2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-6512-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/EuroSP57164.2023.00016

Automatic verification of transparency protocols

Vincent Cheval
INRIA Paris
France
vincent.cheval@inria.fr

José Moreira
Valory AG
Switzerland
jose.moreira.sanchez@valory.xyz

Mark Ryan
University of Birmingham
United Kingdom
m.d.ryan@cs.bham.ac.uk

***Abstract*—Transparency protocols are protocols whose actions can be publicly monitored by observers (such observers may include regulators, rights advocacy groups, or the general public). The observed actions are typically usages of private keys such as decryptions, and signings. Examples of transparency protocols include certificate transparency, cryptocurrency, transparent decryption, and electronic voting. These protocols usually pose a challenge for automatic verification, because they involve sophisticated data types that have strong properties, such as Merkle trees, that allow compact proofs of data presence and tree extension.**

We address this challenge by introducing new features in ProVerif, and a methodology for using them. With our methodology, it is possible to describe the data type quite abstractly, using ProVerif axioms, and prove the correctness of the protocol using those axioms as assumptions. Then, in separate steps, one can define one or more concrete implementations of the data type, and again use ProVerif to show that the implementations satisfy the assumptions that were coded as axioms. This helps make compositional proofs, splitting the proof burden into several manageable pieces. We illustrate the methodology and features by providing the first formal verification of the transparent decryption and certificate transparency protocols with a precise modelling of the Merkle tree data structure.

***Index Terms*—protocols, transparency protocols, automatic verification, ProVerif, symbolic model**

1. Introduction

Many security protocols assume the availability of trusted third parties, such as cloud computing operators. These are entities which are inherently trusted by definition, and whose corruption or misbehavior might be undetected and have catastrophic consequences from a security standpoint. With the advent of web3 and distributed ledger technologies, there is an ever growing interest in the concepts of transparency and accountability of misbehaviour in security protocols: whereas we cannot prevent a malicious action from happening, we can track actions and take the appropriate corrective measures on the entities that have broken the presumed trust assumptions.

The focus of this paper is on transparency. In general terms, we say that a protocol is a *transparency protocol* if (some of) its actions can be publicly monitored by a collection of observers, which can, at a later stage, provide evidence that certain actions occurred. By technical measures to ensure that the details of such actions are made available to relevant parties (whether the general public, regulators, rights advocacy groups or individuals), the protocol reduces the amount of trust required of trusted third parties, and deters malicious actions.

To motivate our work, we consider two prominent use cases of transparency protocols. On one hand, we consider the Google initiative on *certificate transparency* [1] for monitoring and autiting digital certificate issuance which has become an IETF standard [2]. It enables the detection of illegitimate certificates that have been produced either erroneously or deliberately. On the other hand, we also consider *transparent decryption* [3], a protocol that ensures visibility of decryption requests, and has applications in a variety of areas such as surveillance, data sharing and location-based services, among many others.

Appropriate transparency conditions for a protocol can be defined using an append-only ledger, or a blockchain. We assume that some party is willing to maintain the ledger, and we demonstrate how this can be organised in such a way that the maintainer can produce data demonstrating that it is, indeed, maintaining the log correctly. The ledger contents are intended to be accessible by any party to whom transparency is being offered. Therefore, any such party can determine that the ledger is running correctly (or that it is not running correctly, or that their access to it has been denied).

Automatic verification. Our paper concerns how to verify systems for transparency protocols. That is, how to prove that the system really does guarantee that the actions of monitored parties are made available to observers.

Several tools have been proposed for automated analysis of security protocols. Some of these tools impose restrictions on the protocols in order to achieve termination of the analysis. For example, the tools may assume a bounded number of sessions, like Avispa [4], DeepSec [5], or Akiss [6]. These tools are efficient at finding attacks on small protocols but quickly face state explosion for complex protocols. Hence for large and complex protocols, tools like Tamarin [7] and ProVerif [8] are often preferred. They both offer a flexible framework to model a protocol and its primitives, as well as their security properties. One key feature of Tamarin is that it offers an interactive mode when the tool fails to prove a protocol, while ProVerif typically offers more automation.

We work within the framework of ProVerif. It supports cryptographic primitives including symmetric and asymmetric encryption; digital signatures; hash functions;

© 2023, Vincent Cheval. Under license to IEEE.
DOI 10.1109/EuroSP57164.2023.00016

bit-commitment; and signature proofs of knowledge. The tool is capable of evaluating secrecy properties, authentication properties, and indistinguishability properties. In ProVerif, protocol analysis is considered with respect to an unbounded number of sessions and an unbounded message space. The tool is capable of attack reconstruction: when a property cannot be proved, an execution trace which falsifies the desired property can often be constructed.

The transparency ledger may be implemented as a Merkle tree, which means that the ledger maintainer can produce proofs that a data item is in the ledger (*proof of presence*), and that the ledger is only being appended to (*proof of extension*). To model this in ProVerif, we use user-defined predicates whose semantics is defined with Horn clauses. ProVerif can work with arbitrary Horn clauses, but adding them often leads to non termination of the ProVerif resolution strategy. Recent work [9] has extended ProVerif with notions of lemmas and axioms, in an effort to address this non-termination issue. However, lemmas and axioms have strong limitations with user-defined, attacker or message predicates. We address these limitations in this paper.

Our contributions. The paper develops new ProVerif capabilities and a methodology for using them. Our contributions are as follows:

- We introduce new capabilities in ProVerif; more precisely, we define semantics and algorithms that allow ProVerif to work with lemmas and axioms that involve user-defined predicates.
- We prove soundness of the algorithms, and implement them in a new version of ProVerif.
- We introduce a new methodology for ProVerif, in which the proof of a protocol can be given based on assumptions about the behaviour of a data type (we code these assumptions as an *interface*); then, in separate steps we formally prove the assumptions hold for one or more concrete realisations of the data type. This methodology would syntactically not be possible without our extension of ProVerif.
- We model two transparency protocols (transparent decryption and certificate transparency), and successfully instantiate the proposed methodology.

The paper is supported by our new version of ProVerif and the ProVerif scripts for transparent decryption, which can be found in [10]. Our code has been reviewed by the ProVerif owners, and it will be incorporated in the next ProVerif release. Detailed definitions and proofs are provided in [11].

2. Background and related work

2.1. Transparency protocols

Several recent protocols use a publicly-accessible append-only log data structure to achieve a *transparency property*. One of the earliest and most widely deployed protocol in this category is *certificate transparency* [1]. The core idea of certificate transparency is that certificates are accepted by browsers only if they are accompanied by a proof that they are present in an appropriate log. Insisting on certificates being in a publicly-accessible log means that the existence of the certificate is *transparent*. It prevents situations in which corrupted CAs issue rogue certificates without being noticed. This idea has been generalised to define more ambitious public-key infrastructures, such as ARPKI [12] and DTKI [13], which aim to make all the infrastructure parties behave transparently.

The log and its proof data. As mentioned, transparency protocols rely on an append-only log. The log is not assumed to be trustworthy; rather, anyone can verify the data it outputs, and if this verification succeeds, then the transparency property is upheld. The *expected behaviour* of the log may vary from protocol to protocol; here, we give a generic example [14]. The log L is organised as an append-only Merkle tree. For our purposes, a Merkle tree storing data $R_1, R_2, \ldots, R_n$ is a binary tree whose leaves (when considered in left-to-right order) store the data $R_1, R_2, \ldots, R_n$ and whose non-leaf nodes store $H(c_\ell, c_r)$ where H is a hash function and c_ℓ and c_r is the data stored at the left and right child node respectively. The maintainer of L runs three protocols:

- On request, it outputs the current value h stored at L's root (called the *root tree hash* of L).
- On input R, it outputs data which proves that R is present in L (or it outputs $\perp$ if that is not the case). This data consists of data stored in some of the nodes of L, and its size is $O(\log n)$.
- On input h_1 and h_2, it outputs data which proves that L previously had the root tree hash h_1, and subsequently the root tree hash h_2 (or it outputs $\perp$ if this is not the case). This data also consists of data stored in some of the nodes of L, and its size is $O(\log n)$.

The log's behaviour is fully verifiable, and therefore there is no trust assumption on the log maintainer. It can be malicious, and still the security property is upheld.

Verification of transparency protocols. Several previous papers have applied protocol verification techniques to transparency protocols. The papers on ARPKI and DTKI [12], [13] both use the Tamarin prover to prove some security properties. However, both papers make the same huge abstraction: they treat the log as a list. The DTKI paper acknowledges that formalising and modeling the complex data structures of transparency protocols is an unsolved problem. Certificate Transparency was proved using Tamarin in [15] using a simplified setting, for example by modelling the log as a trusted global database shared between agents. Additionally, in that work, proofs of presence and proof of extension were not modeled; instead, they use placeholders in the shared trusted database to act as proofs. In contrast, we model proofs and their data structure precisely.

2.2. Transparent decryption

Transparent decryption is a transparency protocol, aiming to prevent certain decryptions from being performed stealthily; rather, the decryption operation inevitably produces evidence of the fact that the decryption has taken place. This can be used, for example, to support privacy: it can mean that a subject is alerted to the fact that information about them has been decrypted. Among other uses, transparent decryption has been proposed for

accountable execution of search warrants and data interception [3], [16], [17]; data sharing between organisations [18]; in vehicle and IoT data applications [19]. A company has begun building products using these ideas [20]. Transparent decryption is an accountable algorithm in the sense of [21].

In transparent decryption, the decryption key is distributed among a set of agents (called *trustees*); they use their key share only if the required transparency conditions have been satisfied. Typically, the transparency condition can be formulated as the presence of the decryption request in a transparency log [22].

We present a minimal system for transparent decryption below. The system satisfies the basic security property for transparent decryption, which we also detail below.

How it works. More formally, the system works as follows:

- *Subjects* create ciphertexts using a public encryption key ek.
- Shares $dk_1, \ldots, dk_n$ of the decryption key are held by *trustees* $T_1, \ldots, T_n$. For example, this might be a threshold decryption system, so that any m out of n trustees are sufficient to decrypt.
- A *decryption requester* G can request the decryption of a ciphertext. This involves recording the request in a *log* L.
- L is organised as a Merkle tree. This means that the maintainer can issue data that demonstrates it's maintaining L in an append-only fashion (see below).
- *Trustees* are automatic processes which accept ciphertexts and log data as input. The log data attests that certain information has been placed in the log L, and that L has been maintained append-only since it was last seen by the trustee. The trustees verify the log data. If (and only if) the data verifies correctly, a trustee will perform its part in decrypting the relevant ciphertext and output the result.
- Subjects can try to inspect the log contents. If their attempt is successful, they will see from the log whether their ciphertexts have been decrypted or not. If they are not successful (for example, the data is inconsistent or their access is denied) then they should assume that their ciphertexts have been decrypted.

The trust assumption for transparent decryption is that the trustees behave correctly. As mentioned, this means that they perform their part of the decryption if, and only if, the verification of the proofs in the input data is successful.

