protocols $\mathcal{P}$ in which states are constraints and the initial state is the empty constraint 0. The transition

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

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 an injective 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 not in $\mathit{fresh}(T)$ to new variables that do not occur in $\mathcal{A}$ or $\mathcal{P}$ (that is, $\mathit{dom}(\rho) = \mathit{fv}(T) \setminus \mathit{fresh}(T)$ 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 $\mathcal{I} \models \mathcal{A} \cdot \mathsf{send}\,\mathsf{attack}$. 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 to each other:

Theorem 1. [3]

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

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 [15] 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, if a transaction uses a variable for 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,

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

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: $fv(S_u) \cup fv(S_s) \subseteq fv(S_r) \cup fv(S_c) \cup fresh(T)$
C2: $fresh(T) \cap (fv(S_r) \cup fv(S_c)) = \emptyset$
C3: $fresh(T) \subseteq fv(S_s) \cup \{x \mid \mathsf{insert}\ x\ s \in S_u\}$

The second condition simply states that values that are freshly generated by a transaction T must neither occur in the received messages nor in the checks of T.

A second concern is whether protocols allow the intruder to actually produce fresh values. That is not ensured in the protocol model we have presented so far, but the intruder being able to produce fresh values is standard in many models of security protocols and we want it to be also the case for our work. Suppose e.g. that a specification contains no transaction that generates any fresh value, but, say, only an attack rule like this:

attackDef2(PK: value)
  receive PK
  attack.

One might naturally expect that said protocol is not secure, but this is only the case if the intruder can actually produce fresh values.

We will therefore require that protocols include transactions that immediately produce and reveal fresh values to the intruder. We denote such transactions as initial-value-producing transactions. 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 an initial-value-producing transaction:

Definition 1. *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 $t_1, \ldots, t_n$ *where $t_i = x$ for some i, no other variable than x occurs in $t_1, \ldots, t_n$ 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.

A third concern is that the abstraction approach that we employ, 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). To avoid bothering the user with this, our tool can make this transformation automatically using the following function:

Definition 2. *Let $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$ be a transaction, let $\{y_1, \ldots, y_n\}$ be its (possibly empty) set of fresh variables and let $\{x_1, \ldots, x_m\}$ be its (possibly empty) set of other free variables.*

We define a function add_occurs_msgs that ensures that occurs *messages are being received and sent:*

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

where

$$S'_r = \textbf{if } \{x_1, \ldots, x_m\} = \emptyset \textbf{ then } S_r \textbf{ else } \mathsf{receive}(\mathsf{occurs}(x_1), \ldots, \mathsf{occurs}(x_m)) \cdot S_r$$

$$S'_s = \textbf{if } \{y_1, \ldots, y_n\} = \emptyset \textbf{ then } S_s \textbf{ else } \mathsf{send}(\mathsf{occurs}(y_1), \ldots, \mathsf{occurs}(y_n)) \cdot S_s$$

Note that the occurs messages are only added during verification. One may wonder if the transformation is sound – after all, there is now the additional requirement that the intruder needs to send occurs messages in order to run the transactions. The transformation *is* sound, and the argument is essentially that if there was an attack in the original protocol, then the transformed protocol will have a similar attack that simply receives and sends the introduced occurs messages as needed. However, proving the transformation sound involves a challenge that stems from an interaction between the modified protocol and the typing result from [13]. A requirement of applying the typing result is namely that there are infinitely many *public* constants of type value that the intruder can get access to through the Compose rule.[4] This is in contrast to the presentation we gave in Section 2 where we defined the type value to contain only *private* values (i.e. $\mathbb{V}$). The public constants of type value are problematic because the intruder does not have occurs messages for those. Furthermore, one might fear that attacks that used public constants of type value before the transformation are "blocked" after the transformation because of them not having corresponding occurs messages. As part of the following soundness proof we will therefore prove that whenever the intruder might have used one of these public constants of type value, he can instead use private constants of type value that he obtains by using an initial-value-producing transaction. This is both central to the soundness of the occurs transformation and for us to justify not including public constants of type value in the model presented in Section 2. 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 private ones:

[4]The reason is that one of the main arguments used in the typing result is essentially that for attacks on type-flaw resistant protocols that use ill-typed messages, the intruder might as well have picked well-typed messages. For this to be true for an unbounded number of messages containing terms that should have had an atomic type like value, the intruder thus needs to be able to pick enough of such constants and this is what this requirement ensures.

Lemma 1. *Let $\mathcal{P}$ be a protocol that includes an initial-value-producing transaction and which has a well-typed attack $\mathcal{A} \cdot \mathsf{attack}$ with model $\mathcal{I}$. Then there exists constraint $\mathcal{B}$ and interpretation $\mathcal{J}$ such that $\mathcal{B} \cdot \mathsf{attack}$ 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 from $\mathbb{V}$.*

Proof Sketch. 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 $T = S_r \cdot S_c \cdot F \cdot S_u \cdot S_s$ to $\mathcal{A} \cdot \mathit{dual}(\rho(\sigma(S_r \cdot S_c \cdot S_u \cdot S_s)))$ and then show that the corresponding $\mathcal{B}$ can 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 an initial-value-producing transaction and which has a well-typed attack. Then the protocol $\{\mathit{add\_occurs\_msgs}(T) \mid T \in \mathcal{P}\}$ also has a well-typed attack.*

Proof Sketch. 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 $\{\mathit{add\_occurs\_msgs}(T) \mid T \in \mathcal{P}\}$. The third step proves that $\mathcal{J}$ is also a model of $\{\mathit{add\_occurs\_msgs}(T) \mid T \in \mathcal{P}\}$. □

The final concern to be discussed in this section is a small technical difficulty that 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 [15] simply defines the semantics to be injective on variables.[6] 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 yields a consistent transaction, i.e. a transaction that avoids logically inconsistent checks such as x in ring(a) followed by x notin ring(a), which will necessarily fail. If so, we add it to the protocol. Then we add the constraint $x \neq y$ to the original rule. We do that until all rules have $x \neq y$ for all pairs of variables that are not freshly generated. For instance, in the keyserver example, we have only one rule to look at: keyUpdateServer with variables *PK* and *NPK*. Since unifying *PK* and *NPK* gives an unsatisfiable rule, it is safe to add $PK \neq NPK$ to it.

4. Set-Based Abstraction

We now come to the core of our approach: for a given protocol, how to automatically verify and generate a security proof that Isabelle can accept. As explained earlier, this is based on an abstract

[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.

[6] We elaborate on this in Section 5.1 after Definition 8.

interpretation method called set-based abstraction [8, 14, 15]. Essentially the method computes a fixed point which over-approximates the intruder's accumulated knowledge in any sequence of transactions. 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. The core of PSPSP is this coverage check that automatically performs a security proof in Isabelle. The fixed point can thus be regarded as a proof idea that no trace can contain an attack. If there was any mistake in the fixed point computation (or if the abstraction approach were actually unsound), then in the worst case Isabelle would fail to be convinced by the coverage check.

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 $\models^M_D$, 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). Note also that the values that can no longer be considered fresh are the set $C = \mathit{subterms}(\mathcal{A}) \cap \mathbb{V}$. We can thus characterize the *state* of the entire system after a number of instantiated transactions by these three items M, D and C.

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* $[PK \mapsto \mathsf{pk}_1, U \mapsto \mathsf{a}, NPK \mapsto \mathsf{pk}_2]$*):*

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

Suppose we start in state $M_0 = \emptyset$, $D_0 = \emptyset$ and $C_0 = \emptyset$. After this trace we have

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

□

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. We thus write A for the abstract value that corresponds to the subset A of $\mathbb{S}$, e.g., if $s_1, s_2 \in \mathbb{S}$ then $\{s_1, s_2\} \in \mathbb{A}$. Remember that $\mathbb{A}$ is included in Σ^0 so we can build *abstract terms* that include elements of $\mathbb{A}$ as *abstract constants*.[7]

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 [8, 14, 15] do not directly compute the fixed point 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 [13] 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 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 [12].) Thus, it is sound to limit the fixed point and the intruder deduction to well-typed terms, which makes the fixed point finite.

4.1. Defining Fixed Points

Let us now see in more detail how to formally define 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 in 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 the abstraction of values play a crucial role in the approach, we define the following notion:

[7] In fact, in the Isabelle type system, Σ cannot contain both $\mathbb{S}$ and subsets of $\mathbb{S}$; we thus technically define the elements of $\mathbb{A}$ as a constructor applied to any subset of $\mathbb{S}$. This is reflected here by the shading to ease presentation.

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 abstraction 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 now 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 $c \;{}_a{\twoheadrightarrow}_b\; c$ *for* $c \in \mathbb{A} \subseteq \Sigma^0$ *holds because of 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:*

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

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*).

Closing the fixed point under the term implication graph 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,*[8] *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: $TI \subseteq \mathbb{A} \times \mathbb{A}$*.*

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

4.2. Limitations

There are some limitations of our approach that we now mention. First, PSPSP is limited to reachability properties. This is because privacy-type properties are quite hard to handle in infinite state verification. To our knowledge, regarding automated tools for infinite sessions, only Tamarin [11] and ProVerif [5] handle this by restriction to diff-equivalence [25].

We inherit the free algebra term model from [12] (two terms are equal iff they are syntactically equal) and so we do not support algebraic properties such as needed for Diffie-Hellman. 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. That would not work if we allowed composed terms in the sets.
- 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.
- We cannot check the equality of values, again because two values may have the same abstraction.
- As the following protocol shows, the abstraction may lose "connections" between values and this can lead to spurious attacks. The example illustrates this using the transactions send_h and del_set which result in the intermediate fixed point $\{\mathsf{h}(\{\mathsf{set}\},\{\mathsf{set}\})\}$ and the term implication $\{\mathsf{set}\} \twoheadrightarrow \emptyset$. Thus under term implication we get the terms $\mathsf{h}(\{\mathsf{set}\},\{\mathsf{set}\})$, $\mathsf{h}(\emptyset,\emptyset)$, $\mathsf{h}(\{\mathsf{set}\},\emptyset)$ and $\mathsf{h}(\emptyset,\{\mathsf{set}\})$. The latter term triggers attack_def which emits an attack. However, on the concrete level this attack is not possible because there any concrete chosen values for *N1* and *N2* in $\mathsf{h}(N1,N2)$ are either both in set or both not in set. We remark that many protocols do not rely on such "connections" between values as demonstrated by our benchmark suite.

send_h()
- new *N1*
- new *N2*
- insert *N1* set
- insert *N2* set
- send $\mathsf{h}(N1,N2)$.

del_set(*N1*: value, *N2*: value)
- receive $\mathsf{h}(N1,N2)$
- delete *N1* set
- delete *N2* set.

attack_def(*N1*: value, *N2*: value)
- receive $\mathsf{h}(N1,N2)$
- *N1* in set
- *N2* notin set
- attack.

Our approach allows for an unbounded number of sessions. The only difference here between our work and, e.g., Tamarin [11] and ProVerif [5] 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. There is potential for overcoming this limitation: Comon-Lundh and Cortier [26, 27] show that a restriction to finitely many agents can often be without a loss of attacks. [15] indirectly shows that similar results are possible for set-based abstraction, since one can compute a symbolic fixed point leaving agents uninstantiated variables. This however requires several restrictions on the use of negation, suggesting close limits of such an approach for set-based abstraction. As this would have also required a

substantially more complicated concept for the fixed point computation and checking, we opted for not using this approach in PSPSP/Isabelle.

