

ProVerif with Lemmas, Induction, Fast Subsumption, and Much More

Bruno Blanchet, Vincent Cheval, Véronique Cortier

▶ **To cite this version:**

Bruno Blanchet, Vincent Cheval, Véronique Cortier. ProVerif with Lemmas, Induction, Fast Subsumption, and Much More. S&P 2022 - 43rd IEEE Symposium on Security and Privacy, May 2022, San Francisco, United States. hal-03366962

HAL Id: hal-03366962
https://inria.hal.science/hal-03366962

Submitted on 7 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ProVerif with Lemmas, Induction, Fast Subsumption, and Much More

Bruno Blanchet
Inria Paris
F-75012 Paris, France

Vincent Cheval
Inria Paris
F-75012 Paris, France

Véronique Cortier
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

***Abstract*—This paper presents a major overhaul of one the most widely used symbolic security protocol verifiers, ProVerif. We provide two main contributions. First, we extend ProVerif with lemmas, axioms, proofs by induction, natural numbers, and temporal queries. These features not only extend the scope of ProVerif, but can also be used to improve its precision (that is, avoid false attacks) and make it terminate more often. Second, we rework and optimize many of the algorithms used in ProVerif (generation of clauses, resolution, subsumption, ...), resulting in impressive speed-ups on large examples.**

1. Introduction

Security protocols aim at securing communications. They are used in various applications: establishment of secure channels over the Internet, secure messaging, electronic voting, mobile communications, etc. Their design is known to be error prone and flaws are difficult to fix once a protocol is largely deployed. Hence a common practice is to analyze the security of a protocol using formal techniques and in particular automatic tools. For example, TLS 1.3 has been designed while research groups were developing formal models in parallel and suggesting modifications [4].

Several tools have been proposed for automatized security analysis of protocols. Some tools focus at restricted classes of protocols for which the analysis is deemed to terminate. They typically focus on a bounded number of sessions like Avispa [3], DeepSec [15], or Akiss [12]. 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 [31] and ProVerif [6] 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.

ProVerif has been developed for 20 years and has been used to analyze hundreds of protocols, including major deployed protocols such as TLS [4], Signal [25], Noise [26], [29], the avionic protocol Arinc823 [9], and the Neuchâtel voting protocol [17]. ProVerif is taught in several universities (in specialized Masters) and summer schools. The tool can analyze a large class of security properties, either specified as correspondence properties (for example requesting that an event occurs before another one) or as equivalence properties, which state that an attacker should not be able to distinguish between two scenarios. Correspondence properties can be used to specify authentication, consistent views between parties, or verifiability properties, while equivalence properties are often used to specify privacy properties like anonymity, non traceability, or vote privacy. Given a protocol and a security property, ProVerif may either prove that the property is satisfied or exhibit an attack. It may also return "cannot be proved" meaning that it can not reach a conclusion. Finally, it may be that ProVerif is not efficient enough to conclude in a reasonable amount of time, or that ProVerif does not terminate at all.

Our contributions. We have carried out a major overhaul of ProVerif, improving its precision, its efficiency, its expressiveness, and introducing some level of interactivity by allowing users to declare intermediate properties helping ProVerif to complete proofs. In more details, our contributions can be summarized as follows:

- support for axioms, lemmas, and restrictions as in Tamarin, in order to obtain the best of the two tools: a high level of automation as well as the possibility to interact with the tool. Lemmas specify intermediate properties meant to help the proof. Axioms are similar to lemmas but do not need to be proved since they are typically guaranteed by other means (e.g. proof by hand). Restrictions are a convenient modeling technique to exclude behaviors that cannot occur in practice (e.g. concurrent access to a lock state).
- improved precision: ProVerif returns "cannot be proved" less often. We introduce a `precise` option that automatically generates (sound) axioms that refine the abstractions made by ProVerif when analyzing a protocol. This helps to conclude that a protocol is either secure or has an attack.
- support for natural numbers together with addition (between integers and at most one variable) and comparison. Natural numbers can be used to specify protocols with counters or can model time evolution.
- support for temporal queries. A query is a security property that ProVerif should prove. Temporal queries are queries in which some events are proved to happen before others, such as $\mathsf{event}(Counter(c_1))@t_1 \wedge \mathsf{event}(Counter(c_2))@t_2 \Rightarrow t_1 > t_2 \vee c_1 \leq c_2$ where