Security property. We aim to prove the following property:

> Suppose an honest subject encrypts a secret s with ek, and later the secret s becomes known by some other party (e.g., any of the mentioned parties, or an attacker, or anyone else). Suppose the subject successfully accesses the log L and successfully verifies the log data. Then the subject sees the decryption request for s in L.

Trustees and their actions. Trustees are designed to be very simple and to have minimal computational requirements, so that their trustworthiness can be established as straightforwardly as possible. They do not have to store any voluminous data; they store just three data items, and they run two protocols (see Fig. 1).

T_i stores: h, dk_i, sk_i	
• Input: R, h', π, ρ • Compute: – Verify π: R in h' – Verify ρ: h' extends h – result := dec(dk_i, R) – $h := h'$ • Output: result	• Input: v • Compute: r := sign($sk_i, (v, h)$) • Output: r

Figure 1. Protocols run by trustee T_i. The trustee stores the most recent root tree hash h of the log that it has seen, and a decryption key dk_i and a signing key sk_i share. The protocol on the left inputs a request R and some other parameters, and outputs a decrypted result. The protocol on the right inputs a nonce v, and outputs a signature on (v, h).

Trustees can be implemented in a variety of ways. For example, they may be cloud-based software processes run by organisations with a high reputation such as charities and foundations. These organisations can use hardware-based attestation to give further confidence about the binary code trustees are running, and the secure storage and use of their keys. Alternatively, trustees could be implemented on dedicated hardware modules, such as the TPM [23] or Google Titan chip [24], or a RISC-V chip like Open Titan [25], [26].

The system we have described is a minimal one that provides decryption transparency. It could readily be extended to have some additional properties, such as trustee obvliviousness (namely, the inability of a trustee to obtain any information about the decryption request or its result), and proper authentication of the decryption request (see, e.g. [16]).

Applications. Transparent decryption can be applied in many areas to enhance privacy. We give some examples.

- Alex can choose to share her location in encrypted form with some nominated friends and family, called *angels* by the app that implements this idea [27]. No-one except her angels can view her location; and she can monitor whether and when they do so.
- Suppose Alex is being investigated by the police. In an effort to establish her innocence, she may choose to hand over her phone. With transparent decryption, Alex can upload her phone contents in encrypted form. Then Alex gets evidence of what part of this uploaded material is decrypted.
- Alex provides *know-your-customer* (KYC) information to her bank, so that if necessary later, it can carry out anti-money laundering procedures. Recent proposals [28] suggest centralising KYC registers, to make the procedure more efficient. With transparent decryption, the ill effects of such centralisation can be mitigated by making money laundering investigations more transparent.

2.3. ProVerif

ProVerif is a software tool for automated reasoning about the security properties of cryptographic protocols.

It was first released in 2002, and has been continuously developed for the last 20 years. It has been used to analyze hundreds of protocols, including major deployed protocols such as TLS [29], Signal [30], Noise [31], avionic protocols [32], and the Neuchâtel voting protocol [33].

A cryptographic protocol in ProVerif is specified as follows:

- Cryptographic primitives (such as symmetric and asymmetric encryption or digital signatures) are specified typically as *reduction rules*, such as this one for public key encryption and decryption: $\mathsf{decr}(k, \mathsf{encr}(\mathsf{enc_key}(k), m)) = m$.
- The behaviour of the protocol participants is described using the process calculus syntax (see below).
- The properties which are to be checked are specified as queries. ProVerif supports different kinds of properties; in this paper, we restrict our attention to reachability and correspondence properties. For example, the correspondence property $\mathsf{event}(ev(x)) \Rightarrow \mathsf{att}(x)$ says that if the event ev occurs with a parameter value x, then the value x was previously known by the attacker.

Syntax. A simplified syntax for the process calculus *terms*, *expressions*, *events*, *predicates* and *processes* is displayed in Fig. 2. ProVerif's calculus also supports additional constructs, e.g. for tables, phases, extended terms, ..., but we omit them for simplicity as our results can be easily generalized to these constructs.

Terms $M, N, \ldots$ are built over variables, names and application of *constructor* function symbols from a finite set $\mathcal{F}_c$. *Destructor* function symbols, from a finite set $\mathcal{F}_d$, can manipulate terms and must be evaluated in the *assignment* construct. Unlike constructor function symbols, the evaluation of a destructor may *fail*, or in other words, may evaluate to the special constant fail. Typically, in the assignment construct $\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$, the expression D will be evaluated; if its evaluation fails then the process Q will be executed, otherwise the variable will be instantiated by its result and P will be executed. The exact behavior of a destructor function symbol is defined by a list of rewrite rules given by the user (see [9] for the complete definition of the evaluation of an expression). For example,

$$\mathsf{in}(c, y); \mathsf{let}\ x = \mathsf{decr}(k, y)\ \mathsf{in}\ \mathsf{out}(c, x)\ \mathsf{else}\ \mathsf{out}(c, 0)$$

outputs the plaintext m if a ciphertext of the form $\mathsf{encr}(\mathsf{enc_key}(k), m)$ is given as input, otherwise it outputs 0.

A substitution σ is an assignment of terms to some variables; for example, $\{x \mapsto \mathsf{encr}(k, m)\}$ is a substitution. If M is a term, then $M\sigma$ is the term obtained by replacing any x mapped by the substitution with the term that it maps to. For processes P and facts F, applying the substitution to obtain $P\sigma$ and $F\sigma$ is defined similarly (taking care not to substitute bound variables).

The process calculus also contains standard constructs $\mathsf{out}(N, M); P$ (representing the output of a term M on a channel N), $\mathsf{in}(N, x); P$ (the input on channel N of a message which gets bound to a variable x), $\mathsf{new}\ a; P$ (the generation of a fresh name a), $P \mid Q$ (the concurrent execution of processes), $\mathsf{event}(ev(M_1, \ldots, M_k)); P$ (the recording of event execution), and $!P$ (the concurrent execution of an unbounded number of copies of a process).

P, Q ::=	processes
0	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a; P$	restriction
$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	assignment
$\mathsf{let}\ x_1, \ldots, x_n\ \mathsf{suchthat}\ p(M_1, \ldots, M_k)\ \mathsf{in}\ P\ \mathsf{else}\ Q$	predicate evaluation
$\mathsf{event}(ev(M_1, \ldots, M_k)); P$	event

Figure 2. Syntax of the core language of ProVerif.

Less common is the construct for predicate evaluation, that is, $\mathsf{let}\ x_1, \ldots, x_n\ \mathsf{suchthat}\ p(M_1, \ldots, M_k)\ \mathsf{in}\ P\ \mathsf{else}\ Q$. In this construct, the variables $x_1, \ldots, x_n$ must occur in the predicate $p(M_1, \ldots, M_k)$. If $x_1, \ldots, x_n$ can be instantiated, say by a substitution σ, such that $p(M_1, \ldots, M_k)\sigma$ holds then $P\sigma$ is executed; otherwise Q is executed. Note that predicate evaluations are most commonly used with a classical if-then-else conditional, corresponding in fact to the evaluation without variables ($n = 0$).

User-defined predicates and clauses. ProVerif allows users to define predicates, and to give their semantics by means of Horn clauses. This is useful for defining predicates on data types. For example, the list data structure can be represented by a constant nil and a constructor cons. One can define a membership predicate mem using the Horn clauses:

$$\forall x, \ell.\ \mathsf{mem}(x, \mathsf{cons}(x, \ell))$$
$$\forall x, y, \ell.\ \mathsf{mem}(x, \ell) \rightarrow \mathsf{mem}(x, \mathsf{cons}(y, \ell)).$$

As variables in Horn clauses are always universally quantified, we will omit writing the quantifier in the rest of this paper.

Definition 1 (derivation)**.** *A derivation $\mathcal{D}$ of a fact F from a set of clause $\mathbb{C}_{user}$ is a tree whose nodes are labeled by Horn clauses in $\mathbb{C}_{user}$ and edges are labeled by ground facts such that the incoming edge of the root is labeled by F. For all nodes η in $\mathcal{D}$ labelled by a clause $F_1 \wedge \ldots \wedge F_n \wedge \phi \rightarrow C$, there exists a substitution σ such that: (i) the incoming edge of η is labeled by $C\sigma$; (ii) η has n outgoing edges labeled by $F_1\sigma, \ldots, F_n\sigma$ respectively; (iii) $\phi\sigma$ is true.*

The derivability of facts allows us to define the true statements of a predicate p, denoted $\mathsf{sem}(p)$, as the set of facts $F = p(M_1, \ldots, M_n)$ derivable from $\mathbb{C}_{user}$. Note that only user-defined predicates, equalities and disequalities on terms can occur in the clauses from $\mathbb{C}_{user}$. As such, the semantics of a user-defined predicate is independent from any protocol.

Example 1. $\mathsf{mem}(a, \mathsf{cons}(b, \mathsf{cons}(a, \mathrm{nil}))) \in \mathsf{sem}(\mathsf{mem})$ *as it is derivable by the following derivation with $\sigma_1 = \{x \mapsto a; y \mapsto b; \ell \mapsto \mathsf{cons}(a, \mathrm{nil})\}$ and $\sigma_2 = \{x \mapsto a; \ell \mapsto \mathrm{nil}\}$.*

$$\mathsf{mem}(a, \mathsf{cons}(b, \mathsf{cons}(a, \mathrm{nil}))) \downarrow \boxed{\mathsf{mem}(x,\ell) \rightarrow \mathsf{mem}(x, \mathsf{cons}(y,\ell))} \ \textit{with } \sigma_1$$

$$\mathsf{mem}(a, \mathsf{cons}(a, \mathrm{nil})) \downarrow \boxed{\mathsf{mem}(x, \mathsf{cons}(x,\ell))} \ \textit{with } \sigma_2$$

Optionally, a predicate can be declared as a "blocking" predicate, meaning that there are no clauses containing the predicate in the conclusion of the clause. In this case, ProVerif proves properties that hold for *any* definition of the considered blocking predicate.

Semantics of processes. The semantics of processes is defined by the means of a reduction relation $\xrightarrow{\ell}$ between *configurations* which express the current state of the the execution of the processes interacting with the attacker. Formally, a configuration is a triple $\mathcal{E}, \mathcal{P}, \mathcal{A}$ where $\mathcal{E}$ is the set of names used in the configuration, $\mathcal{P}$ is a multiset of processes, and $\mathcal{A}$ is a set of terms representing the knowledge of the attacker.

The full set of rules defining the relation $\xrightarrow{\ell}$ is provided in [11] (and in [34]) and we only show a small extract below. For example, the following rule represents that an event is triggered:

$$\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{event}(ev).P\}\!\}, \mathcal{A} \xrightarrow{\mathsf{event}(ev)} \mathcal{E}, \mathcal{P} \cup \{\!\{P\}\!\}, \mathcal{A}$$

The rule for predicate evaluation (when the predicate evaluates to true) is:

$$\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{let}\ x_1, \dots, x_n\ \mathsf{suchthat}\ pred\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\} \rightarrow \mathcal{E}, \mathcal{P} \cup \{P\sigma\}, \mathcal{A}$$

when $pred = p(M_1, \dots, M_k)$ and there exists a substitution σ such that $\mathrm{dom}(\sigma) = \{x_1, \dots, x_n\}$ and $p(M_1\sigma, \dots, M_k\sigma) \in \mathsf{sem}(p)$.