A minor limitation is that the method of PSPSP requires a typed model. In order to also be safe against type-flaw attacks one needs to employ a typing result like [13] and thus also needs to satisfy the requirements of that result, but this is not a serious limitation for the class of protocols we consider and it is automatically checked (see Section 2.3).

Finally, PSPSP has a number of smaller limitations that we list here for completeness:

- We require that all variables must have atomic type. This restriction could actually be overcome because for a type-flaw resistant protocol it is sound to replace a variable of composed type with a corresponding composed term of atomic type variables. However we consider it out of scope to implement and formally verify this transformation and its soundness and thus require that all variables must have atomic type.
- Our definition of the $(Decompose)$ rule using Ana_f means that one cannot let secret keys be values SK and then define the corresponding public key as $\mathsf{pk}(SK)$. This is because Ana_f does not allow definitions on the form $\mathsf{Ana}(\mathsf{crypt}(\mathsf{pk}(k), m)) = (\{k\}, \{m\})$. We consider this a small restriction because, as the examples show, one can instead let the public keys be values, and then have a function inv mapping them to private keys.
- We are using pattern matching in the receipt of messaging. The user of our tool thus needs to understand that to implement a specified protocol they need their implementation to reject messages received that are not in the expected format. This is in contrast to ProVerif's π-calculus [5] where the steps checking the format are stated more explicitly in the specifications. We do not consider this a significant limitation because this kind of pattern matching is quite standard in the modeling of security protocols, e.g., [11, 28].

4.3. Example of a Fixed Point Computation

Consider again the keyserver protocol defined in Section 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

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

The term implication graph TI_{ks} can be represented graphically as follows with each edge $a \twoheadrightarrow b$ corresponding to an element of TI_{ks}:

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

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:

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

To compute FP_{ks}, we first consider the transaction outOfBand where a fresh key is inserted into both ring(a) and 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.[9] Note that this rule cannot produce anything else so we do not consider it for the remainder.

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

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

Applying the keyUpdateServer transaction to the first signature we have obtained (i.e., with $PK = \{\mathsf{valid(a)}\}$ and $NPK = \{\mathsf{ring(a)}\}$), we get the term implications $\{\mathsf{valid(a)}\} \twoheadrightarrow \{\mathsf{revoked(a)}\}$ and $\{\mathsf{ring(a)}\} \twoheadrightarrow \{\mathsf{ring(a)}, \mathsf{valid(a)}\}$, and the intruder learns $\mathsf{inv}(\{\mathsf{revoked(a)}\})$. Applying the keyUpdateServer transaction with the second signature (i.e., with $PK = \emptyset$ and NPK as before) is not possible because $\mathsf{valid(a)} \notin \emptyset$ and thus the transaction's check of $PK \in \mathsf{valid(a)}$ will not be satisfied. However, because of the intruder's knowledge of the first signature and the previously stated term implication $\{\mathsf{ring(a)}\} \twoheadrightarrow \emptyset$, the intruder also knows the signature $\mathsf{sign}(\mathsf{inv}(\{\mathsf{valid(a)}\}), \mathsf{pair}(\mathsf{a}, \emptyset))$. If we apply the keyUpdateServer transaction with this signature (i.e., with $PK = \{\mathsf{valid(a)}\}$ and $NPK = \emptyset$) then we get the new term implication $\emptyset \twoheadrightarrow \{\mathsf{valid(a)}\}$, the term implication $\{\mathsf{valid}(a)\} \twoheadrightarrow \{\mathsf{revoked}(a)\}$, which we already knew, and the intruder learns $\mathsf{inv}(\{\mathsf{revoked}(a)\})$, but that was already in the intruder knowledge.

Any other signature that the intruder knows as a consequence of the two signatures sent by the agent and the term implications will not satisfy the checks of the keyUpdateServer transaction. Note though, that we must also check if the intruder can generate a signature that works with keyUpdateServer: however, the only private keys he knows are those represented by $\mathsf{inv}(\{\mathsf{revoked(a)}\})$, and they are not accepted for this transaction. (In a model with dishonest agents, the intruder can of course produce signatures with keys registered to a dishonest agent name, but here we have just one honest user a.)

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

5. Computing Fixed Points and Checking Fixed Point Coverage

In this work we automatically calculate fixed points with the abstract interpretation approach as a "proof idea" for conducting the security proof on the concrete protocol. A major contribution of our work is the ability to turn this fixed point into a formal security proof in Isabelle. Essentially, we prove that the fixed point indeed "covers" everything that can happen. We break this down into an induction

[9] 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).

proof: given any trace that is covered by the fixed point, if we extend 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*. The induction is done in the proof of a protocol-independent Isabelle theorem (Theorem 3) that shows that any protocol that passes said checks is indeed secure. Note that these checks are not only fully automated, but they are also terminating in all but a few degenerate cases.[10]

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

Definition 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 two next checks, F2 and F3, verify that 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 check that is already covered by FP. The transaction can produce outgoing messages and 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:

[10] 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.

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:*

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

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

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 $\mathcal{I}(x) = \mathcal{I}(y)$. Thanks to the preparations described in Section 3.3 we have in every transaction the constraint that all value variables have pairwise disjoint concrete values.

This leads to the following two post-conditions:

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. $\delta(x) \twoheadrightarrow^* \delta_u(x)$ *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 correspond to 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: the pre-conditions restrict the coverage check to those abstraction substitutions that are actually possible in the fixed point. The post-conditions check 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. [11] *Let $\mathcal{P}$ be a protocol and let $\mathsf{FP} = (FP, TI)$ be a fixed point. If $\mathsf{attack} \notin FP$, and if $\mathcal{P}$ is covered by* FP*, then $\mathcal{P}$ is secure.*

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

Proof Sketch. We gave the intuition of this proof in introductory text of this section. We supplement that here with some further intuition. First, we define a notion of abstract intruder knowledge namely $\alpha_{ik}(\mathcal{A},\mathcal{I}) = \alpha_{db(\mathcal{A},\mathcal{I},\emptyset)}(\mathcal{I}(ik(\mathcal{A})))$. The proof then relies on two insights about this abstract intruder knowledge.

The first insight is that if a term t is in the intruder knowledge then its abstraction is also in the abstract intruder knowledge. Formally if $t \in \mathcal{I}(ik(\mathcal{A}))$ then $\alpha_{db(\mathcal{A},\mathcal{I},\emptyset)}(t) \in \alpha_{ik}(\mathcal{A},\mathcal{I})$.

The second insight is to prove that for any reachable constraint $\mathcal{A}$, with satisfying interpretation $\mathcal{I}$, it is the case that any message t (e.g. the attack constant) in the abstract intruder knowledge is also derivable from the fixed point. Formally if $t \in \alpha_{ik}(\mathcal{A},\mathcal{I})$ then $cl_{TI}(FP) \vdash t$. The proof of this second insight is by an induction on how $\mathcal{A}$ was reached by $\Rightarrow^{\bullet}$. The central part is the induction step where a reachable constraint $\mathcal{A}'$ is extended with the dual of a transaction T. Here we see that the abstract intruder knowledge achieved by running the dual of T is a result of messages sent in T. The definition of coverage ensures that these sent messages are also derivable from the fixed point's term implication closure. □

Example 6. *Consider the key update transaction* keyUpdateServer *from Section 3.1. We now show that the fixed point* FP_{ks} *defined in Section 4.3 covers this transaction, i.e., satisfies Definition 10.*

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 these 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* $\mathsf{valid}(\mathsf{a}) \in \delta(PK)$ *and* $\mathsf{valid}(\mathsf{a}), \mathsf{revoked}(\mathsf{a}) \notin \delta(NPK)$*, 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* $\mathsf{valid}(\mathsf{a}) \in b$*. Hence, the only signatures available to him—that also satisfy the constraints for* Δ *that we have deduced so far—are* $\mathsf{sign}(\mathsf{inv}(\{\mathsf{valid}(\mathsf{a})\}), \mathsf{pair}(\mathsf{a}, b))$ *for each* $b \in \{\{\mathsf{ring}(\mathsf{a})\}, \emptyset\}$*. The only surviving substitutions are*

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

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

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

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

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 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:*

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

where

$$\Delta^T_{FP,TI} \equiv \{\delta \mid dom(\delta) = fv(T), \mathsf{pre}((FP, TI), T, \delta)\}$$
$$\widehat{FP}^T_\delta \equiv \{\delta_u(t_i) \mid \mathsf{send}\ t_1, \ldots, t_n \text{ occurs in } T, 1 \leqslant i \leqslant n\}$$
$$\widehat{TI}^T_\delta \equiv \{(\delta(x), \delta_u(x)) \mid x \in fv(T) \setminus fresh(T)\}$$

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

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 describe a number of improvements that are essential to an efficient check. Since we prove the checks correct in Isabelle there is no risk of affecting the correctness of the entire approach.

There are two major issues that make the coverage check from the previous section quite inefficient when implemented directly. One concerns the fact that to see if a protocol is covered we need to consider for every received term any message that is in the fixed point closed under term implications and intruder deduction. 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 as a member of the fixed point or as a subterm 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 $1 \leqslant i \leqslant n$.

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

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

This relation 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

It has been observed that many intruder deduction problems can be regarded as local theories, i.e., so that all intermediate terms in the deduction are subterms of the given intruder knowledge or the goal term to construct. This is because the intruder may of course compose a message and then decompose it again, but in most theories that would only yield one of the terms that we started with. Observe that we only support theories where the result of an analysis step is x direct subterms of the term being analyzed. This means that all deduction proofs that contain such a pair of encryption-decryption steps

can be normalized to a simpler proof, eliminating the unnecessary detour. Thus, all decomposition steps are applied to messages composed by an honest agent, and thus all intermediate steps in an intruder deduction are subterms of what the intruder wants to construct or knows in advance.[12]

The proof normalization argument we just sketched would be very difficult to integrate in Isabelle directly, since we would have to meta-reason in Isabelle about the definition of the intruder deduction rules. In fact, we have a similar way when reasoning about homogeneous message deduction in compositional proofs (i.e., that the intruder never needs to mix messages from the protocols being composed). Our Isabelle proofs work by induction over the size of terms (not over the structure of the deduction) that everything deducible is also deducible with a more restricted notion, in this case that it is sufficient to only compose terms if the intruder knowledge is already analyzed. We note that such arguments do not hold in general when algebraic properties are considered (like commutativity of exponents in Diffie-Hellman). For that case, our theory would need non-trivial adaptions.

The idea is thus 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 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 Section 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 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:

[12] We only mention here that this is indeed close to the concept of a local theory: it is sufficient to consider in a deduction $M \vdash m$ only intermediate terms that are subterms of M or m. However, there are some exceptions to the locality property: in decomposition the necessary key may not be a subterm of M or m. For instance to decrypt $\mathsf{crypt}(K, M)$ one needs $\mathsf{inv}(K)$, which may neither be part of M nor m. (In that case, since inv is private, the deduction step is only possible if $\mathsf{inv}(K)$ already occurs in M.)

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 three "if"-cases 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 $\mathbb{A} \cap \mathit{subterms}(\mathsf{Keys}(t)) = \emptyset$ **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: Suppose for a term $t \in M$ the three if-conditions of the lemma fail, i.e., when we have to check for every $t' \in cl_{TI}(t)$ that t' is analyzed in $cl_{TI}(M)$. If t' can be decrypted and the keys are available, but the result $\mathsf{Result}(t')$ of the decryption cannot be composed in M, then (and only then) we add this result to M. 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.

The improvements described from Section 6 up to here are the *Default* version of our coverage check. We will now explain an alternative check that improves the run time for some protocols. Note that in our implementation we can make use of Isabelle's parallelization framework, executing the default version and the alternative version in parallel, returning the result of whichever check terminates first. By that, we can ensure that the user always benefits from the run time of the fastest check.

6.3. Restricting the Number of Abstraction Substitutions Further

We return to the second issue described in the beginning of this section: 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 set of abstraction substitutions considered to those that satisfy conditions F2 to F4 of Definition 7. We now show how to take into consideration also condition F1 to further restrict that set. This will not be exactly the set of substitutions that satisfy F1 to F4, but rather an over-approximation that can be computed in an efficient way.

F1 essentially says that for each received term t of a considered transaction, its abstraction $\delta(t)$ must be derivable from the fixed point using the $\vdash$ relation. However, we will, based on the previous two sections, restrict ourselves to the $\vdash_c$ relation.

We therefore in this section define functions that together calculate an overapproximation of the set of these δ. These functions can be seen as implementing the $\vdash_c$ relation. We will now in this first part of this section consider two functions based on the Axiom rule of $\vdash_c$ and in the second part of this section we will then consider a function combining the Axiom rule and the Compose rule. We therefore will now introduce two functions match' and match that are based on the Axiom rule. They work by matching terms in receive-steps, possibly containing variables, with ground terms in the fixed point, possibly containing abstract values. The idea is that by inspecting a term t from the receive-steps 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 $\delta(t)$ is to be a term from the fixed point. This is illustrated by the following example:

Example 8. *Consider the term* $t = \mathsf{f}(X, X)$ *from a* receive*-step in a transaction of a protocol and the term* $s = \mathsf{f}(\{\mathsf{valid}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\}, \{\mathsf{revoked}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\})$ *from a given fixed point. Observe that s and t have similar structure, but the first and second argument of s are different, so there is no match between s and t. However, modulo term implication there could be a match, if the two abstract values in s can flow together.*

Consider the following term implication graph:

$$\{\mathsf{valid}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\} \twoheadrightarrow \{\mathsf{revoked}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\} \twoheadrightarrow \{\mathsf{revoked}(\mathsf{a})\}$$

The closure of $\{\mathsf{valid}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\}$ *is the set* $\{\{\mathsf{valid}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\}, \{\mathsf{revoked}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\}, \{\mathsf{revoked}(\mathsf{a})\}\}$ *and the term implication closure of* $\{\mathsf{revoked}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\}$ *is* $\{\{\mathsf{revoked}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\}, \{\mathsf{revoked}(\mathsf{a})\}\}$. *We can see that both considered abstract values share that they actually represent* $\{\mathsf{revoked}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\}$ *and* $\{\mathsf{revoked}(\mathsf{a})\}$. *Thus, t can be made equal to s* modulo term implication *by choosing either to replace all occurrences of X with* $\{\mathsf{revoked}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\}$ *or by choosing to replace all occurrences of X with* $\{\mathsf{revoked}(\mathsf{a})\}$. *Therefore there are two abstraction substitutions* δ *such that* $s \twoheadrightarrow \delta(t)$, *namely* $\delta = [X \mapsto \{\mathsf{revoked}(\mathsf{a}), \mathsf{ring}(\mathsf{a})\}]$ *and* $\delta = [X \mapsto \{\mathsf{revoked}(\mathsf{a})\}]$. □

What we have done in this example is to use three steps to find out if there is a δ such that $\delta(t)$ can be found in $cl_{TI}(FP)$. More generally, step 1 is to see if there is a term $s \in FP$ that matches t, where different occurrences of the same variable may need to be matched with different abstract values (as for X in the above example). Step 2 is to collect for each variable X the set of every abstract value a that works for every occurrence of X simultaneously modulo term implication, i.e., for every abstract value b computed in step 1, $b \twoheadrightarrow a$. Step 3 is to calculate every possible abstraction substitution δ from step 2.

Let us first consider match' that captures the idea of step 1. It takes two terms t and s, where t is a term from a rule (that may have variables) and s a term from the fixed point (that may not have variables). The function match' checks if t and s can be made equal by replacing *occurrences* of variables in t with abstract values. If this is possible it returns a singleton set containing function θ mapping each variable x to a *set* of abstract values, where each member of $\theta(x)$ is a possible replacement for x. If it is not possible it returns the empty set. When trying to match a variable x with an abstract value a, match' returns a singleton set $\{\theta\}$ and θ maps x to $\{a\}$. For a composed term we recursively compute the set of matches for each subterm and then pointwise merge the results, so that for each variable x we get all abstract values it needs to have in some occurrence of x in t in order to allow for the match with s. Note that in the following definition δ and θ map from variables to *sets* of abstract values.

Definition 13. *The function* match' *is defined as follows:*

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

Example 9. *We return to Example 8 and see that* match' *indeed will calculate what the occurrences of X should be replaced by. Here we have that* $\mathsf{match}'(\mathsf{f}(X,X), \mathsf{f}(\{\mathsf{valid(a)},\mathsf{ring(a)}\}, \{\mathsf{revoked(a)},\mathsf{ring(a)}\}) = \{[X \mapsto \{\{\mathsf{valid(a)},\mathsf{ring(a)}\}, \{\mathsf{revoked(a)},\mathsf{ring(a)}\}\}]\}$. *We notice that the set which X is mapped to consists exactly of the two abstract values that we expected to get from Example 8.* □

Having defined a function that implements step 1 we now turn to implementing a function match that combines this with step 2. The function match checks if t and s can be made equal *modulo term implication* by replacing variables in t with abstract values. If this is possible it returns a singleton set containing a function θ mapping each variable x to a set of abstract values, where each member of $\theta(x)$ is a possible replacement for x. If it is not possible it returns the empty set.

The function does this as follows: If match' gave the solution σ, x is a variable, and $a \in \sigma(x)$, i.e., a is one of the abstract values x needs to match in some occurrence, then we compute all b that can be reached from a with term implication. Next, $\sigma'(x)$ is the intersection of these sets (for any $a \in \sigma(x)$). Thus, $b \in \sigma'(x)$ holds iff $a \twoheadrightarrow b$ for every $a \in \sigma(x)$. This ensures that the term s of the given fixed point can be rewritten by term implications so that it matches t in every occurrence of x. For instance in Example 8, the variable X could either be $\{\mathsf{revoked(a)},\mathsf{ring(a)}\}$ or $\{\mathsf{revoked(a)}\}$, to allow a match between s and t modulo term implication. Match also checks that that all free variables in t can actually represent some abstract value, because if not then it is not at all possible to make t equal to s modulo term implication. Finally, every variable that does not occur in t we map to OCC, the set of all abstract values that can occur.

Definition 14. *Let* $\mathsf{FP} = (FP, TI)$ *be a fixed point. Let OCC be the abstract values that occur in* FP*, i.e.* $OCC = \mathbb{A} \cap subterms(cl_{TI}(FP))$*. The function* match *is defined as follows:*

$$\begin{aligned}
&\mathsf{match}(t,s) = \{\theta\} \textit{ if } \sigma \in \mathsf{match}'(t,s),\\
&\qquad\qquad \forall x.\ \sigma'(x) = \textstyle\bigcap_{a\in\sigma(x)}\{b \mid (a,b) \in TI^*\},\\
&\qquad\qquad \forall x \in fv(t).\ \sigma'(x) \neq \emptyset,\\
&\qquad\qquad \forall x.\ \theta(x) = \textbf{if } x \in fv(t) \textbf{ then } \sigma'(x) \textbf{ else } OCC\\
&\mathsf{match}(t,s) = \emptyset \quad \textit{otherwise}
\end{aligned}$$

We additionally lift match *to sets of terms M using the following definition:*

$$\mathsf{match}(t, M) = \{\theta \in \mathsf{match}(t,s) \mid s \in M\}$$

We can finally conclude with step 3: $\mathsf{match}(t,s)$ either is empty (if there is no match) or a single map θ that tells for every variable the possible values under which we can match the terms. Now every abstraction substitution δ that maps each variable x to an element of $\theta(x)$ is a solution:

Lemma 6. *If* $\delta(t) \in cl_{TI}(s)$ *and* $fv(s) = \emptyset$*, there are no abstract values occurring in t, and* δ *is an abstraction substitution, then* $\mathsf{match}(t,s) = \{\theta\}$ *for some* θ *such that for all variables x, if* $x \in fv(t)$ *then* $\delta(x) \in \theta(x)$*, and if* $x \notin fv(t)$ *then* $\theta(x) = OCC$*.*

Example 10. *Assume that the term implication graph is as in Example 8. Consider then the following calculation by the* match *function:*

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

Here the terms $\mathsf{f}(\{\mathsf{revoked(a)}, \mathsf{ring(a)}\}, \{\mathsf{revoked(a)}, \mathsf{ring(a)}\})$ *and* $\mathsf{f}(\{\mathsf{revoked(a)}\}, \{\mathsf{revoked(a)}\})$ *are indeed equal to the term* $\mathsf{f}(\{\mathsf{valid(a)}, \mathsf{ring(a)}\}, \{\mathsf{revoked(a)}, \mathsf{ring(a)}\})$ *modulo term implication. As explained in Example 8 this will lead to two abstraction substitutions that map the term* $\mathsf{f}(X,X)$ *into the term* $\mathsf{f}(\{\mathsf{valid(a)}, \mathsf{ring(a)}\}, \{\mathsf{revoked(a)}, \mathsf{ring(a)}\})$ *modulo term implication, namely* $[X \mapsto \{\mathsf{revoked(a)}, \mathsf{ring(a)}\}]$ *and* $[X \mapsto \{\mathsf{revoked(a)}\}]$*.* □

This concludes our explanation of the two functions that are based on the Axiom rule of the $\vdash_c$ relation: For a term t from a receive-step and a fixed point FP the function match allows us to calculate the abstraction substitutions δ such that $\delta(t) \in cl_{TI}(FP)$, or in other words the abstraction substitutions δ such that $\delta(t)$ can be derived from FP using the Axiom rule.

As promised we will now introduce a function rcvconstrs that combines the Axiom and Compose rules. This function works by matching terms in receive-steps, possibly containing variables, with ground terms derivable from the fixed point. The idea is that by inspecting the terms $t_1, ..., t_n$ from the receive-steps and comparing them to the terms derivable from the fixed point we will be able to see what the variables in $t_1, ..., t_n$ could be instantiated to if all terms $\delta(t_1), ..., \delta(t_n)$ are to be terms derivable from the fixed point. This is illustrated by the following example:

Received term	Matching terms derivable from FP	Replace X with	Replace Y with
$\mathsf{c_{pub}}$	$\mathsf{c_{pub}}$	$\{\mathsf{valid(a)}\}$ or $\emptyset$ or $\{\mathsf{revoked(a)}\}$	$\{\mathsf{valid(a)}\}$ or $\emptyset$ or $\{\mathsf{revoked(a)}\}$
X	$\{\mathsf{valid(a)}\}$ and $\{\mathsf{revoked(a)}\}$	$\{\mathsf{valid(a)}\}$ or $\{\mathsf{revoked(a)}\}$	$\{\mathsf{valid(a)}\}$ or $\emptyset$ or $\{\mathsf{revoked(a)}\}$
$\mathsf{f_{priv2}}(X,Y)$	$\mathsf{f_{priv2}}(\{\mathsf{revoked(a)}\},\{\mathsf{revoked(a)}\})$ and $\mathsf{f_{priv2}}(\{\mathsf{revoked(a)}\},\{\mathsf{valid(a)}\})$	$\{\mathsf{revoked(a)}\}$	$\{\mathsf{valid(a)}\}$ or $\{\mathsf{revoked(a)}\}$
$\mathsf{f_{pub2}}(\mathsf{f_{priv1}}(X),Y)$	$\mathsf{f_{pub2}}(\mathsf{f_{priv1}}(\{\mathsf{revoked(a)}\}),\{\mathsf{revoked(a)}\})$ and $\mathsf{f_{pub2}}(\mathsf{f_{priv1}}(\emptyset),\{\mathsf{revoked(a)}\})$ and $\mathsf{f_{pub2}}(\mathsf{f_{priv1}}(\{\mathsf{revoked(a)}\}),\{\mathsf{valid(a)}\})$ and $\mathsf{f_{pub2}}(\mathsf{f_{priv1}}(\emptyset),\{\mathsf{valid(a)}\})$	$\{\mathsf{revoked(a)}\}$ or $\emptyset$	$\{\mathsf{valid(a)}\}$ or $\{\mathsf{revoked(a)}\}$

Table 1

Matching received terms with terms derivable from the fixed point.