An *execution trace* of a process P is then defined as a sequence of applications of the relation $\xrightarrow{\ell}$ starting from the initial configuration $\mathcal{C}_1 = (\emptyset, \{\!\{P\}\!\}, \emptyset)$, i.e. $T = \mathcal{C}_1 \xrightarrow{\ell_1} \dots \xrightarrow{\ell_n} \mathcal{C}_{n+1}$.

In addition to user-defined predicates, ProVerif considers several native predicates: $\mathsf{att}(M)$ indicating that the attacker knows M; $\mathsf{mess}(M, N)$ indicating that a message N has been sent on the channel M and $\mathsf{event}(ev)$ indicating that an event ev has been raised. Satisfiability of a fact F by a trace T, denoted $T \vdash F$, is given by the labels and configurations in T (e.g. when $F = \mathsf{event}(ev) = \ell_i$ for some i). Naturally, $T \vdash F$ with $\mathrm{pred}(F) \in \mathcal{F}_p$ when $F \in \mathsf{sem}(p)$.

Finally, a correspondence query $F_1 \wedge \dots \wedge F_n \Rightarrow \psi$ can be seen as the first order logic formula $\Psi = \forall \tilde{x}.(F_1 \wedge \dots \wedge F_n \Rightarrow \exists \tilde{y}.\psi)$ where $\tilde{x} = \mathrm{vars}(F_1, \dots, F_n)$ and $\tilde{y} = \mathrm{vars}(\psi) \setminus \tilde{x}$. The correspondence query holds when for all traces T of P, $T \vdash \Psi$.

Lemmas and axioms. The problem that ProVerif tries to solve is undecidable in general [35]; therefore, by design ProVerif is not complete: it may fail to terminate, and it may yield false attacks. Much work has been done to make it more complete in practice. A significant step in this direction introduces *lemmas and axioms* [9] as a way to guide derivations in ProVerif, and also to deal with some of the abstractions introduced by the tool.

An *axiom* is an instruction to ProVerif to consider some facts as true, even if they cannot be proved by ProVerif from the protocol process. Typically, an axiom is used if one has a separate (perhaps manual) proof of the fact in question. Consider, for example, a smartcard which stores two secrets, s_1 and s_2. It allows the user to choose either one of them to be revealed, but not both. (This might be used in a lottery, for example.) We could model the smartcard with the following process:

$$\mathsf{in}(c, x); (\mathsf{if}\ x = 1\ \mathsf{then}\ \mathsf{out}(c, s_1) \mid \mathsf{if}\ x = 2\ \mathsf{then}\ \mathsf{out}(c, s_2))$$

The user chooses to enter 1 or 2, and obtains the corresponding secret; after that, the smartcard does not accept any further input.

The intended security property is that at most one secret is revealed. Unfortunately, ProVerif is not able to prove the security of this device with the given process description. The reason is that ProVerif introduces an abstraction, which allows it to consider a derivation in which x has sometimes the value 1 and sometimes the value 2, and hence the output of s_1 and s_2 can both occur. This is not a valid trace; it is a false attack introduced by ProVerif's abstraction.

Axioms allow us to rectify this situation. We write the process as follows:

$$\begin{aligned} &\mathsf{new}\ st; \mathsf{in}(c, x); \mathsf{event}(\mathrm{Uniq}(st, x)); \\ &\quad (\mathsf{if}\ x = 1\ \mathsf{then}\ \mathsf{out}(c, s_1) \mid \mathsf{if}\ x = 2\ \mathsf{then}\ \mathsf{out}(c, s_2)) \end{aligned}$$

and we add the axiom

$$\mathrm{Uniq}(st, x_1) \wedge \mathrm{Uniq}(st, x_2) \Rightarrow x_1 = x_2.$$

The axiom asserts that only one value of x is allowed. This axiom is valid, since for a given st there can be only a single input of x. ProVerif is able to use this axiom to prove the security of the device. This kind of axiom, stating that only one value of an input is allowed, is very useful in increasing the precision of ProVerif.

A *lemma* is similar to an axiom as ProVerif uses it in proofs to establish the desired property but it must be able to prove it first (while it does not try to prove axioms). Lemmas are therefore a useful way of decomposing a verification into smaller pieces.

3. A methodology to model protocols with complex data structures

An intuitive, first attempt to model the transparent decryption protocol in ProVerif requires one to define:

- The required equational theories (e.g. public key encryption).
- The predicates and clauses defining the data structure for the log maintainer.
- The predicates and clauses that represent true statements about the proof of presence and extension that the data structure must satisfy.
- The process defining the protocol.
- The security properties that we are interested to test.

Unfortunately, this "monolithic" attempt to prove the security properties does not always work when the protocol has to deal with complex or recursive data structures,

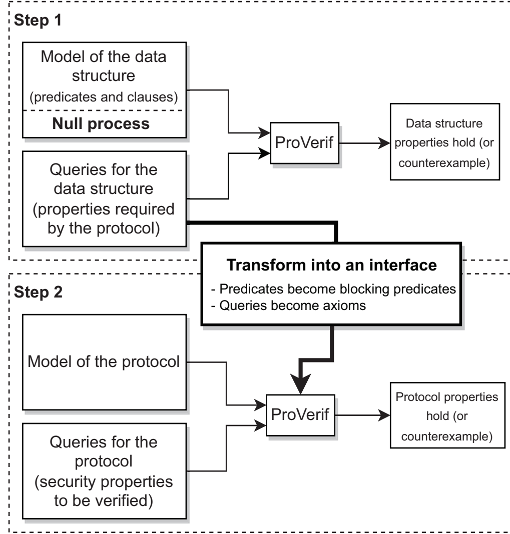


Figure 3. A methodology to prove security in protocols involving complex data structures

and it will often result in ProVerif failing to terminate. In fact, we have encountered this issue even when we try this approach by substituting Merkle trees with a simpler data structure like a hash list.

One way to avoid the monolithic proof is to make an appropriate abstraction of the Merkle tree data structure. This could be achieved by defining the properties the data structure is expected to satisfy, and proving the protocol based on assuming those properties. Later, as a separate step, one can look at whether a particular implementation satisfies the assumed properties.

This observation suggests the following generic methodology to approach proving security when a protocol requires usage of complex data structures. The key idea is to decompose the proof, separately proving security for the data structure and for the protocol separately, using these steps:

- S1: Identify and extract the properties of the data structure that are required by the protocol, and prove that there is an implementation of a data structure that satisfies these properties. As the semantics of the data structure are protocol-independent, so are these properties.
- S2: Assuming that we have a data structure with such properties, prove security for the protocol itself.

We depict this methodology in Fig. 3. For the case of accountable decryption it works as follows. In Step S1, we propose a suitable model for the data structures employed by the ledger, which we instantiate either as a hash list or as a Merkle tree. We also define the desired properties of the data structure from the point of view of the protocol. The reason to consider two data structures is to show that the identified "interface of security properties" and the methodology are sufficiently generic to allocate several data structures and be considered for several protocols. Next, we use ProVerif to prove that the data structure satisfies the properties in the null process.

On the other hand, in Step S2, we adopt the properties for the data structure as an interface. This requires to take two actions: transform these properties as ProVerif axioms, and transform any related predicate to a blocking predicate. This will allow us to prove the security property of the protocol disregarding the particular implementation of the data structure. Observe that this step is independent from Step S1 above, i.e. they both can be executed in parallel.

By decomposing the problem into two parts, ProVerif has a better chance of avoiding nontermination, without losing soundness of the proof. Moreover the "interface" approach allows for proving security for more complex data structures than Merkle trees, and allows proving security of the protocol for any underlying data structure satisfying the data structure properties.

The remainder of this section describes in detail the methodology, by detailing the implementation of the two data structures commented above, the interface of security properties of the data structure, and the modelling and analysis of the accountable decryption protocol.

3.1. Modeling the ledger data structure

Recall that the ledger is an untrusted party that maintains an append-only log L. From the point of view of proving security properties for a protocol, it does not matter which is the particular data structure employed by the log maintainer, as long as it is append-only, and provides an interface to construct and verify proofs of presence and proofs of extension. Whichever data structure we use must define the clauses from which valid proofs of presence and the proofs of extension can be derived. We discuss two common data structures below, namely hash lists and Merkle trees.

Even though the most interesting case is indeed the latter one, the main ideas of our approach are more clearly seen using hash lists. For this reason, we provide a more detailed explanation on how to model the proofs on hash lists, whereas we provide a brief example on Merkle trees, avoiding routine technicalities that might hinder the main points we want to state. The complete models, both for hash lists and Merkle trees are available at [10].

For convenience, we parameterize the hash function H with two values, e.g. $H(x, h)$, where h is an output of the hash function. This can be implemented simply as concatenating the values x and h in a regular hash function. Hashes of single values x are interpreted as $H(x, h_0)$, where h_0 denotes the null hash.

3.1.1. Hash lists.

First, consider the case where L is a hash list. That is, L is represented by h, where

$$h = H(R_n, \ldots, H(R_2, H(R_1, h_0)) \cdots).$$

In this case, a proof of presence of R_i in the list represented by h simply consists of the elements inserted after R_i, plus the hash of the list before inserting R_i, i.e.

$$\pi = \mathsf{pp}\big((R_n, \ldots, R_{i+1}), h'\big),$$

where $h' = H(R_{i-1}, \ldots, H(R_1, h_0) \ldots)$ and pp is the constructor for the proof of presence. The predicate

$\mathsf{verify_pp}(\pi, R_i, h)$ states that π is a valid proof of presence of R_i in the list represented by h. For a hash list, verify_pp will check that the following equation holds:

$$h = H(R_n, \dots, H(R_{i-1}, H(R_i, h')) \cdots). \tag{1}$$

In order for ProVerif to handle predicates in the resolution algorithm, we need to provide the set of Horn clauses from which all valid predicates can be derived inductively. In the case of the proof of presence, these clauses are

$$\begin{aligned}
&\mathsf{verify_pp}(\mathsf{pp}(\mathrm{nil}, h'), R, H(R, h')), \quad (2)\\
&\mathsf{verify_pp}(\mathsf{pp}(\ell, h'), R, h) \rightarrow\\
&\quad \mathsf{verify_pp}(\mathsf{pp}(\mathsf{cons}(Q, \ell), h'), R, H(Q, h)). \quad (3)
\end{aligned}$$

Clause (2) states the "base case": a proof of presence to verify the last entry on a hash list is an empty list of elements and the immediate hash h' before inserting R. It can be readily seen that the verification from Eq. (1) holds. Clause (3) states the recursive nature of the proof of presence: if a list contains R, an extension of this list with an element Q also contains R, and a proof of presence can be easily derived by prepending Q into the list of elements of the original proof of presence.

For the proof of extension, let L_1 and L_2 be two hash lists represented by h_1 and h_2, respectively, and with lengths $n_1 \leq n_2$. A proof of extension ρ that L_2 extends L_1 simply consists of the list of elements inserted into L_2 after the last element R_{n_1} inserted into L_1, that is,

$$\rho = \mathsf{pe}(R_{n_2}, \dots, R_{n_1+1}),$$

where pe is the constructor for proofs of extension. The implementation of predicate $\mathsf{verify_pe}(\rho, h_1, h_2)$, which verifies that ρ is a valid proof of extension for the lists L_1, L_2, will check that the equation below holds:

$$h_2 = H(R_{n_2}, \dots, H(R_{n_1+1}, h_1) \cdots). \tag{4}$$

The predicate verify_pe is defined by the following Horn clauses:

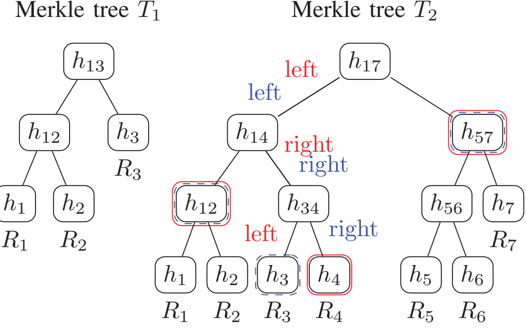
$$\begin{aligned}
&\mathsf{verify_pe}(\mathsf{pe}(\mathrm{nil}), h, h), \quad (5)\\
&\mathsf{verify_pe}(\mathsf{pe}(\ell), h_1, h_2) \rightarrow\\
&\quad \mathsf{verify_pe}(\mathsf{pe}(\mathsf{cons}(R, \ell)), h_1, H(R, h_2)). \quad (6)
\end{aligned}$$

Indeed, Clause (5) above is the "base case," and indicates that a hash list L represented by h extends itself trivially, hence the proof of extension is the empty list. Clause (6) states that given a valid proof of extension indicting that a list L_2 represented by h_2 extends L_1 represented by h_2, then $\mathsf{pe}(\mathsf{cons}(R, \ell))$ is a valid proof of extension stating that R appended to L_2 also extends L_1.

Note that both π and ρ are data structures of size $O(n)$, and the verification of these proofs (Eq. (1) and (4)) also takes time $O(n)$.

Finally, we also require a predicate $\mathsf{repr}(\ell, h)$ to state the fact that h represents a hash list data structure containing the elements in $\ell = (R_n, \dots, R_1)$, inserted in reverse order. Clearly, the clauses defining this predicate are

$$\begin{aligned}
&\mathsf{repr}(\mathrm{nil}, h_0),\\
&\mathsf{repr}(\ell, h) \rightarrow \mathsf{repr}(\mathsf{cons}(R, \ell), H(R, h)).
\end{aligned}$$



Label h_{ij} is the digest containing $R_i, \dots, R_j$.
The proof of presence of R_4 in T_2 is
$\big((\mathrm{left}, h_{57}), (\mathrm{right}, h_{12}), (\mathrm{right}, h_3)\big)$, shown in blue.
The proof of extension of T_1 to T_2 is
$R_3, \big((\mathrm{left}, h_{57}), (\mathrm{right}, h_{12}), (\mathrm{left}, h_4)\big)$, shown in red.

Figure 4. Examples of Merkle trees

3.1.2. Merkle trees.

We assume that the reader has some familiarity with Merkle trees. It is a more efficient data structure to construct the proofs: the memory and time requirements are reduced to $O(\log n)$ (compared with $O(n)$ for hash lists). We omit most of the formalities and consider the example of Merkle trees depicted in Fig. 4. Recall that a Merkle tree is a binary tree that assigns a hash value for each node, computed as the hash of its child nodes, e.g. $h_{12} = H(h_1, h_2)$. By convention, the hashes on the leaf nodes are defined as the hash of the associated element, i.e. $h_i = H(R_i, h_0)$. Hence, the root tree hash that represents the tree T_2 in Fig. 4 is $h = h_{17} = H(h_{14}, h_{57})$.

A proof of presence that R_i is present in the tree is constructed by providing the complementary node at each tree level. For example, the proof of presence π of R_4 in T_2 is the list of values

$$\pi = \mathsf{pp}\big((\mathrm{left}, h_{57}), (\mathrm{right}, h_{12}), (\mathrm{right}, h_3)\big).$$

The constants "left" and "right" are appended to each element to record the relative position of the path leading to the root three hash. Then, $\mathsf{verify_pp}(\pi, R_4, h_{17})$ will check that

$$h = H(H(h_{12}, H(h_3, H(R_4, h_0))), h_{57}).$$

The predicate verify_pp is defined by the following Horn clauses:

$$\begin{aligned}
&\mathsf{verify_pp}(\mathsf{pp}(\mathrm{nil}), R, H(R, h_0)), \quad (7)\\
&\mathsf{verify_pp}(\mathsf{pp}(\ell), R, h_\ell) \rightarrow\\
&\quad \mathsf{verify_pp}\big(\mathsf{pp}\big(\mathsf{cons}((\mathrm{left}, h_r), \ell)\big), R, H(h_\ell, h_r)\big), \quad (8)\\
&\mathsf{verify_pp}(\mathsf{pp}(\ell), R, h_r) \rightarrow\\
&\quad \mathsf{verify_pp}\big(\mathsf{pp}\big(\mathsf{cons}((\mathrm{right}, h_\ell), \ell)\big), R, H(h_\ell, h_r)\big). \quad (9)
\end{aligned}$$

It is not difficult to see that these clauses represent inductively the proof of presence, starting from the base case in Clause (7), and defining recursively the proofs whose next step are left or right paths in Clauses (8) and (9), respectively.

A proof of extension for Merkle trees can be seen as proofs that the last element of the smaller tree is present in the smaller and the larger tree, and a relationship between the two proofs. For example, to prove that T_2 extends T_1 in Fig. 4, the proof of extension ρ must include the following values:

$$\rho = \mathsf{pe}\big(R_3, \big((\text{left}, h_{57}), (\text{right}, h_{12}), (\text{left}, h_4)\big)\big).$$

The verification of the proof of extension consists of (i) verifying that the list $\big((\text{left}, h_{57}), (\text{right}, h_{12}), (\text{left}, h_4)\big)$ is a proof of presence of R_3 in T_2; and (ii) that this list filtered by only keeping the elements (right, x) is a proof of presence of R_3 in T_1.

We refer the reader to the repository of the models [10] for the Horn clause definitions of the predicates repr and verify_pe.

3.2. Properties for the data structure

Regardless of what data structure is used to model the ledger, the protocol expects that a number of properties hold, which can be abstracted away from the particular data structure. In order to express these properties, we consider the same three predicates we presented above (verify_pp, verify_pe and repr) and we axiomatize their intuitive semantics, i.e. verify_pp validates proofs of presence, verify_pe validates proofs of extension and repr validates that a digest represents the contents of the data structure. We present the properties as first-order logical formulas as follows:

Proof for the empty list. A data structure h_0 represents the situation where the list is empty:

$$\forall h.\ \mathsf{repr}(\text{nil}, h) \implies h = h_0. \tag{P1}$$

Correctness of the proof of presence. $\mathsf{mem}(R, \ell)$ iff its presence can be proved:

$$\forall R, \ell, h, \pi.$$
$$\mathsf{repr}(\ell, h) \wedge \mathsf{mem}(R, \ell) \implies \mathsf{verify_pp}(\pi, R, h), \tag{P2}$$
$$\mathsf{repr}(\ell, h) \wedge \mathsf{verify_pp}(\pi, R, h) \implies \mathsf{mem}(R, \ell). \tag{P3}$$

Correctness of the proof of extension. A data structure for a list ℓ extends the data structure for any of its suffixes:

$$\forall R, \ell, h, \rho.\ \mathsf{repr}(\mathsf{cons}(R, \ell), h) \implies \exists h'.\ \mathsf{repr}(\ell, h') \wedge \mathsf{verify_pe}(\rho, h', h). \tag{P4}$$

Transitivity of the proof of extension. If h_3 extends h_2, which in turn extends h_1, then h_3 extends h_1:

$$\forall \rho_1, \rho_2, h_1, h_2, h_3.$$
$$\mathsf{verify_pe}(\rho_1, h_1, h_2) \wedge \mathsf{verify_pe}(\rho_2, h_2, h_3) \implies \exists \rho_3.\ \mathsf{verify_pe}(\rho_3, h_1, h_3). \tag{P5}$$

Compatibility of the proofs of extension and presence. If an element is present in a list, then it remains present after further elements have been added to it:

$$\forall R, \pi_1, \rho, h_1, h_2.$$
$$\mathsf{verify_pp}(\pi_1, R, h_1) \wedge \mathsf{verify_pe}(\rho, h_1, h_2) \implies \exists \pi_2.\ \mathsf{verify_pp}(\pi_2, R, h_2). \tag{P6}$$

```
process
  !
  new dk; // Decryption key
  new sk; // Signing key
  out(c, (enc_key(dk), ver_key(sk))); // Output public
    keys
  new cell;      // Memory cell of the trustee
  new monitor; // Priv. chan. to store generated
    ciphertexts
  ( out(cell, (0,h0)) // Initialize trustee memory cell
    | ( ! // Trustee decrypt protocol
      in(cell, (i,h));
      in(c, (R,π,ρ,h'));
      if verify_pe(ρ,h,h') then
      if verify_pp(π,R,h') then
      event Decrypted(cell,i,R);
      out(c, decr(dk,R));
      out(cell, (i+1,h'))
    ) | ( ! // Trustee sign hash protocol
      in(cell, (i,h));
      in(c, v);
      event Signature(cell,i,h,sign(sk,(v,h)));
      out(c, sign(sk,(v,h)));
      out(cell, (i,h))
    ) | ( ! // Subject ciphertext generation
      new s;
      event Secret(s);
      out(c, encr(ek, s));
      out(monitor, (s, encr(ek, s)))
    ) | ( ! // Monitor ciphertext
      in(monitor, (s, R));
      in(c, =s);
      new v;
      event Name(v,s,enc_key(dk),ver_key(sk));
      out(c, v);
      in(c, σ);
      let (=v,h) = checksign(vk,σ) in
      event AfterSeeingSecret(R,h)
    ))
```

Figure 5. Transparent decryption model (sketch)

Consistency of digest representation. If two lists are represented by the same digest, then they are equal:

$$\forall \ell_1, \ell_2, h.\ \mathsf{repr}(\ell_1, h) \wedge \mathsf{repr}(\ell_2, h) \implies \ell_1 = \ell_2. \tag{P7}$$

As described above, (P1)-(P7) can be regarded as an "interface" of security properties, which is instantiated as ProVerif queries when we prove them for the data structure in Step S1 of the methodology, and it is instantiated as ProVerif axioms with blocking predicates when they are used to prove security in the protocol, in Step S2.