Example 11. *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:

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

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* $\Vdash_c$. *We will now investigate whether each of the received terms matches a term derivable from the fixed point and additionally determine for the variables X and Y the set of possible abstract values. The first step is to compute* $\mathsf{match}(t,s)$ *for every term t in the received messages and every term s in the fixed point. Additionally, if* $t = f(t_1,\ldots,t_n)$ *for a public* f *like* $\mathsf{f_{pub1}}$, *then we recursively check each* $\mathsf{match}(t_i,s)$ *(for every s in the fixed point) since the intruder can then apply* f *to the respective instances of* t_i *to generate an instance of t that matches. We summarize this in Table 1.*

Consider the row for $\mathsf{c_{pub}}$. *The term* $\mathsf{c_{pub}}$ *is immediately derivable using the Comp rule and this is the case with any replacement of X or Y in the term* $\mathsf{c_{pub}}$ *because these variables do not occur in* $\mathsf{c_{pub}}$. *Consider the row for X. For this to be a derivable abstract value it needs to be some abstract value in FP, i.e. either* $\{\mathsf{valid(a)}\}$ *or* $\{\mathsf{revoked(a)}\}$. *Consider the row for* $\mathsf{f_{priv2}}(X,Y)$. *Since* $\mathsf{f_{priv2}}$ *is private we cannot derive it by composition and so the only option is to use the two matching terms in the fixed point. Consider the row for* $\mathsf{f_{pub2}}(\mathsf{f_{priv1}}(X),Y)$. *No term in FP matches it, but two matching terms can be constructed from the fixed point using the Comp rule.*

Let us now construct the abstraction substitutions that map the variables in such a way that all the received terms are terms derivable from the fixed point. For X we thus need to pick the abstract values that appear in all cells in the "Replace X with" column. In this case there is only one such abstract value, namely $\{\mathsf{revoked(a)}\}$. *For Y we can pick* $\{\mathsf{valid(a)}\}$ *or* $\{\mathsf{revoked(a)}\}$. *This leads to two possible abstraction substitutions:* $[X \mapsto \{\mathsf{revoked(a)}\}, Y \mapsto \{\mathsf{revoked(a)}\}]$ *and* $[X \mapsto \{\mathsf{revoked(a)}\}, Y \mapsto \{\mathsf{valid(a)}\}]$.

□

Example 12. *Consider a transaction receiving the set of terms* $\{\mathsf{f}_{\mathsf{priv2}}(X,Y)\}$ *where* $\mathsf{f}_{\mathsf{priv2}}$ *is a private function. Consider the fixed point* $FP = \{\mathsf{f}_{\mathsf{priv2}}(\{\mathsf{valid(a)}\},\{\mathsf{revoked(a)}\}), \mathsf{f}_{\mathsf{priv2}}(\{\mathsf{revoked(a)}\},\{\mathsf{valid(a)}\})\}$ *and* $TI = \emptyset$*. Assume also for the sake of simplicity that there are no analysis rules. We apply the approach from Example 11. Thus, both X and Y can be* $\{\mathsf{revoked(a)}\}$ *or* $\{\mathsf{valid(a)}\}$*, and thus we get four possible abstraction substitutions, 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 will tolerate this overapproximation.* □

The function rcvconstrs implements the idea from Examples 11 and 12. The function $\mathsf{rcvconstrs}_x(M)$ computes the set of abstract values that the variable x can take so that the intruder can produce all terms in the set M. This set is empty in case the intruder cannot produce all the terms in M (for any choice of x). We will use this for a transaction T that has the terms M as receive steps and x is any free variable in T.

Definition 15. *Let* $M \neq \emptyset$ *be a set of messages, x be a variable, and* $\mathsf{FP} = (FP, TI)$ *be an analyzed fixed point. We define* $\mathsf{rcvconstrs}_x(M)$ *recursively as follows:*

$$
\begin{array}{rll}
\mathsf{rcvconstrs}_x(\{y\}) = \{b \mid \exists a \in FP \cap \mathbb{A}.\ (a,b) \in TI^*\} & \text{if } x = y \\
\mathsf{rcvconstrs}_x(\{y\}) = OCC & \text{if } x \neq y \\
\mathsf{rcvconstrs}_x(\{c\}) = \emptyset & \text{if } c \in \mathcal{C}_{priv} \text{ and } c \notin FP \\
\mathsf{rcvconstrs}_x(\{c\}) = OCC & \text{if } c \in \mathcal{C}_{pub} \text{ or } c \in FP \\
\mathsf{rcvconstrs}_x(\{f(t_1,\ldots,t_n)\}) = A & \text{if } f \in \Sigma^n_{priv}, n > 0, \\
 & \Delta = \mathsf{match}(f(t_1,\ldots,t_n), FP) \\
 & A = \bigcup_{\delta \in \Delta} \delta(x), \\
\mathsf{rcvconstrs}_x(\{f(t_1,\ldots,t_n)\}) = A_1 \cup A_2 & \text{if } f \in \Sigma^n_{pub}, n > 0, \\
 & \Delta = \mathsf{match}(f(t_1,\ldots,t_n), FP) \\
 & A_1 = \bigcup_{\delta \in \Delta} \delta(x), \\
 & A_2 = \mathsf{rcvconstrs}_x(\{t_1,\ldots,t_n\}), \\
\mathsf{rcvconstrs}_x(\{t_1,\ldots,t_n\}) = \bigcap_{i \in \{1,\ldots,n\}} \mathsf{rcvconstrs}_x(\{t_i\}) & \text{if } n > 1
\end{array}
$$

The definition of $\mathsf{rcvconstrs}_x(M)$ expresses how the requirement that the intruder has to produce the messages in M constrains the choice of abstract values of x. The first equation says that if $y = x$ is the message the intruder has to produce, we can choose for x only abstract values that are in the fixed point or reachable via term implication. The second equation is the case that the intruder has to produce any other variable $x \neq y$. In this case this gives no constraint on x, and we have thus OCC. The third equation says that the intruder cannot produce a private constant that is not in the fixed point. The fourth equation says that the intruder can always produce all public constants and all constants in the fixed point, again not constraining the choice of x. The fifth equation says that if the intruder has to produce a term $f(t_1,\ldots,t_n)$ that is composed with a private function then we use the match function to determine all the mappings Δ that map variables to set of abstract values under which said term is in the fixed point modulo term implication. We then return the union A of all abstract values that any of these mappings allows for x. The sixth equation is similar, but here f is public, so we additionally consider the case that the intruder can produce the subterms $t_1,\ldots,t_n$ (and apply f to them), so we have the union of the

values A_1 that give a direct match in the fixed point with the values A_2 under which the subterms can be produced. The final equation lifts $\mathsf{rcvconstrs}_x$ to sets of terms: to produce all the terms, the intruder must produce every single one of them.

In our implementation we have one small optimization not presented above to avoid cluttering the presentation here. Consider again the sixth equation. If one of the t_i is a private constant not in FP then we can "short circuit" the calculation of $\mathsf{rcvconstrs}_x(\{t_1, ..., t_n\})$ because we then know that it will be equal to $\emptyset$.

In Example 11 we saw how calculating the possible replacements for each variable related to the abstraction substitutions that instantiate the received terms to derivable terms. We express this idea in the following lemma that captures the idea of 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 $FP \Vdash_c^{TI} \delta(\{t_1, \ldots, t_n\})$ then $\delta(x) \in \mathsf{rcvconstrs}_x(\{t_1, \ldots, t_n\})$.*

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 x will indeed be one of the abstract values that $\mathsf{rcvconstrs}_x$ calculates.

The following examples illustrate the idea:

Example 13. *We here return to Example 11 and now apply the* $\mathsf{rcvconstrs}_X$ *and* $\mathsf{rcvconstrs}_Y$ *functions:*

$$\mathsf{rcvconstrs}_X(\{\mathsf{c_{pub}}, X, \mathsf{f_{priv2}}(X,Y), \mathsf{f_{pub2}}(\mathsf{f_{priv1}}(X), Y)\}) = \{\{\mathsf{revoked(a)}\}\}$$
$$\mathsf{rcvconstrs}_Y(\{\mathsf{c_{pub}}, X, \mathsf{f_{priv2}}(X,Y), \mathsf{f_{pub2}}(\mathsf{f_{priv1}}(X), Y)\}) = \{\{\mathsf{valid(a)}\}, \{\mathsf{revoked(a)}\}\}$$

We see that this corresponds to the choices for X and Y that we saw in Example 11. As in that example this leads to two possible abstraction substitutions, namely $[X \mapsto \{\mathsf{revoked(a)}\}, Y \mapsto \{\mathsf{valid(a)}\}]$ *and* $[X \mapsto \{\mathsf{revoked(a)}\}, Y \mapsto \{\mathsf{revoked(a)}\}]$. □

Example 14. *We here return to Example 12 and now apply the* $\mathsf{rcvconstrs}_X$ *and* $\mathsf{rcvconstrs}_Y$ *functions:*

$$\mathsf{rcvconstrs}_X(\{\mathsf{f_{priv2}}(X,Y)\}) = \{\{\mathsf{revoked(a)}\}, \{\mathsf{valid(a)}\}\}$$
$$\mathsf{rcvconstrs}_Y(\{\mathsf{f_{priv2}}(X,Y)\}) = \{\{\mathsf{revoked(a)}\}, \{\mathsf{valid(a)}\}\}$$

We see that this corresponds to the choices for X and Y that we saw in Example 12. As in that example this leads to four possible abstraction substitutions, 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* $[X \mapsto \{\mathsf{valid(a)}\}, Y \mapsto \{\mathsf{valid(a)}\}]$. *We see again here that an overapproximation is happening because actually only the first two abstraction substitutions will generate terms represented by the fixed point.* □

We combine the techniques described from Section 6 up to here into an implementation of the coverage check that we denote the *Receive* coverage check.