3.3. Modelling the transparent-decryption protocol and its security properties

In this section, we provide the most relevant details of the model of the protocol for transparent decryption. A sketch of the model is depicted in Fig. 5. We omit the definitions of the required equational theories for hashing, public key encryption, signatures, types and events. The model essentially consists of four sub-processes running in parallel.

The main process starts by creating private and public keys and initializing the public channel and two private

channels. The requirement of private channels is because, whereas private constant values such as private keys can simply be defined within the scope of the trusted process as new values, the standard way to store mutable values in applied-pi calculus is through private channels. Thus, the channels used by our model are:

- Channel `c`: public channel used for communication across the parties.
- Channel `cell`: private channel used as a memory cell for the trustee, in order to store the last seen hash value.
- Channel `monitor`: private channel used to pass secret values and ciphertexts from the subject party to a "monitor process" that will be discussed below.

Below we relate how the four sub-processes map to the different parties presented in Sec. 2.2. However, we remark that, as our trust assumptions are very weak, we do not require to model all the parties so that ProVerif can reason about the security of the protocol:

- Subjects are represented by a sub-process that creates new secrets s and output ciphertexts generated using the trustee's public encryption key ek. An event `Secret(`s`)` is required to be used in the definition of the security properties.
- Trustees run the two protocols from Fig. 1, and thus are modelled using two sub-processes. We do not consider threshold decryption schemes for this proof of concept. Note that the first and last step in each of these protocols consists in reading and writing to the trustee memory cell, respectively. Again, the events `Decrypted(cell,`i, R`)` and `Signature (cell,`i, h, σ`)` are used to define the security properties. The former is executed when the trustee accepts to decrypt R, and the latter is used when the trustee casts a signature of its last seen hash.
- The decryption requester and the log L are untrusted entities, as discussed in Sec. 2.2. Hence, there is no need to model them, as the Dolev-Yao adversary will emulate their behaviour.
- Finally, a monitor process is required to ensure that the protocol works as intended and satisfies the claimed security properties. Its main task consists in interacting with the trustee through its second protocol (right column on Fig. 1) by generating a random nonce v and obtaining the signature of the trustee's last seen hash. Once this happens, the event `AfterSeeingSecret(`R, h`)` is declared, stating that the monitor has a proof that the trustee has stored the hash value h after the subject has created the ciphertext.

The sub-processes are accordingly replicated to model arbitrary executions of the protocols, and arbitrary number of independent trustees that might monitor different logs.

The formalization of the main security property of the accountable decryption protocol, presented in Sec. 2.2, is

$$\forall R, h.\ \text{AfterSeeingSecret}(R, h) \implies \exists \pi.\ \mathsf{verify_pp}(\pi, R, h). \quad (10)$$

Indeed, the location of `AfterSeeingSecret(`R, h`)` in the monitor process captures the fact that a certain secret s associated to ciphertext (decryption request) R has been observed in the public channel, and the last observed hash value by the trustee is h. Therefore, any occurrence of this event means that there must exist a proof of presence π stating that R is in the data structure represented by h.

In order to prove this query, we consider some additional lemmas such as the following one:

$$\begin{aligned}&\forall v, s, i, h, \mathit{cell}, sk, dk.\\ &\ \text{Signature}(\mathit{cell}, i, h, \mathsf{sign}(sk, (v, h)))\wedge\\ &\quad \text{Name}(v, s, \mathsf{enc_key}(dk), \mathsf{ver_key}(sk)) \implies\\ &\qquad \exists j.\ \text{Decrypted}(\mathit{cell}, j, \mathsf{encr}(\mathsf{enc_key}(dk), s)) \wedge j < i.\end{aligned}$$

Notice that the memory cell of the trustee stores two pieces of information: the number of times it decrypted a ciphertext, and the latest hash value it received. Hence this lemma states that when the trustee signed the name v generated by the monitor (which monitors the trustee with encryption key $\mathsf{enc_key}(dk)$ and verification key $\mathsf{ver_key}(sk)$) after receiving a secret s, then the trustee must have decrypted it *strictly before* (i.e. $j < i$).

This lemma allows us to help ProVerif by linking the content of the memory cell (i.e. the number of times it decrypted a encryption) with the order of events that were emitted in the trace. Such links are usually abstracted away by ProVerif during the saturation procedure hence the need for us to provide it within a lemma. Note that we also considered some additional lemmas and axioms, specific to the management of memory cells, in the vein of [36].

Our ProVerif models only take couple of seconds to execute on a standard laptop.

4. Extending ProVerif to support arbitrary predicates in lemmas and axioms

To complete the methodology depicted in Fig. 3 on our running example, we need to (1) prove the properties P1 to P7, and (2) prove the main protocol while expressing the properties P1 to P7 as axioms. All the properties in our interface are in fact correspondence properties that are within the scope of ProVerif. For example, the query corresponding to Property P6 would be expressed as follows:

```
query pe1,pe2,pe3:proof_of_extension,
      d1,d2,d3:digest;
verify_pe(pe1,d1,d2)&&verify_pe(pe2,d2,d3)
      ==> verify_pe(pe3,d1,d3)
```

However, in its current version, ProVerif imposes a syntactic restriction on axioms and lemmas: the conclusion of a lemma can only contain events, equalities, disequalities and blocking user-defined predicates. Specifically, the native facts $\mathsf{att}(M)$ and $\mathsf{mess}(M, N)$ as well as the clause-based user-defined predicates cannot be used in the conclusion of a lemma.

This restriction of facts prevents us from achieving both steps of our methodology. The second step is unattainable as ProVerif will directly reject such a query if it is written as an axiom. For the first step, the query will be accepted by ProVerif but it will fail to prove it. Such a query requires a proof by induction, which internally corresponds to transforming a query into an *inductive lemma* that has the same syntactic limitation as declared axioms and lemmas. By extending ProVerif to allow any

predicates in the conclusion of axioms and lemmas, we are able to complete our methodology.

In this section, we provide a high-level description of ProVerif's procedure and how we extended it.

4.1. Description of ProVerif's procedure

Horn clauses are used to describe the semantics of user-defined predicates as previously described but they are also the building blocks of ProVerif's internal procedure to prove a secrecy property and more generally a correspondence property. Specifically, ProVerif first translates the protocol given as input into a set $\mathbb{C}$ of Horn clauses. It then proceeds to *saturate* this set $\mathbb{C}$ and the clauses that define the user-defined predicates, yielding a simpler set of clauses that derives the same facts. The procedure completes by verifying that the saturated clauses satisfy the security property.

Translation into Horn clauses. In addition to the user-defined predicates, ProVerif considers natively four additional predicates over terms representing the interactions between the attacks and the processes: $\mathsf{att}(M)$, $\mathsf{mess}(M, N)$, and two predicates for events $\mathsf{s\text{-}event}(ev)$ and $\mathsf{event}(ev)$. *Sure-events*, i.e. $\mathsf{s\text{-}event}(ev)$, will only appear in hypotheses of Horn clauses, whereas events $\mathsf{event}(ev)$ will only appear in their conclusion. This separation ensures that events are not *resolved* during the saturation procedure and so their occurrence in a Horn clause is preserved through resolution rule.

Using these predicates, ProVerif generates a set of clauses representing the capabilities of the attacker, which include, for example:

$$\mathsf{att}(x) \wedge \mathsf{att}(y) \rightarrow \mathsf{att}(\mathsf{encr}(x, y)) \quad (11)$$

$$\mathsf{att}(\mathsf{encr}(x, y)) \wedge \mathsf{att}(x) \rightarrow \mathsf{att}(y) \quad (12)$$

$$\mathsf{att}(x) \wedge \mathsf{mess}(x, y) \rightarrow \mathsf{att}(y) \quad (13)$$

$$\mathsf{att}(x) \wedge \mathsf{att}(y) \rightarrow \mathsf{mess}(x, y) \quad (14)$$

The first two clauses model that the attacker can encrypt and decrypt provided that it knows the secret key. The last two clauses model that the attacker can read and write on a channel that it knows.

The formal description of processes into Horn clauses is out of scope of this paper (see [9] for more details), but we provide some intuition in the following example.

Example 2. *The translation of the* `Ciphertext Generator` *process would yield at least the clause*

$$\mathsf{s\text{-}event}(\mathrm{Secret}(s)) \rightarrow \mathsf{att}(\mathsf{encr}(ek, s)) \quad (15)$$

indicating that the attacker can obtain the encryption of the secret s by the key ek. The Horn clause also indicates that the event $\mathrm{Secret}(s)$ is triggered before the encryption is sent to the attacker.

Similarly, the translation of process modelling the accountable decryption device will generate, in particular, the clause:

$$\mathsf{mess}(d, (i, H_0)) \wedge \mathsf{att}(\mathsf{encr}(ek, x)) \wedge \mathsf{att}((pi, r, H_1)) \wedge \\ \mathsf{verify_pe}(r, H_0, H_1) \wedge \mathsf{verify_pp}(pi, \mathsf{encr}(ek, x), H_1) \wedge \\ \mathsf{s\text{-}event}(\mathrm{Decrypted}(i, \mathsf{encr}(ek, x))) \rightarrow \mathsf{att}(x) \quad (16)$$

This clause represents the informal statement: Assuming that the cell of the device contains the i-th digest received H_0, if the attacker can provide a proof of extension r from H_0 to a new digest H_1, a proof of presence pi of some ciphertext $\mathsf{encr}(ek, x)$ in the digest H_1 then the attacker can obtain the plain text x. As in the previous clause, the event $\mathrm{Decrypted}(i, \mathsf{encr}(ek, x))$ will be triggered before the attacker obtains x.

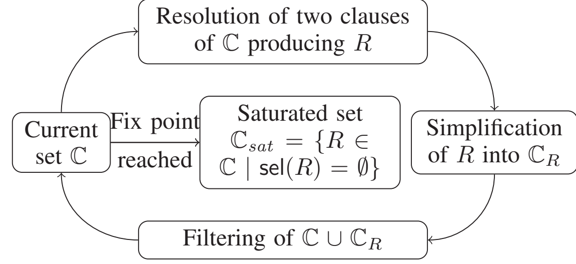


Figure 6. A schematic summary of ProVerif's saturation procedure

Saturation. The core step in the saturation procedure consists of taking two existing Horn clauses and combining them into a new one, hopefully simpler. This process is called the *resolution step*. For example, the hypothesis of clause 16 contains in hypothesis the fact $F = \mathsf{att}(\mathsf{encr}(ek, x))$ representing that the attacker must know the ciphertext $\mathsf{encr}(ek, x)$. In order to deduce all the possible ways the attacker may deduce this ciphertext, ProVerif resolves this fact by combining it with Horn clauses whose conclusion can be unified with F. This is the case with clause 15, which results in the following clause:

$$\mathsf{mess}(d, (i, H_0)) \wedge \mathsf{s\text{-}event}(\mathrm{Secret}(s)) \wedge \mathsf{att}((pi, r, H_1)) \wedge \\ \mathsf{verify_pe}(r, H_0, H_1) \wedge \mathsf{verify_pp}(pi, \mathsf{encr}(ek, s), H_1) \wedge \\ \mathsf{s\text{-}event}(\mathrm{Decrypted}(i, \mathsf{encr}(ek, s))) \rightarrow \mathsf{att}(s).$$

Note that the hypothesis $\mathsf{s\text{-}event}(\mathrm{Secret}(s))$ of clause 15 has replaced F in the new clause and the unification of the two facts resulted in x being instantiated by s.