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. Proving such theorems means that we do not need to be suspicious of implementation bugs even if a strategy gives significant

						Verification						
	Initialization		Fixed Point			Safe Check		NBE Check		Eval Check		
Protocol	Transl.	Setup	Comp.	|FP|	|TI|	Default	Receive	Default	Receive	Default	Receive	Heuristic
KS_2_1	00:00:03	00:00:27	00:00:03	13	28	00:00:28	00:01:07	00:00:19	00:00:15	00:00:14	00:00:12	00:00:46
KS_3_1	00:00:04	00:00:26	00:00:03	17	41	00:00:40	00:03:02	00:00:21	00:00:18	00:00:16	00:00:12	00:01:01
KS_4_1	00:00:04	00:00:26	00:00:03	21	54	00:01:29	00:07:58	00:00:20	00:00:26	00:00:14	00:00:13	00:01:52
KS2_2_1	00:00:04	00:00:27	00:00:03	11	5	00:00:30	00:00:29	00:00:17	00:00:15	00:00:13	00:00:13	00:00:46
KS2_3_1	00:00:04	00:00:28	00:00:04	14	7	00:00:43	00:00:40	00:00:19	00:00:16	00:00:16	00:00:13	00:01:00
KS2_4_1	00:00:04	00:00:28	00:00:03	17	9	00:00:55	00:01:05	00:00:20	00:00:18	00:00:15	00:00:15	00:01:14
KS_Comp_2_1	00:00:05	00:00:27	00:00:04	22	108	00:07:35	00:47:07	00:00:50	00:01:52	00:00:16	00:00:14	00:07:49
KS_Comp_3_1	00:00:05	00:00:27	00:00:06	29	156	00:33:57	03:16:32	00:02:13	00:06:13	00:00:19	00:00:14	00:33:21
KS_Comp_4_1	00:00:05	00:00:25	00:00:10	36	204	01:44:22	10:18:07	00:06:08	00:16:19	00:00:26	00:00:16	01:45:54
NSLclassic	00:00:03	00:00:26	00:00:03	35	6	00:03:15	00:03:06	00:00:20	00:00:18	00:00:32	00:00:14	00:03:28
NSPKclassic	00:00:04	00:00:28	00:00:03	16	6	attack	attack	attack	attack	attack	attack	attack
PKCS#11_3	00:00:06	00:00:36	00:00:08	8	4	attack	attack	attack	attack	attack	attack	attack
PKCS#11_7	00:00:06	00:00:37	00:00:26	13	30	attack	attack	attack	attack	attack	attack	attack
PKCS#11_9	00:00:05	00:00:37	00:00:12	40	20	attack	attack	attack	attack	attack	attack	attack
TLS12_auth_simp	00:00:08	00:00:34	00:00:07	40	20	--:--:--	04:23:08	01:31:14	00:03:05	00:00:38	00:00:30	04:30:37
Logos	00:00:07	00:00:37	00:00:14	48	31	--:--:--	11:01:55	01:38:56	00:36:29	00:01:23	00:01:07	11:13:23
TLS12_SSO	00:00:13	00:00:26	00:15:25	359	35	--:--:--	--:--:--	02:58:21	04:49:35	00:00:37	00:00:23	--:--:--

Table 2

Runtime Measurements (Time Format: *hh:mm:ss*) Experiments that took longer than 12 hours are marked with "--:--:--".

improvements on some examples. 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

In this section, we discuss empirical results. We start by introducing our main benchmark suite for individual protocols, and discuss the performance of our tool, PSPSP, on this benchmark suite. Finally, we discuss an example of a composed protocol.

7.1. Our Benchmark Suite

We evaluated our tool, PSPSP, using the following protocols (see Table 2): The first group of protocols are based on a keyserver. The example KS_*h*_*d* is our keyserver running example for *h* honest agents and *d* dishonest agents.[13] The example KS_Comp_*h*_*d* with *h* honest agents and *d* dishonest agents is inspired by [12] where another keyserver protocol—named KS2_*h*_*d* here—runs in parallel on the same network and where databases are shared between the protocols.

The next group are NSLclassic and NSPKclassic, which are based on the NSL and Needham-Schroeder protocol specifications shipped with AIF-ω [15].

[13] 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.

Then, we have several scenarios of the "PKCS#11" model that is distributed with AIF-ω [15]. Scenario 3 and 7 (PKCS#11_3and PKCS#11_7) 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. There is a third scenario (PKCS#11_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.

With TLS12_auth_simp we have looked at one protocol that has been inspired by a practical protocol, i.e., TLS 1.2. For our experiments, we simplified the protocol by modeling only one variant of the flow and also simplifying the hashing.

Finally, we modeled a protocol developed by the Danish company Logos and a single sign-on protocol (TLS12_SSO) that composes the TLS key exchange with a custom SSO protocol. We will discuss TLS12_SSO in Section 7.3 and the industrial case study (Logos) in Section 9.

7.2. Evaluation of our Benchmark Suite

Table 2 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. All experiments have been conducted on a shared Linux server with an Intel Xeon E5-2640 CPU and 96GB main memory using PSPSP based on Isabelle 2024. PSPSP provides an option to measure the elapsed time (wall-clock) required for executing individual "top-level" commands (e.g., `protocol_security_proof`, see Section 8). 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.

First, we report the time for translating[14] 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 next six columns, we report the run-time of four different strategies for the security proof that we will explain in the following. The last column reports the run-time of a heuristic aims for selecting the fasted strategy automatically. We will explain the details of this heuristic later.

In the *safe* configuration, all proof steps are checked by Isabelle's LCF-style kernel; internally, this makes use of the simplifier configuration called *code-simp* of Isabelle's simplifier. *NBE* employs normalization by evaluation, a technique that uses a partially symbolic evaluation approach that, to a limited extent, relies on Isabelle's code generator. Finally, *eval* is an approach that directly employs the code generator and internally uses the proof method *eval*. In general, the configurations NBE and eval require the user to trust the code generator. From a correctness perspective, the main difference is that the configuration of the code generator used by NBE is very small and cannot be changed by Isabelle users. Hence, this configuration can be easily audited manually and, moreover, is thoroughly tested. In contrast, eval uses a user-configurable configuration of the code generator and, hence, is hard to audit and any theory our work depends on could, potentially, have modified this configuration. While Isabelle's code generator is thoroughly tested, it is not formally verified. While, at the end, it is up to the user to decide

[14]This includes the time for parsing the trac specification and, more importantly, the generation of the conservative HOL definitions.

which approach to use,[15] we consider NBE to be a trustworthy configuration. Furthermore, a pragmatic approach is to use eval during the development (specification) of protocol and only after no attacks are found, switching to NBE or safe for a higher level of trust.

For the first three strategies, we report the time of our default version of the coverage check (*Default*) which implements the improvements described in Section 6 up to and including Section 6.2. We also report the time of our alternative version of the coverage check (*Receive*) which also implements these improvements in addition to the further restriction of the number of abstraction substitutions described in Section 6.3. There is a fourth strategy, called *Heuristic*, that automatically selects the faster coverage check in the configuration *Safe*. We will explain its details at the end of this section.

For all our examples, verification times, using the eval check (i.e., making full use of Isabelle's code generator) are at most 1 1/2 minutes.[16] 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 (we record runtimes of more than 12 hours as $--:--:--$) 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 most protocols, NBE provides a good middle-ground, bringing the verification times down to under a minute for most examples, and below 40 minutes for all examples using the fastest variant of the coverage check.

Furthermore, the Receive coverage check introduced in Section 6.3 significantly reduces the safe verification time for TLS12_auth_simp and Logos in safe mode from over 12 hours to around 4 1/2 (TLS12_auth_simp) and around 11 hours (Logos). For NBE, it reduces the runtime for both protocols from around 1 1/2 hours to 3 minutes (TLS12_auth_simp) respectively around 40 minutes (Logos). For the configuration fully relying on code-generation (eval), the improvements are minor.

Comparing the runtime of the two coverage checks for all examples shows that the relative runtime behavior of the two coverage checks seems not to depend on the simplification strategy (i.e., eval, NBE, safe). This motivated us to implement a heuristic that, first, runs the security proof for *NBE* for both coverage checks in parallel, measures their runtime, and then runs the *Safe* strategy using the coverage check that, for the given protocol, was faster for NBE. We have chosen NBE as the basis, as the runtimes for eval for both coverage checks are often very similar and, hence, cannot provide a solid base measurement. In theory, this heuristic should show a runtime behavior that roughly is the maximum of the runtime of the two coverage checks for NBE plus the runtime for the faster coverage check in safe configuration. Our measurements (last column in Table 2) do confirm this behavior for our example protocols. This makes our heuristic a reliable way for automatically selecting the faster coverage check.

Finally, as mentioned at the beginning of this section, we report the elapsed time in Table 2 and not the total CPU time. On average, the security proof using the default coverage check can run certain parts in parallel, resulting (in our benchmark suite) on an average speedup of 2.5. The Receive coverage check cannot be parallelized that well (having an average speedup of 1.3).

7.3. An Example of Protocol Composition

Table 3 shows the runtimes for an example of a protocol composition, implementing a TLS-based single sign-on (TLS12_SSO). Note that the format of the table differs from Table 2: we report the

[15] Interested users should consult [29] to understand the software stack that needs to be trusted for each configuration.

[16] We do not report runtimes for protocols with an attack. After the fixed point computation, checking for an attack takes less than a second, as we only need to check for the presence of the attack constant. We will discuss the presentation of attack traces for such protocols in Section 8.

									Verification				
	Initialization		Fixed Point (TLS)			Fixed Point (SSO)			TLS		SSO		
Check	Trans.	Setup	Comp.	|FP|	|TI|	Comp.	|FP|	|TI|	Default	Receive	Default	Receive	Composition
Safe Check	00:00:13	00:00:30	00:00:05	25	16	00:07:57	202	38	00:04:57	00:10:24	--:--:--	--:--:--	--:--:--
NBE Check	00:00:13	00:00:30	00:00:05	25	16	00:07:57	202	38	00:00:41	00:00:46	02:18:27	01:59:03	00:00:12
Eval Check	00:00:13	00:00:30	00:00:05	25	16	00:07:57	202	38	00:00:20	00:00:15	00:00:18	00:00:10	00:00:04

Table 3
Runtime Measurements (Time Format: *hh:mm:ss*) for the composed protocol TLS12_SSO.

runtimes for the three different verification checks (safe, NBE, and eval) 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 runtimes (as in Table 2 for both coverage checks) for the individual verification of the two sub-protocols. The final column reports the runtime 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 than the proof that the composition of the two protocols is secure. For instance, the security proof for SSO takes several hours using the NBE check (the safe check takes more than 12 hours), while the proof that the composition of the two protocols (TLS and SSO) is secure, only takes a few seconds.

More importantly, the total runtime of the fastest NBE configuration is roughly two hours, compared to 3h 15m 25s for directly verifying the monolithic TLS12_SSO protocol model using the default coverage check (cf. Table 2). The compositional verification takes only 2h 8m 41s in total using the default coverage check for the TLS component and the receive coverage check for the SSO component (see Table 3).[17]

Finally, note that the TLS protocol used in this composition case study focuses only on the core of TLS: the key exchange. In contrast, TLS12_auth_simp in Table 2 also includes the password authentication and the transmission of some dummy data. This explains the difference in runtimes (and fixed point sizes) between these two versions of TLS.

8. Isabelle/PSPSP

We implemented our approach on top of the Isabelle framework [30], resulting in a tool called Isabelle/PSPSP [17], which is now part of the Archive of Formal Proofs (AFP).[18] 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 methods tools [31].

[17] Using the receive coverage check for both components increases the total runtime only by 5s.