In our example, the fact F in clause 16 could also have been resolved with clause 12, which would result in a new rule like clause 16 but with F replaced by the two facts $\mathsf{att}(\mathsf{encr}(x', \mathsf{encr}(ek, x)))$ and $\mathsf{att}(x')$. This would directly lead to a loop during the saturation procedure as the former fact could once again be resolved by the same clause 12. To avoid this problem, ProVerif uses a *selection function* on Horn clauses that returns the set of facts from the hypothesis of a clause that can be resolved, i.e. $\mathsf{sel}(H \rightarrow C) = S$ with $S \subseteq H$. In particular, the facts $\mathsf{att}(x)$ with x a variable are never in S.

The resolution process repeats until a fixed point is reached, i.e. until no resolution can produce a clause that is not redundant with an existing one. Once a fixed point is reached, ProVerif only keeps the set of *saturated clauses*, which are the clauses R from $\mathbb{C}$ such that $\mathsf{sel}(R) = \emptyset$. A schematic summary of ProVerif's saturation procedure is given in Fig. 6.

Generalizing the events and sure-events separation. We previously mentioned that ProVerif considers two

different predicates for events: the sure-event predicate s-event and the event predicate event. During the translation of processes into Horn clauses, the former type of event only occurs in the hypotheses of the clauses whereas the latter only occurs in the conclusion of the clauses. It ensures that an application of the resolution rule never resolves a sure-event $\mathsf{s\text{-}event}(ev)$ from a clause. This is critical for the verification of correspondence queries. For example, to verify the query $\mathsf{event}(A) \Rightarrow \mathsf{event}(B)$, ProVerif will check that for all saturated clauses in $\mathbb{C}_{sat}$ of the form $H \to \mathsf{event}(A)$, the sure-event $\mathsf{s\text{-}event}(B)$ occurs in H. In this case, ProVerif concludes that the query holds. Note that to ensure soundness, the selection function will never consider sure-events, i.e. $\mathsf{s\text{-}event}(ev) \notin \mathsf{sel}(R)$, which guarantees that the resolution rule does not try to resolve a fact that cannot be resolved by design.

Since sure-events are never selected by the selection function, they can be seen as the *blocking* counterpart of the predicate event. To add clause-based, user-defined predicates in the conclusion of lemmas, as well as the predicates $\mathsf{att}(M)$ and $\mathsf{mess}(M, N)$, we generalize this concept by associating to all predicates p a blocking predicate denoted b-p. Therefore, we consider natively the predicates b-att and b-mess. Moreover, for all user-defined predicates $p \in \mathcal{F}_p$, we consider the blocking predicate b-p and denote by $\mathcal{F}_{bp}$ their set. Finally, we rename the predicate s-event as b-event.

We also amend the selection function by additionally requiring that no blocking fact can be selected: For all F, $\mathsf{b}\text{-}F \notin \mathsf{sel}(H \to C)$. Thus, the resolution rule will never attempt to resolve a blocking predicate.

Application of lemmas. ProVerif axioms and proved lemmas are applied during the simplification phase of the saturation procedure. Consider a lemma $\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j$, where ψ_j are conjunctions of facts. In the current ProVerif, each ψ_j could only be composed of events, disequalities and inequalities. The lemma would be applied on a clause $H \to C$ when there exists a substitution σ such that $F_i\sigma$ is in H for all $i \in \{1, \dots, n\}$. The application of the lemma would then produce a set of m clauses $\{H \wedge \psi'_j\sigma \to C\}_{j=1}^{m}$, where each ψ'_j is the disjunct ψ_j with events replaced by their sure-event counterpart.

Thanks to our extension, the disjunct ψ_j may now contain any type of predicate. We update the simplification rule by first defining the transformation $\lceil\psi\rceil^b$ built from ψ where all facts $p(M_1, \dots, M_n)$ in ψ are replaced by b-$p(M_1, \dots, M_n)$. The rule for applying lemmas is then defined as follows:

$$\frac{\mathbb{C} \cup \{R = (H \to C)\} \qquad (\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j) \in \mathcal{L} \quad \forall i, \mathsf{b}\text{-}F_i\sigma \in H \text{ or } F_i\sigma \in H}{\mathbb{C} \cup \{H \wedge \lceil\psi_j\sigma\rceil^b \to C\}_{j=1}^{m}}$$

Note that the application condition requires that either $\mathsf{b}\text{-}F_i\sigma \in H$ or $F_i\sigma \in H$. Indeed, the lemma can be applied on facts introduced by a previous application of another lemma or more commonly on an event, hence the condition $\mathsf{b}\text{-}F_i\sigma \in H$. Since a lemma may have an attacker fact, a message fact or a fact using a user predicate in its premisse, we also need to match the non-blocking form of the fact, i.e. $F_i\sigma \in H$. Note that all facts added in the clauses, i.e. facts in $\lceil\psi_j\sigma\rceil^b$, are blocking.

Improving other simplification rules. The purpose of applying lemmas is to increase ProVerif precision or to help it terminate. As such, we can amend other simplification rules used by ProVerif to benefit from the blocking facts. For example, ProVerif employs the following simplification rule to remove tautologies:

$$\frac{\mathbb{C} \cup \{F \wedge H \to F\}}{\mathbb{C}}$$

Now that we may have blocking predicates in the hypotheses of the clause, we can consider an additional tautology simplification rules defined as follows:

$$\frac{\mathbb{C} \cup \{\mathsf{b}\text{-}F \wedge H \to F\} \qquad \mathrm{pred}(p) \notin \mathcal{F}_p}{\mathbb{C}}$$

We also improve the simplification rule that removes the redundant facts from the hypotheses of a clause, which is critical to avoid termination issues:

$$\frac{\mathbb{C} \cup \{H' \wedge H \wedge \phi \to C\} \qquad \lceil H'\sigma\rceil^b \subseteq \lceil H\rceil^b \quad \phi \models \phi\sigma \qquad \mathrm{dom}(\sigma) \cap \mathrm{vars}(H, C) = \emptyset}{\mathbb{C} \cup \{H \wedge \phi\sigma \to C\}}$$

Intuitively, this rule states that to derive C, it suffices to know the derivations of H, as a derivation for H' can be build from the derivations of $H'\sigma \subseteq H$. Note that by requiring $\lceil H'\sigma\rceil^b \subseteq \lceil H\rceil^b$, a clause $\mathsf{att}(M) \wedge \mathsf{b\text{-}att}(M) \wedge G \to C$ can be either simplified into $\mathsf{att}(M) \wedge G \to C$ or into $\mathsf{b\text{-}att}(M) \wedge G \to C$. Both variants are correct and their application may be parametrized depending on one's needs. The former could be used to discard the application of a lemma that is redundant with the hypothesis of the clause whereas the latter could help terminate, as the fact $\mathsf{b\text{-}att}(M)$ will not be resolved.

Verification of the query. Once the saturation process ends, ProVerif still needs to verify the queries. Intuitively, on a query $F_1 \wedge \cdots \wedge F_n \Rightarrow \psi$ with $F_i = p_i(t_1^i, \dots, t_{m_i}^i)$, ProVerif starts by generating a clause $F_1 \wedge \cdots \wedge F_n \to C$ with $C = \mathsf{q}(t_1^1, \dots, t_{m_n}^n)$ and q a special predicate only used in the verification of query. Typically, C represents the conjunction of facts $F_1, \dots, F_n$. ProVerif then applies once again a saturation of $\mathbb{C}_{sat} \cup \{F_1 \wedge \cdots \wedge F_n \to C\}$ and checks the validity of ψ on the obtained saturated set. Note that this second saturation is slightly different from the first saturation in the sense that it is *order preserving*, i.e. in a clause $F \wedge H \to \mathsf{q}(t_1^1, \dots, t_{m_n}^n)\sigma$, Proverif can indicate, for any i, whether the fact F was generated when resolving F_i. This property is particularly important for proving queries by induction. The details of how the orders are preserved can be found in [9].

Example 3. *Let us show how we are able to prove by induction the transitivity of proof of extension in the hash list data structure:*

```
query pe1,pe2,pe3:proof_of_extension,
      d1,d2,d3:digest;
verify_pe(pe1,d1,d2)&&verify_pe(pe2,d2,d3)
      ⟹ verify_pe(pe3,d1,d3) [induction].
```

To ease the reading, let us denote by v_{pe} *the predicate* verify_pe. *Recall that the proof of extension is defined by the following two clauses.*

$$\mathsf{v}_{\mathsf{pe}}(\mathsf{pe}(\mathrm{nil}), d, d), \quad (17)$$
$$\mathsf{v}_{\mathsf{pe}}(\mathsf{pe}(\ell), d_1, d_2) \rightarrow \mathsf{v}_{\mathsf{pe}}(\mathsf{pe}(\mathsf{cons}(x, \ell)), d_1, H(x, d_2)). \quad (18)$$

These clauses will be left unchanged through the saturation procedure, i.e. if $\mathbb{C} = \{(17), (18)\}$ *then* $\mathsf{sat}(\mathbb{C}) = \mathbb{C}$.

To verify the query, ProVerif first considers the query clause $RQ = F_1 \wedge F_2 \rightarrow C$ *where* $F_1 = \mathsf{v}_{\mathsf{pe}}(pe_1, d_1, d_2)$, $F_2 = \mathsf{v}_{\mathsf{pe}}(pe_2, d_2, d_3)$ *and* $C = \mathsf{q}(pe_1, d_1, d_2, pe_2, d_2, d_3)$. *ProVerif then applies the saturation procedure on* $\mathbb{C} \cup \{RQ\}$.

We illustrate the application of the inductive lemma on one of the resolutions that ProVerif will apply during this second saturation procedure: the resolution of $\mathsf{v}_{\mathsf{pe}}(pe_2, d_2, d_3)$ *from RQ with the clause (18) which yields the following clause*

$$RQ' = F_1\sigma \wedge \mathsf{v}_{\mathsf{pe}}(\mathsf{pe}(\ell), d_2, d_3') \rightarrow C\sigma$$

with $\sigma = \{pe_2 \mapsto \mathsf{pe}(\mathsf{cons}(x, \ell)), d_3 \mapsto H(x, d_3')\}$.