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

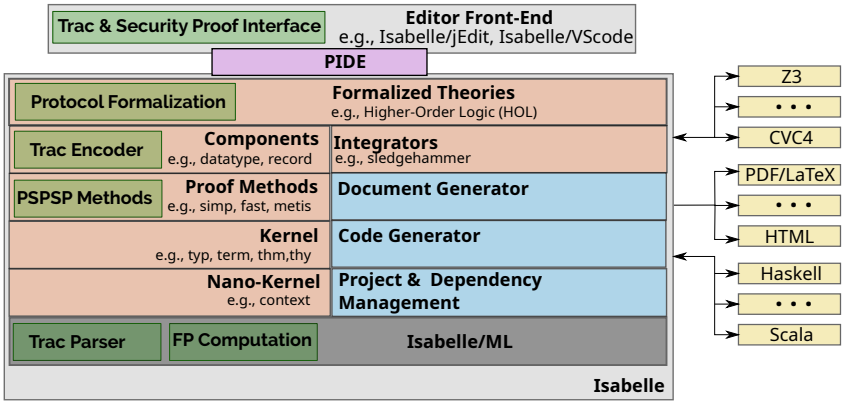

Figure 1. The system architecture of Isabelle and Isabelle/PSPSP.

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 [32], 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 [17]. 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 [33] and the Isabelle/ML interface. Isabelle/ML is Isabelle's programming API that allows one to extend Isabelle using the SML [34] programming language. We use this, in particular, for computing fixed point that builds the backbone 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

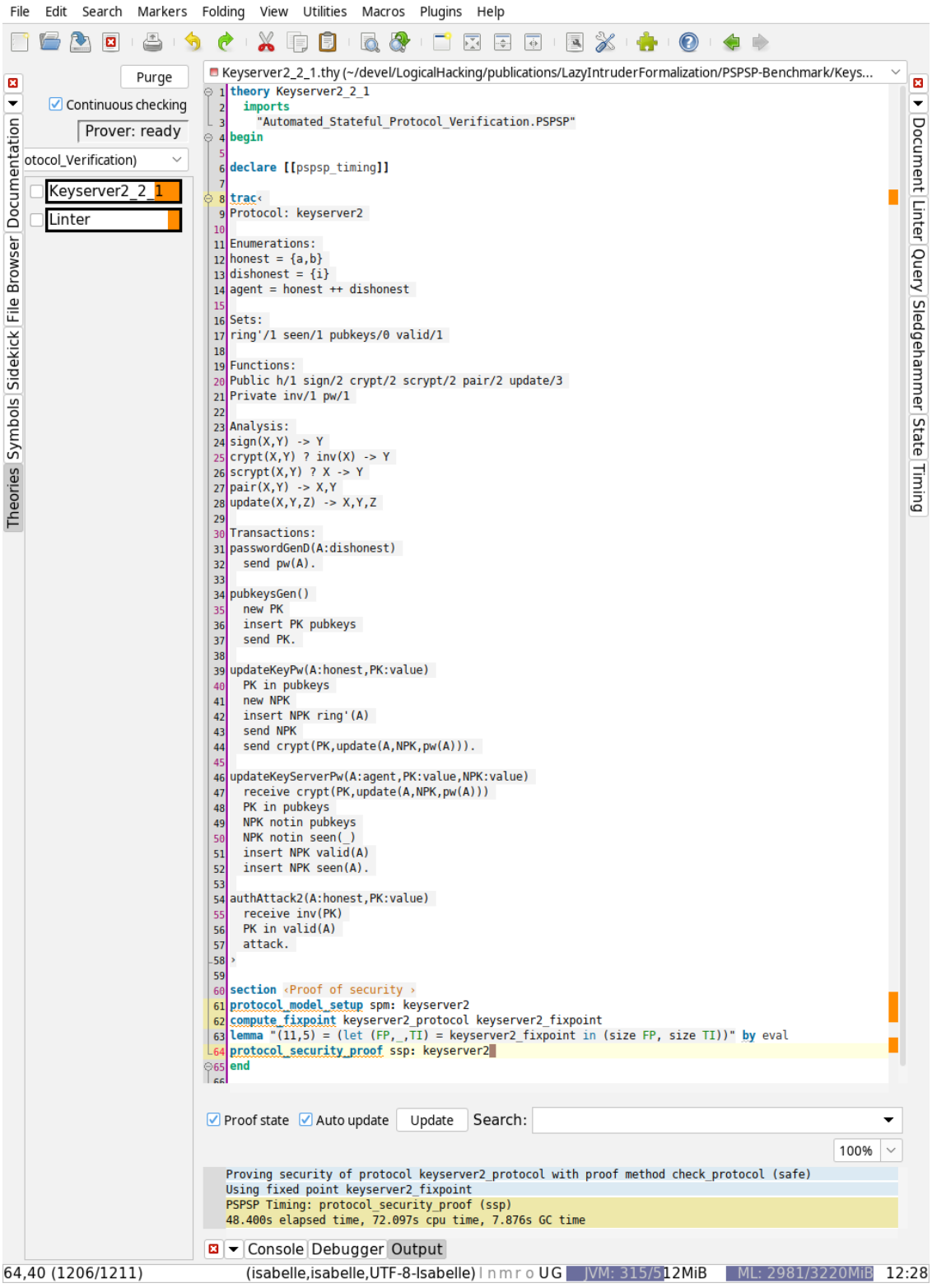

Figure 2. Using Isabelle/PSPSP for verifying a keyserver protocol (KS2_2_1).

(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:

- The protocol is specified using the domain-specific language trac that, e.g., could also be used by a security protocol model checker (line 8–58). 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 61) 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 62, 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 64). The top-level command `protocol_security_proof` proves automatically a theorem showing the security of the defined protocol. This successful proof took ca. 48s (see lower part of the Isabelle/jEdit window.)

Moreover, for the security proofs, we have also manual variants (e.g., `manual_protocol_setup`, `manual_protocol_security_proof`, or `manual_protocol_composition_proof`) of the automated proof commands that only create a proof obligation (i.e., establish the proof state) allowing the user to do an interactive proof utilizing the full power of Isabelle, instead of using the automated proof methods provided by us.[19] This is potentially useful for introducing protocol-specific optimizations or for inspecting what the automated proof methods do.

In case of an insecure protocol, i.e., the fixed point containing the attack constant, Isabelle/PSPSP can support the protocol developer with an attack trace: optionally, while computing the fixed point, the computed traces are bound to a logical constant, which can then be printed in a user-friendly way. Figure 3 shows an example for the protocol PKCS#11_3: the proof that the attack constant is contained in the fixed point (line 95) usually takes only a few seconds. The Isar top-level command `print_attack_trace` (line 98) then prints an attack trace (lower part of the window in Figure 3) showing the steps executed until an attack event is created.

8.3. Compositionality

PSPSP is part of a larger Isabelle infrastructure for security protocols that allows also for compositionality [1], 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.

[19] The commands still allow the user to copy the automated proof script, using "point-and-click" in the same way as the default sledgehammer tool, into the theory file.

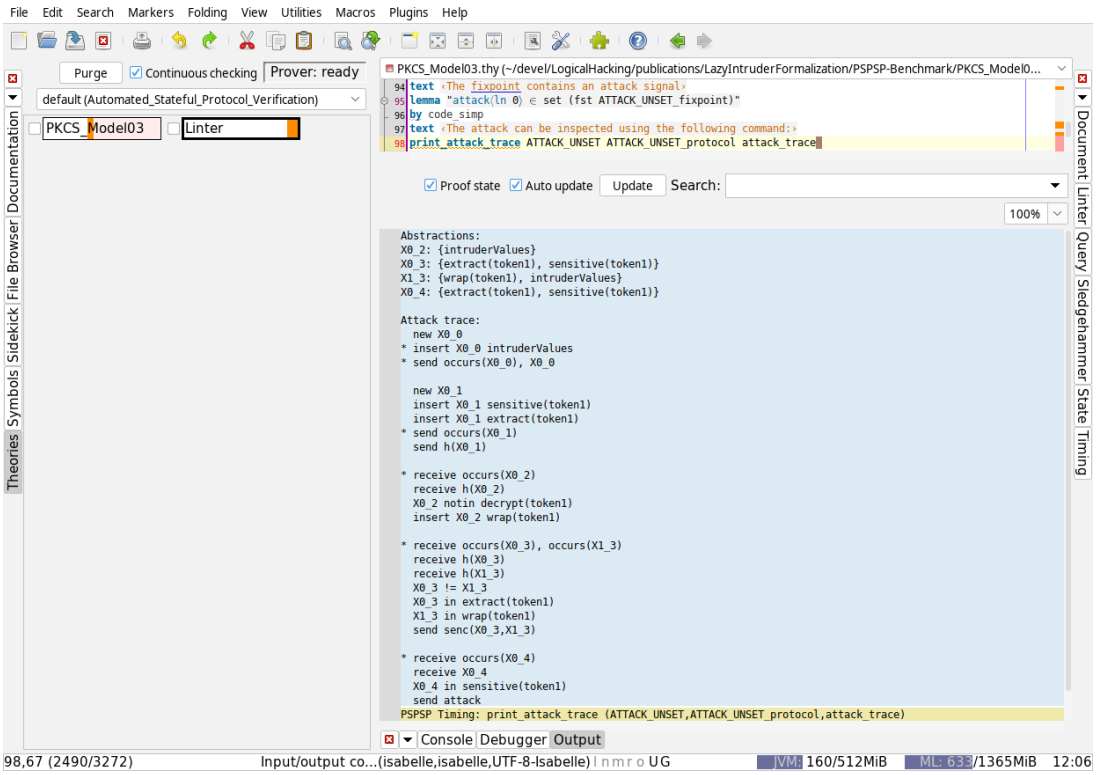

Figure 3. An example attack trace (for the protocol PKCS#11_3).

As an example, we have modeled in [1] a composition of TLS 1.2 and SAML Single-Sign-On (SSO): TLS establishes a secure channel between a client and a server where the client is not yet authenticated. SSO then uses such a channel between a client and the identity provider to first authenticate the client to the identity provider (e.g. using a password); the identity provider then provides a credential for the client that the client can use to authenticate another TLS channel with a relying party. More in detail, the TLS protocol stores 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 reflect 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). The composed protocol is specified in the trac specification language (see Figure 4, until line 299). We compute the sub-message patterns (SMP) common between both protocols (lines 305–309, see [1] for more details) and fixed points (lines 312 and 313) for both protocols "in isolation". These are used, to prove the security of each of the two sub-protocols (lines 315–318). Then we prove the security of the composition: we compute the shared secrets (line 321) and then use an automated proof method for the protocol composition proof (lines 323-237). Finally, we show the security of the composition (lines 329–340) with a simple proof "by auto".

Note that the run time (recall Table 2 and Table 3) for the TLS component of the composition is much smaller than the runtime of TLS12_auth_simp, because it is purely the key-exchange while all

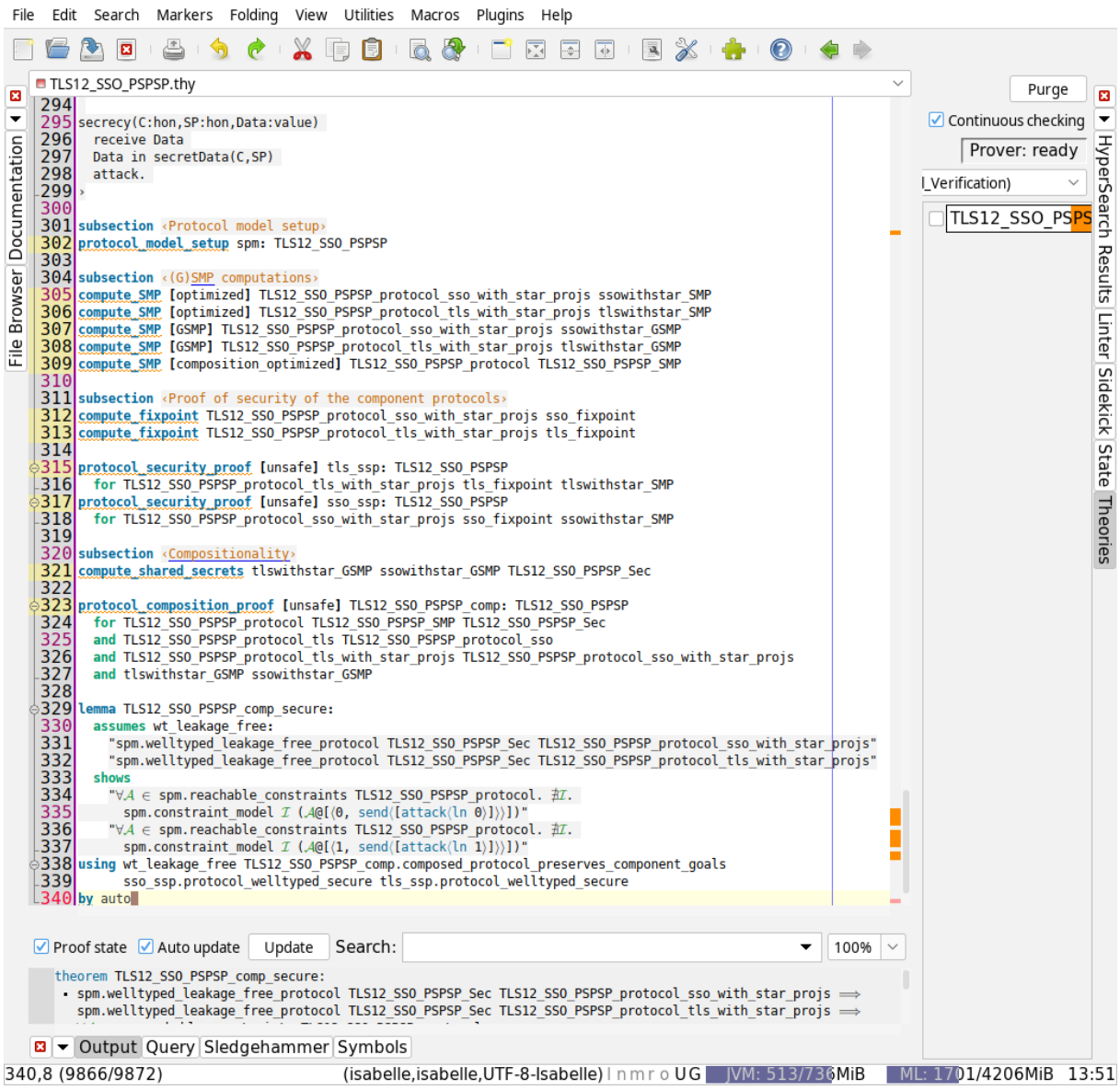

Figure 4. Using Isabelle/PSPSP for verifying the composition of TLS 1.2 and SAML.

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

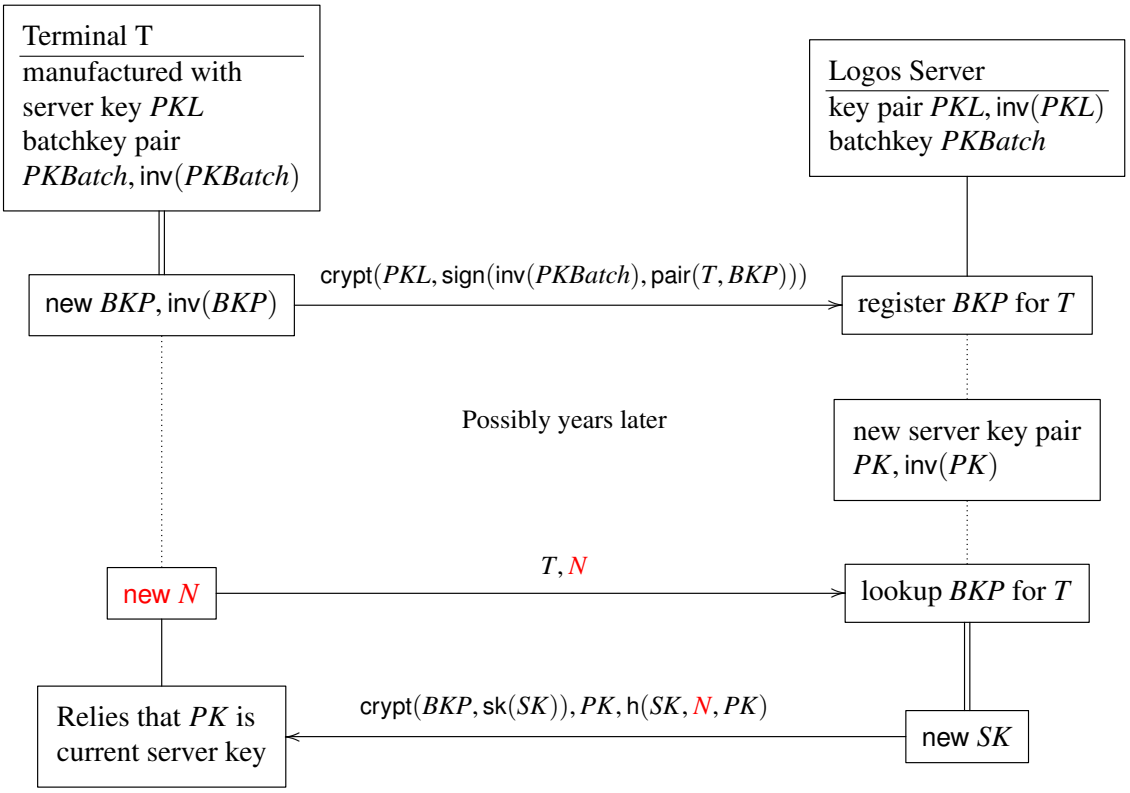

Figure 5. The Logos system in Alice-and-Bob notation.

We had the opportunity to use PSPSP to formally verify a protocol by the Danish company Logos. This protocol stems from 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* want to model the possibility that an intruder obtains old private keys—after all that is the reason for updating them regularly. Such a protocol, even though its messages are fairly simple, can be a challenge for verification tools as it requires mutable long-term states at its core. With some simplifications, we have built a model in PSPSP and found a security flaw. We 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). The protocol is summarized in Figure 5 and we discuss it together with the PSPSP model of every step in the following.

9.1. Epochs

The abstract interpretation approach of PSPSP was originally inspired by ProVerif; both approaches essentially discard the notion of time to arrive at an over-approximation of everything that the intruder can ever know without the normal state explosion and restriction to bounded sessions. That however makes it challenging to model any aspect of time, like injective agreement. However, the only aspect of time that we really need in the model of the Logos protocols is a distinction between past and present events, e.g. distinguish nonces that were generated a long time ago from nonces that are fresh. More precisely, we divide the timeline into two *epochs* $\mathsf{epoch} = \{\mathsf{e1}, \mathsf{e2}\}$. We do not care about the order of events in each epoch: as a notion of time the model only distinguishes between events that happened in e1 and those that happened in e2. In this model all events of e1 happen before all events of e2.

9.2. Key Generation and Revocation

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).

The Logos server has a public-private key pair that is updated regularly.[20] We model this in PSPSP by the following two rules that can be applied in either epoch E:

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

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

Disregarding the epochs briefly, this simply means that the server can at every time point generate a new public-private key pair (publishing the public key) and at any time revoke one. (Like in the keyserver examples before, the knowledge of the corresponding private key inv(PKL) is implicit for honest agents like the Logos server.) In this model, the server may thus have any number of public keys at the same time; this is just an over-approximation. The fact that the set server_keys is parameterized over the epoch E means that no key generated in the first epoch is available in the second and vice versa. Thus, the "cut" between the two epochs coincides with a key update, reflecting our intention that e1 represents events that happened longer in the past while e2 represents recent events.

When a batch of terminals is manufactured, each gets a unique hardware-id in hw_id. Since some sets will be parameterized over this hardware id, we need to give a finite enumeration and we limit ourselves to two hardware-ids $\mathsf{hw\_id} = \{\mathsf{t1}, \mathsf{t2}\}$. During manufacturing, all terminals of a batch have the same public-private key pair, called the batch key, which is registered also with the Logos server, so the terminals can authenticate themselves as legitimate terminals to the Logos server later. We assume here that batch generation happens only in epoch e1, because with e2 we want to model only what happens

[20] In fact, there are several kinds of keys for different purposes; for simplicity, we consider only one kind of key.

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 *PKBatch*
  insert *PKBatch* batch_keys(e1).

batch_keys_revoke(*PK*: value)
  PKBatch in batch_keys(e1)
  delete *PKBatch* batch_keys(e1).

Again, the corresponding private key inv(*PKBatch*) is implicit, since only the manufactured terminals would have it and are assumed to be honest. When manufactured, the terminals also have the current public key of the Logos server. This is the starting state for the Logos protocol of Figure 5.

9.3. The Bootstrap Protocol

Each terminal is required to run a bootstrap protocol with the server in order to establish an individual key *BKP*. This must happen within 30 days after manufacture. 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 key pair, consisting of the bootstrap public key *BKP*, and the corresponding private key inv(*BKP*). For now even the public key *BKP* is kept secret between terminal and server. This later allows the server to send messages to the terminal that can only come from the server. This is an unusual way to employ a public key; a more conventional solution would probably be a shared symmetric key instead. However, this is how the protocol works and we have not found a principal security problem with it (other than the attack reported below that is independent of this).

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*))).

This rule corresponds to sending the first message in Figure 5. Here we have modeled that the terminal knows the batch key pair and the public key of the Logos server simply by looking them up in the respective sets. 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*).

This corresponds to the reception of the first message in Figure 5. The server expects a message that is encrypted with its current public key *PKL* (and thus discards the message if decryption with the current private key failed), and then checks that the content is a signature that verifies with (one of) the current batch public keys *PKBatch* (and discard the message if signature verification fails). The signed message is expected to be a pair where the first is interpreted as a hardware-id T and the second as the bootstrap public key *BKP* which the server stores in the keys(T), so that later a secure connection with T can be established using *BKP*. Note again that this bootstrap protocol can only be executed in epoch e1 as demanded by the parameters of the server and batch key sets, but it can happen for arbitrarily many batches and updates of the server keys in e1.

9.4. The Truststore Protocol

The second part of the Logos protocol, in Figure 5 separated by the dotted lines, is the truststore protocol. This protocol may be executed by a terminal after it was many years in storage. This is possible in both epochs. To initiate, the terminal sends 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 the terminal nor the Logos server 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 nonce N is inserted into an epoch-specific set, in order to express the property that the terminal would not accept an answer in epoch e2 with a nonce it has generated in epoch e1.

The server now receives the request, looks up the *BKP* of the claimed T, and constructs a so-called truststore message, i.e., a message with the current public key of the Logos server that the terminal can verify. 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 N) 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).

Here, the unary function sk is just a format to distinguish this message from other kinds of encrypted messages. This is the version for the case that the transaction is executed in e1. Here we actually require that N notin nonce(e2). This does not correspond to a check that the server does, but our model that

everything in epoch e1 happens before e2, and this constraint reflects that N cannot come from the future.

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). Thus we have a variant of the previous rule for e2:

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).

We have also in both rules the insertion of PK into the set witness$(T, \cdot)$. This is just for formulating the authentication goal below.