Let us denote $F_3 = \mathsf{v}_{\mathsf{pe}}(\mathsf{pe}(\ell), d_2, d_3')$. *Note that* $C\sigma$ *represents the conjunction* $F_1\sigma \wedge F_2\sigma$ *with:*

$$F_2\sigma = \mathsf{v}_{\mathsf{pe}}(\mathsf{pe}(\mathsf{cons}(x, \ell)), d_2, H(x, d_3'))$$

Since F_3 *was obtained while resolving* $F_2\sigma$, *any instantiation* $F_3\alpha$ *would be* satisfied strictly before $F_2\sigma\alpha$ *(we detail this notion in the next section). Thus, ProVerif can apply our inductive hypothesis on* $F_1\sigma, F_3$ *and so it will add the blocking fact* $\mathsf{b}\text{-}\mathsf{v}_{\mathsf{pe}}(pe', d_1, d_3')$ *in the hypothesis of the clause, yielding:*

$$RQ_2 = F_1\sigma \wedge \mathsf{v}_{\mathsf{pe}}(\mathsf{pe}(\ell), d_2, d_3') \wedge \mathsf{b}\text{-}\mathsf{v}_{\mathsf{pe}}(pe', d_1, d_3') \rightarrow C\sigma$$

On the clause RQ_2, *ProVerif will be able prove the query, i.e. finding a derivation of* $\mathsf{v}_{\mathsf{pe}}(pe_3, d_1, d_3)\sigma = \mathsf{v}_{\mathsf{pe}}(pe_3, d_1, H(x, d_3'))$ *for some* pe_3, *by using the blocking fact* $\mathsf{b}\text{-}\mathsf{v}_{\mathsf{pe}}(pe', d_1, d_3')$ *and the clause (18).*

4.2. Soundness

The soundness of the saturation procedure comes in three steps.

Derivation of satisfiable facts. Given a process P and a trace T of P, one can show that that for all satisfiable facts F in T, i.e. $T \vdash F$, there exists a derivation $\mathcal{D}$ of F in the set of initial clauses $\mathbb{C}_{init}(P)$ translated from P and the set of clauses $\mathbb{C}_{\mathsf{b}}(T) = \{\lceil F \rceil^b \mid T \vdash F\}$. The original soundness result [9, Theorem 1] was only considering the sure-events, i.e. $\{\mathsf{s\text{-}event}(ev) \mid T \vdash \mathsf{event}(ev)\}$.

Since blocking predicates are an artifice to prevent resolution within the saturation procedure, they should not affect their satisfiability in a trace T. Hence, we augment the satisfaction relation $\vdash$ by requiring that for all blocking facts $\mathsf{b}\text{-}F$, $T \vdash \mathsf{b}\text{-}F$ if and only if $T \vdash F$.

In the current ProVerif, the derivation $\mathcal{D}$ is shown to satisfy an invariant on the Horn clauses labeling its nodes that relates satisfaction of facts with the size of the trace. Consider a ground fact F such that $T \vdash F$. We define $\mathsf{min}_T(F) = \min\{|T'| \mid T' \text{ prefix of } T \wedge T' \vdash F\}$ which represents the size of the smallest prefix of T that satisfies F. The invariant intuitively indicates that in a derivation, all instantiated facts are satisfied by T and if a clause $H \rightarrow C$ is used to derive C then the facts in H must be satisfied strictly before C in the trace T. We extend this invariant to take into account the user-defined predicates and blocking predicates

However, when a fact F corresponds to a user-defined predicates, e.g. $F = p(M_1, \dots, M_n)$, we always have $\mathsf{min}_T(F) = 0$ as the satisfiability of this predicate do not depend on the protocol but only on the value of the terms $M_1, \dots, M_n$. Hence, $T \vdash F$ if and only if F is satisfied in the empty trace. This prevents us from *ordering* the user-defined predicates within a clause. Hence, we strengthen the minimality function on user-defined predicates by considering the size of the derivation of $p(M_1, \dots, M_n)$ in $\mathbb{C}_{user}$ instead of its satisfiability in T. Formally, we consider $\mathsf{min}_{T,\mathbb{C}_{user}}(F)$ defined as:

- $\mathsf{min}_{T,\mathbb{C}_{user}}(F) = \mathsf{min}_T(F)$ when $\mathrm{pred}(F) \not\in \mathcal{F}_p \cup \mathcal{F}_{bp}$
- $\mathsf{min}_{T,\mathbb{C}_{user}}(F) = \min\{|\mathcal{D}| \mid \mathcal{D} \text{ derives } F' \text{ from } \mathbb{C}_{user}\}$ when $F = F'$ or $F = \mathsf{b}\text{-}F'$ with $\mathrm{pred}(F') \in \mathcal{F}_p$.

We can show state the main invariant on derivations.

Invariant 1. *Let* T *be a trace. Let* $\mathbb{C}_{user}$ *a set of clauses defining predicates in* $\mathcal{F}_p$. *Let* $\mathcal{D}$ *a derivation. We say that the invariant* $\mathsf{Inv}_{T,\mathbb{C}_{user}}(\mathcal{D})$ *holds when for all nodes of* $\mathcal{D}$ *labeled by a Horn clause* $R = H \rightarrow C$ *and a substitution* σ, $T \vdash C$ *and for all* $F \in H$, $T \vdash F$. *Moreover,*

- *if* R *is the attacker rule (13) then* $\mathsf{min}_T(\mathsf{att}(x)\sigma) < \mathsf{min}_T(\mathsf{att}(y)\sigma)$ *and* $\mathsf{min}_T(\mathsf{mess}(x, y)\sigma) = \mathsf{min}_T(\mathsf{att}(y)\sigma)$;
- *otherwise for all* $F \in H$, *if* $(\mathrm{pred}(C) \not\in \mathcal{F}_p$ *and* $\mathrm{pred}(F) \not\in \mathcal{F}_p \cup \mathcal{F}_{bp})$ *or* $(\mathrm{pred}(C) \in \mathcal{F}_p$ *and* $\mathrm{pred}(F) \in \mathcal{F}_p)$ *then* $\mathsf{min}_{T,\mathbb{C}_{user}}(F\sigma) < \mathsf{min}_{T,\mathbb{C}_{user}}(C\sigma)$

As previously mentioned, the invariant intuitively states that all facts in the derivation are satisfied in T. The first bullet point indicates that in the case of the attacker rule (13), representing that the attacker may listen on a channel if it can deduce it, the two facts $\mathsf{mess}(x, y)$ and $\mathsf{att}(y)$ are satisfied at the same moment on T. The second bullet point indicates that for all other rules, the hypohteses are satisfied strictly before the conclusion, in the sense of $\mathsf{min}_{T,\mathbb{C}_{user}}(\cdot)$ and if they belong to the same category, i.e. standard or user-defined predicates.

Preservation of the invariant during the saturation. In the second step of the soundness proof, one can show that one round *resolution-simplification-filtering* preserves the derivability of facts, i.e. if $\mathbb{C}'$ is the new set of clauses after a round of *resolution-simplification-filtering* on $\mathbb{C}$, then F derivable in $\mathbb{C}$ implies that F is derivable in $\mathbb{C}'$, with both derivations satisfying Invariant 1.

The soundness of the new tautology rule is directly given by Invariant 1. Indeed, if $\mathrm{pred}(F) \not\in \mathcal{F}_p$ then $\mathrm{pred}(\mathsf{b}\text{-}F) \not\in \mathcal{F}_p \cup \mathcal{F}_{bp}$ and so $\mathsf{min}_{T,\mathbb{C}_{user}}(\mathsf{b}\text{-}F) < \mathsf{min}_{T,\mathbb{C}_{user}}(F)$ which contradicts minimality.

Note that the soundness of the new rule for redundant facts is direct from the fact that invariant 1 is stable by removal of hypotheses.

Finally, we show that the application of lemmas is also sound. This is achieved thanks again to Invariant 1. In particular, when a lemma $\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j$ matches

H, i.e. $F_i\sigma$ is in H for all $i \in \{1, \dots, n\}$, it implies that each $F_i\sigma$ are satisfied in T. Since the lemma holds on T, at least one of the $\psi_j\sigma$ also holds in T and so all facts in $\lceil \psi_j\sigma \rceil^b$ are derivable from $\mathbb{C}_\mathsf{b}(T)$.

Application of inductive lemmas. In [9], a query $F_1 \wedge \dots \wedge F_n \Rightarrow \psi$ may be proved by induction on the multiset $\{\!\{\mathsf{min}_T(F_1\sigma), \dots, \mathsf{min}_T(F_n\sigma)\}\!\}$ when $T \vdash F_i\sigma$ for all $i = 1, \dots, n$. In practice, ProVerif transforms the query into an *inductive lemma* which is typically a lemma implied by the query. This lemma will be applied as a normal lemma during the saturation procedure except on the attacker rule (13).

The main soundness argument for applying inductive lemma during the saturation procedure is as follows. As we prove the query by induction on $\mathcal{M} = \{\!\{\mathsf{min}_T(F_1\sigma), \dots, \mathsf{min}_T(F_n\sigma)\}\!\}$, we can assume that the query holds for any instance of the premise $F_1\sigma', \dots, F_n\sigma'$ such that $\{\!\{\mathsf{min}_T(F_1\sigma'), \dots, \mathsf{min}_T(F_n\sigma')\}\!\} < \mathcal{M}$. Now consider a rule $H \rightarrow C$ used in a derivation of $F_1\sigma, \dots, F_n\sigma$, we know from Invariant 1 that all facts F in H are satisfied strictly before C which is itself satisfied strictly before at least one of the $F_i\sigma$, i.e. $\mathsf{min}_T(F) < \mathsf{min}_T(F_i\sigma)$. Therefore, if $F_1\sigma', \dots, F_n\sigma' \in H$ then we deduce that $\{\!\{\mathsf{min}_T(F_1\sigma'), \dots, \mathsf{min}_T(F_n\sigma')\}\!\} < \mathcal{M}$ and so we know that the conclusion of the inductive lemma holds, which allows us to apply it.

With user-defined predicates, we cannot use $\mathsf{min}_T(\cdot)$ because the satisfiability of a user-defined predicate does not depend on the trace but on the set of Horn clauses $\mathbb{C}_{user}$ given as input. Instead, we base the induction on the minimal size of the derivation of F_i in $\mathbb{C}_{user}$, i.e. $\mathsf{min}_{T,\mathbb{C}_{user}}(F_i\sigma)$. More specifically, if for all $i = 1, \dots, n$, $\mathrm{pred}(F_i)$ is a user-defined predicate, we prove the query by induction on the multiset $\{\!\{\mathsf{min}_{T,\mathbb{C}_{user}}(F_i\sigma)\}\!\}_{i=1}^n$.

To prove a query by induction whose premises contain both user-defined predicates and standard predicates, i.e. att and mess, we combine the two inductive measures into a single one: without lost of generality, consider the query $F_1 \wedge \dots \wedge F_n \Rightarrow \psi$ where $\mathrm{pred}(F_i), \dots, \mathrm{pred}(F_{k-1})$ are standard predicates and $\mathrm{pred}(F_k), \dots, \mathrm{pred}(F_n)$ are user-defined predicates. We prove the query for all traces T and all substitutions σ with $T \vdash F_i\sigma$, for all $i = 1, \dots, n$, by induction using the lexicographic order on $(\{\!\{\mathsf{min}_{T,\mathbb{C}_{user}}(F_i\sigma)\}\!\}_{i=1}^{k-1}, \{\!\{\mathsf{min}_{T,\mathbb{C}_{user}}(F_i\sigma)\}\!\}_{i=1}^{n})$.