To conclude the truststore 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.

9.5. Goals

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, and so is 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 PK that was not sent like this from the Logos server (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 did not send in epoch e2:

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 notin witness(T, e2)
- attack.

Together with the noninjauth_server_keys we have a goal close to injective agreement. Standard injective agreement [35] in this case means that, on top of the non-injective agreement, it is an attack if the terminal accepts PK more often than it was sent by the server. In PSPSP we would express this by using another set request(T, E) and when the terminal accepts a PK that is already in the set request(T, E), then it is an attack; otherwise it is added to the set request(T, E). Thus, this means it is an attack if the same PK is accepted more than once. This way of defining the goal is not suitable in this case, because during the validity of a single PK it is allowed to run the truststore protocol several times, and then it is correct that the terminal in each run receives the same value PK. It is in fact a limitation of our abstraction approach that we cannot really define the existence of an injective function between events.

The formulation with epochs solves this: it is an attack, if the terminal accepts a value PK in epoch e2 that was not sent by the server in epoch e2.

9.6. The Attack

Observe that in the protocol in Figure 5, we have highlighted the nonce N in red. This is in fact the fixed version, the original version did not have this nonce. The goal replay_server_keys (without the constraint $N \in$ nonce(e2)) is the one we found violated in the original protocol. The attack to the version without nonce N is the following replay attack: a terminal gets manufactured, runs bootstrap, and sometime later it runs the truststore protocol with the intruder in the middle, who records 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 with the *eval* strategy – see Table 2.

10. Conclusion and Related Work

The research into automated verification of security protocols resulted in a large number of tools (e.g., [5, 7, 36–38]). 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., [2, 3, 18, 22, 39, 40]). 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 [6, 19, 41]. Goubault-Larrecq [41] 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 Rocq [42] from S_∞. Meier [6] developed Scyther-proof [43], an extension to the backward-search used by Scyther [7], which is able to generate proof scripts that can be checked by Isabelle/HOL [44]. Brucker and Mödersheim [19] integrate an external automated tool, OFMC [38], 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 [8], but with an important modification. The method of AIF produces a set of Horn clauses that is then analyzed with ProVerif [5] (or SPASS [45]), and the same holds true also for several similar methods for stateful protocol verification, namely StatVerif [9], Set-π [14], AIF-ω [15] and GSVerif [10]. Note that a set of definite Horn clauses (i.e. Horn clauses with exactly one positive literal) in first-order predicate logic always has a trivial model, because on can simply interpret all predicates as true for all arguments. Thus, if we have modeled an attack by the truth of some predicate attack, then there is trivially a model of the Horn clauses where attack is true. Instead, one wants to check that attack does not hold true in one particular model, namely the *least model* corresponding to the free algebra interpretation of terms and being true for the least number of ground predicates. This least model is uniquely defined in case of Horn clauses. This is achieved in ProVerif (and SPASS) by checking whether the Horn clauses *imply* attack: 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 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 [41], 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 thing one can do to verify the result is to compute it again and compare the results. Therefore [41] 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 Rocq that the Horn clauses are consistent with the negation of the attack predicate.

The limitation of [41] 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 [13] 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 is preserved by 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[21], 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 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. 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., those containing the generated fixed point) is, in fact, slow in Isabelle [19].

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

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

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.[22]

Furthermore, our work shares a lot of conceptual similarities with Tamarin [11] which can also be regarded as a kind of theorem prover. In fact, it was inspired by the mentioned work of Meier [43] that generates Isabelle proofs from the Scyther tool, but Tamarin is not based on Isabelle and has instead a specialized proving environment based on a sound and complete constraint solver. This in principle shares two very nice features of Isabelle: 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 [46]. In contrast, PSPSP is, with respect to attacks on the abstract level, a complete decision procedure, assuming that users provide reasonable analysis rules that do not lead to non-termination. One concession in order to achieve this is indeed the abstraction. It ensures that protocols proved secure on the abstract level are also secure on the concrete level, but it may deem some protocols to be not secure on the abstract level due to a spurious attack. While we consider having this kind of decision procedure a selling point of PSPSP, it is worth remarking that in practice PSPSP may time out. 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 [47, 48]. Also here it might be possible to make a connection to a theorem prover like 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

[22] To avoid trivial security proofs, one may check if all transactions can be executed. While this is a useful technique, it is unfortunately in general not sufficient to avoid trivial security proofs. Consider e.g. a protocol with two variants of a transaction where a participant sends out an encrypted message. In one variant it is sent by the attacker who also adds the plain text to his knowledge and in another it is sent by an honest agent only in the encrypted form. Now say that only the attacker's version is wrongly specified. In this case all rules of the protocol may be executable except for the final attack rule, but the protocol is nonetheless trivially secure because the attacker is sending out a wrong message. Thus checking for the executability of rules is not sufficient in general to discover trivial security proofs, however inspecting the Scyther proofs or the fixed point may save the day.

transactions). We believe it is even more challenging to integrate this kind of reasoning into a theorem prover like Isabelle, but achievable. We like to investigate this as future work as it could give interesting ways for an analyst to interact with the proving process and inject proof ideas.

Acknowledgments

This work was partially supported by the Horizon Europe project TaRDIS (Trustworthy And Resilient Decentralised Intelligence For Edge Systems) and the Industriens Fond project Sb3D (Security by Design in Digital Denmark).

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

References

[1] A. Hess, S. Mödersheim and A. Brucker, Stateful Protocol Composition in Isabelle/HOL, *ACM Transactions on Privacy and Security* **26**(3) (2023), 1–36–. doi:https://doi.org/10.1145/3577020.

[2] L.C. Paulson, The Inductive Approach to Verifying Cryptographic Protocols, *Journal of Computer Security* **6**(1–2) (1998), 85–128.

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

[4] L.C. Paulson, Inductive Analysis of the Internet Protocol TLS, *ACM Transactions on Information and System Security* **2**(3) (1999), 332–351.

[5] B. Blanchet, An Efficient Cryptographic Protocol Verifier Based on Prolog Rules, in: *Computer Security Foundations Workshop*, 2001, pp. 82–96.

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

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

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

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

[10] 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*, 2018, pp. 344–358.

[11] S. Meier, B. Schmidt, C. Cremers and D.A. Basin, The TAMARIN Prover for the Symbolic Analysis of Security Protocols, in: *Computer Aided Verification*, 2013, pp. 696–701.

[12] A. Hess, S. Mödersheim and A. Brucker, Stateful Protocol Composition, in: *European Symposium on Research in Computer Security*, 2018, pp. 427–446.

[13] A.V. Hess and S. Mödersheim, A Typing Result for Stateful Protocols, in: *Computer Security Foundations Symposium*, 2018, pp. 374–388.

[14] A. Bruni, S. Mödersheim, F. Nielson and H.R. Nielson, Set-π: Set Membership π-Calculus, in: *Computer Security Foundations Symposium*, 2015, pp. 185–198.

[15] S. Mödersheim and A. Bruni, AIF-ω: Set-Based Protocol Abstraction with Countable Families, in: *Principles of Security and Trust*, 2016, pp. 233–253.

[16] 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)*, Vol. 1, IEEE, 2021, pp. 143–158. doi:10.1109/CSF51468.2021.00006.

[17] A.V. Hess, S. Mödersheim, A.D. Brucker and A. Schlichtkrull, Automated Stateful Protocol Verification, *Archive of Formal Proofs* (2020), http://isa-afp.org/entries/Automated_Stateful_Protocol_Verification.html, Formal proof development.

[18] G. Bella, F. Massacci and L.C. Paulson, Verifying the SET Purchase Protocols, *Journal of Automated Reasoning* **36**(1–2) (2006), 5–37.

[19] A.D. Brucker and S. Mödersheim, Integrating Automated and Interactive Protocol Verification, in: *Formal Aspects in Security and Trust*, 2009, pp. 248–262.

[20] R. Chrétien, V. Cortier, A. Dallon and S. Delaune, Typing Messages for Free in Security Protocols, *ACM Transactions on Computational Logic* **21**(1) (2020), 1:1–1:52.

[21] 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*, 2015, pp. 209–229.

[22] A.V. Hess and S. Mödersheim, Formalizing and Proving a Typing Result for Security Protocols in Isabelle/HOL, in: *Computer Security Foundations Symposium*, 2017, pp. 451–463.

[23] J. Heather, G. Lowe and S. Schneider, How to Prevent Type Flaw Attacks on Security Protocols, *Journal of Computer Security* **11**(2) (2003), 217–244.

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

[25] S. Delaune and L. Hirschi, A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols, *Journal of Logical and Algebraic Methods in Programming* **87** (2017), 127–144.

[26] H. Comon-Lundh and V. Cortier, Security Properties: Two Agents Are Sufficient, in: *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, P. Degano, ed., Lecture Notes in Computer Science, Vol. 2618, Springer, 2003, pp. 99–113. doi:10.1007/3-540-36575-3_8.

[27] H. Comon-Lundh and V. Cortier, Security properties: two agents are sufficient, *Sci. Comput. Program.* **50**(1–3) (2004), 51–71. doi:10.1016/J.SCICO.2003.12.002. https://doi.org/10.1016/j.scico.2003.12.002.

[28] AVISPA, AVISPA Project: Automated Validation of Internet Security Protocols and Applications project. http://www.avispa-project.org/.

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

[30] L.C. Paulson, T. Nipkow and M. Wenzel, From LCF to Isabelle/HOL, *Formal Aspects Comput.* **31**(6) (2019), 675–698. doi:10.1007/S00165-019-00492-1.

[31] M. Wenzel and B. Wolff, Building Formal Method Tools in the Isabelle/Isar Framework, in: *TPHOLs 2007*, K. Schneider and J. Brandt, eds, 2007, pp. 352–367. doi:10.1007/978-3-540-74591-4_26.

[32] A.V. Hess, S. Mödersheim and A.D. Brucker, Stateful Protocol Composition and Typing, *Archive of Formal Proofs* (2020), https://isa-afp.org/entries/Stateful_Protocol_Composition_and_Typing.html, Formal proof development.

[33] D. Matichuk, T. Murray and M. Wenzel, Eisbach: A Proof Method Language for Isabelle, *Journal of Automated Reasoning* **56**(3) (2016), 261–282. doi:10.1007/s10817-015-9360-2.

[34] L.C. Paulson, *ML for the working programmer (2nd ed.)*, Cambridge University Press, USA, 1996. ISBN 052156543X.

[35] G. Lowe, A Hierarchy of Authentication Specifications, in: *CSFW*, 1997, pp. 31–44.

[36] 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*, 2004, pp. 1–11.

[37] Y. Chevalier and L. Vigneron, Automated Unbounded Verification of Security Protocols, in: *Computer Aided Verification*, 2002, pp. 325–337.

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

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

[40] G. Bella, D. Butin and D. Gray, Holistic analysis of mix protocols, in: *Information Assurance and Security*, 2011, pp. 338–343.

[41] J. Goubault-Larrecq, Towards Producing Formally Checkable Security Proofs, Automatically, in: *Computer Security Foundations Symposium*, 2008, pp. 224–238.

[42] 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, p. 500.

[43] 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*, 2010, pp. 231–245.

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

[45] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda and P. Wischnewski, SPASS Version 3.5, in: *Conference on Automated Deduction*, 2009, pp. 140–145.

[46] 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) (2022), 573–598. doi:10.3233/JCS-210053.

[47] 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*, 2007, pp. 523–537.

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

Appendix. 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).