Restriction to $\mathbb{C}_{sat}$. The third and final step consists of showing that when the fix point is reached, restricting the $\mathbb{C}$ to $\mathbb{C}_{sat}$ also preserves derivability. This step remains unchanged from [9, Theorem 2].

The general soundness of the saturation procedure is given by the following property.

Theorem 1. *Let P a process and T a trace of P. Let $\mathcal{L}$ a set of lemmas that hold on P. Let $\mathcal{L}_i$ be a set of inductive lemmas. We denote by $\mathbb{C}_{init}(P)$ the set of initial clauses translated from P. Let F be a fact such that $T \vdash F$. Let $\mathcal{M} = (\emptyset, \mathsf{min}_{T,\mathbb{C}_{user}}(F))$ when $\mathrm{pred}(F) \in \mathcal{F}_p$ and $\mathcal{M} = (\mathsf{min}_{T,\mathbb{C}_{user}}(F), \emptyset)$ otherwise.*

If the lemmas in $\mathcal{L}_i$ hold up to $\mathcal{M}$ excluded then there exists a derivation $\mathcal{D}$ of F from $\mathsf{sat}(\mathbb{C}_{init}(P)) \cup \mathbb{C}_\mathsf{b}(T)$ such that $\mathsf{Inv}_{T,\mathbb{C}_{user}}(\mathcal{D})$.

This theorem is a simplified version of [9, Theorem 1 and 2, simplified] adapted to user-defined predicates and generalised lemmas. In particular, the key difference between these two soundness results is that we consider in the set $\mathbb{C}_\mathsf{b}(T)$ the blocking counter part of all satisfiable facts by the trace T. Note that $\mathbb{C}_\mathsf{b}(T)$ also includes the blocking counter part of all true instances of predicates $p(M_1, \dots, M_n)$ where p is a user-defined predicate.

4.3. Applications

As we previously showed, generalizing lemmas, axioms and inductive proofs is paramount for our new methodology. Thanks to the inductive proofs, we are able to show with ProVerif all the properties P1 to P7 for both hash list and Merkle tree data structures. Moreover, as illustrated in Section 3.3, we are also able to prove the security of certificate transparency and transparent decryption with all properties P1 to P7 declared as axioms. All the proofs of the properties in the interface are proved in a single file for hash list in less than a second. For Merkle trees, the proofs are separated in five different files, each taking less than a second to be verified. The proofs of the protocols themselves with the interface are also done in less than two seconds.

Note that the generalisation of lemmas and axioms can also be of use outside of our methodology. For instance, an earlier version of our work has already been used to prove the Encrypted Client Hello extension of the TLS protocol [37]. Specifically, the saturation procedure was entering into a loop when trying to resolve a clause of the form $psk(id, k, c, s) \wedge H \rightarrow \mathsf{att}(id)$ where $psk(id, k, c, s)$ intuitively represented some shared key k with identity id between a server s and a client c. To prevent the loop, they added a lemma indicating that when $psk(id, k, c, s)$ holds then the identity was already known to the attacker or else was honestly generated:

```
lemma id,k,c,s:bitstring;
psk(id,k,c,s)  ==>  attacker(id) |
      id = honest_id(s,c,k) [induction].
```

The application of the inductive lemma on $psk(id, k, c, s) \wedge H \rightarrow \mathsf{att}(id)$ would yield two clauses, one with $\mathsf{b\text{-}att}(id)$ in the hypothesis and one where id is instantiated by $honest_{id}(s, c, k)$. The former would be removed by our new tautology rule, whereas the instantiated second clause would avoid the loop during the saturation.

5. Conclusion

Transparency in security protocols plays a fundamental role in minimizing the trust conditions for the parties involved. A clear example is that of certificate transparency, where strong security and trust assumptions on certificate authorities can be relaxed by requiring that they publish certificate issuances on a public ledger. In many cases, transparency might well constitute a required building block expected by users of a certain service, especially when it involves incursions into their privacy for a variety of reasons, a situation which is handled by transparent decryption protocols.

For these reasons, it is important to properly verify the core security property of transparency protocols, e.g. that only legitimate certificates are produced (certificate transparency) or that decryptions only take place if the requests for them are entered in a public ledger, i.e. visible to users (transparent decryption).

Because of the complex data structures that transparency protocols rely on, verifying their properties has led to designing a proof-decomposition methodology for ProVerif, as well as adding new features to the way lemmas and axioms are handled. We expect that our methodology and ProVerif enhancements can be applied to other kind of protocols involving tree-based data structures (binary trees, radix trees) and perhaps also other kinds of data structures (e.g. Bloom filters).

Acknowledgements

This work received funding from EPSRC projects *CAP-TEE: Capability Architectures for Trusted Execution*; *SIPP: Secure IoT Processor Platform with Remote Attestation*, and *User-controlled hardware security anchors: evaluation and designs*. It also received funding from the France 2030 program managed by the French National Research Agency under grant agreement No. ANR-22-PECY-0006.

References

[1] B. Laurie, "Certificate transparency," *Communications of the ACM*, vol. 57, no. 10, pp. 40–46, 2014.

[2] B. Laurie, E. Messeri, and R. Stradling, "Certificate transparency version 2.0," Internet Requests for Comments, RFC Editor, RFC 9162, Dec. 2021. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1654.txt

[3] M. D. Ryan, "Making decryption accountable," in *Cambridge International Workshop on Security Protocols*. Springer, 2017, pp. 93–98.

[4] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani *et al.*, "The avispa tool for the automated validation of internet security protocols and applications," in *International conference on computer aided verification*. Springer, 2005, pp. 281–285.

[5] V. Cheval, S. Kremer, and I. Rakotonirina, "Deepsec: deciding equivalence properties in security protocols theory and practice," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 529–546.

[6] R. Chadha, V. Cheval, Ş. Ciobâcă, and S. Kremer, "Automated verification of equivalence properties of cryptographic protocols," *ACM Transactions on Computational Logic (TOCL)*, vol. 17, no. 4, pp. 1–32, 2016.

[7] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *International conference on computer aided verification*. Springer, 2013, pp. 696–701.

[8] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1–2, pp. 1–135, Oct. 2016.

[9] B. Blanchet, V. Cheval, and V. Cortier, "Proverif with lemmas, induction, fast subsumption, and much more," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 69–86.

[10] "Proverif models and proverif source code," https://www.dropbox.com/sh/gbn5dy0amz1106f/AACbcILzg8o1Bhf5D3nMFa2Wa?dl=0, 2022.

[11] V. Cheval, J. Moreira, and M. Ryan, "Automatic verification of transparency protocols (extended version)," https://arxiv.org/abs/2303.04500, Tech. Rep., Mar. 2023.

[12] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "ARPKI: Attack resilient public-key infrastructure," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 382–393.

[13] J. Yu, V. Cheval, and M. Ryan, "DTKI: A new formalized pki with verifiable trusted parties," *The Computer Journal*, vol. 59, no. 11, pp. 1695–1713, 2016.

[14] M. D. Ryan, "Enhanced certificate transparency and end-to-end encrypted mail," in *NDSS Symposium*, 2014.

[15] R. Künnemann, I. Esiyok, and M. Backes, "Automated verification of accountability in security protocols," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 397–39 716.

[16] J. Kroll, E. Felten, and D. Boneh, "Secure protocols for accountable warrant execution," 2014, https://www.jkroll.com/papers/warrant_paper.pdf.

[17] D. Nuñez, I. Agudo, and J. Lopez, "Escrowed decryption protocols for lawful interception of encrypted data," *IET Information Security*, vol. 13, no. 5, pp. 498–507, 2019.

[18] L. Idan and J. Feigenbaum, "Prshare: A framework for privacy-preserving, interorganizational data sharing," in *Proceedings of the 19th Workshop on Privacy in the Electronic Society*, 2020, pp. 137–149.

[19] M. Li, Y. Chen, C. Lal, M. Conti, M. Alazab, and D. Hu, "Eunomia: Anonymous and secure vehicular digital forensics based on blockchain," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[20] "Privacy-preserving accountable decryption," https://pad.tech, 2022.

[21] J. A. Kroll, "Accountable algorithms," Ph.D. dissertation, Princeton University, 2015.

[22] J. Frankle, S. Park, D. Shaar, S. Goldwasser, and D. Weitzner, "Practical accountability of secret processes," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 657–674.

[23] G. Proudler, L. Chen, and C. Dalton, "Trusted platform architecture," in *Trusted Computing Platforms*. Springer, 2014, pp. 109–129.

[24] S. Johnson and D. Rizzo, "Titan silicon root of trust for google cloud," in *Secure Enclaves Workshop*, 2018, https://keystone-enclave.org/workshop-website-2018/slides/Scott_Google_Titan.pdf.

[25] "OpenTitan: an open source, transparent, high-quality reference design and integration guidelines for silicon root of trust (rot) chips," https://opentitan.org/.

[26] B. H. Møller, J. G. Søndergaard, K. S. Jensen, M. W. Pedersen, T. W. Bøgedal, A. Christensen, D. B. Poulsen, K. G. Larsen, R. R. Hansen, T. R. Jensen *et al.*, "Preliminary security analysis, formalisation, and verification of opentitan secure boot code," in *Nordic Conference on Secure IT Systems*. Springer, 2021, pp. 192–211.

[27] "PAD Places - a location sharing app on ios and google implementing privacy-preserving accountable decryption," https://www.pad.tech/pad-places, 2022.

[28] J. Parra Moyano and O. Ross, "KYC optimization using distributed ledger technology," *Business & Information Systems Engineering*, vol. 59, no. 6, pp. 411–423, 2017.

[29] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the tls 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 483–502.

[30] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *2017 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 435–450.

[31] N. Kobeissi, G. Nicolas, and K. Bhargavan, “Noise explorer: Fully automated modeling and verification for arbitrary noise protocols,” in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 356–370.

[32] B. Blanchet, “Symbolic and computational mechanized verification of the arinc823 avionic protocols,” in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 2017, pp. 68–82.

[33] V. Cortier, D. Galindo, and M. Turuani, “A formal analysis of the neuchâtel e-voting protocol,” in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 430–442.

[34] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, “Proverif 2.04: automatic cryptographic protocol verifier, user manual and tutorial,” *Version from*, 2021.

[35] M. Abadi and V. Cortier, “Deciding knowledge in security protocols under equational theories,” *Theoretical Computer Science*, vol. 367, no. 1, pp. 2–32, 2006.

[36] V. Cheval, V. Cortier, and M. Turuani, “A little more conversation, a little less action, a lot more satisfaction: Global states in ProVerif,” in *IEEE Computer Security Foundations Symposium (CSF)*, Jul. 2018, pp. 344–358.

[37] K. Bhargavan, V. Cheval, and C. Wood, “A symbolic analysis of privacy for tls 1.3 with encrypted client hello,” in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS’22)*. Los Angeles, USA: ACM Press, Nov. 2022.