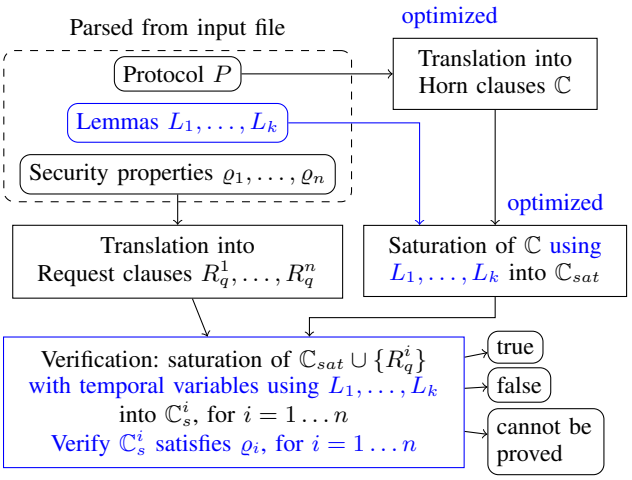
we specify that the counter can only increase; the fact $\mathsf{event}(Counter(c))@t$ means that the counter has value c at time t.
- better treatment of injective queries, that is, queries where the occurence of some event (for example, the delivery of some product) can be associated injectively to another event (for example a payment). The injective property ensures here that there are at least as many payments as the number of deliveries. Such queries previously often yield a "cannot be proved".
- major speed improvements. The new ProVerif typically runs 30-40 times faster than ProVerif 2.00 on large protocols and up to exponentially faster on some examples.

As a result, ProVerif can now support more protocols, in particular protocols with global states, for both reachability and equivalence. Global states include counters, cells, tables and are typically difficult to handle for ProVerif due to its internal abstractions. A tool GSVerif was introduced [14] (in the context of reachability properties) to automatically generate properties that are proved to hold but cannot be proved by ProVerif. The properties generated by GSVerif can now be stated as axioms, and used at an earlier stage of the procedure, yielding more successful proofs. Moreover, our approach now also applies to equivalence properties. Hence ProVerif can automatically prove equivalence properties for protocols with global states, such as vote privacy in voting protocols that require to maintain a table of all received votes.

Moreover, our experiments show that ProVerif can now prove or disprove (that is, exhibit an attack) in many more protocols of the literature. In terms of efficiency, the analysis of 42 protocols from the Noise Protocol Framework took more than 170h and now takes 20min, hence is at least 516 times faster. The analysis of the Neuchâtel voting protocol is parameterized by the number k of voting options. For $k \geq 3$, the verification of ballot privacy took more than 24h and now takes 3s for $k = 3$ and 4h36min for $k = 6$. Our changes have been integrated in the official distribution of ProVerif (release 2.02pl1 available at https://proverif.inria.fr).

A new procedure. All these enhancements and improvements of ProVerif have been obtained through a major rewrite of the internal procedure of the tool, detailed in Section 3. Let us overview the procedure of ProVerif, as summarized in Figure 1. ProVerif first translates protocols into a set $\mathbb{C}$ of Horn clauses, a subclass of first order logic. Then, it saturates the clauses by resolution, yielding a simpler set of clauses $\mathbb{C}_{sat}$ that derive the same facts. If this saturation procedure terminates, then ProVerif verifies the security property by saturating again $\mathbb{C}_{sat}$ with the request clause R obtained by translating the security property, and verifying that the obtained clauses satisfy the property. The correctness of ProVerif ensures that, if this verification succeeds, then the initial property holds. Otherwise, either ProVerif can reconstruct an attack against the initial protocol following the corresponding clause derivation, or ProVerif cannot conclude and returns "cannot be proved".

It is not easy in this context to add lemmas since they



Parts in blue indicate the novelties introduced in this paper.

Figure 1. Overview of the ProVerif procedure.

cannot be easily interpreted by ProVerif as an "help". Instead, we modified the internal saturation procedure of ProVerif to refine clauses. Intuitively, given an already proved lemma $A \Rightarrow B$, a clause $H \rightarrow C$ can be replaced by $H \wedge B \rightarrow C$ as soon as H entails A. This yields a more precise clause, possibly helping termination. This is sound as soon as we only use lemmas that have been already proved. However, we also allow a lemma L_i to be proved by induction on the length of the execution trace. We can prove that the first saturation procedure remains sound for such induction thanks to the invariant that facts in hypotheses of clauses always happen before facts in the conclusion, and strictly before for facts that occur in lemmas. However, in the verification step, this property is lost. Therefore, we have entirely revisited the verification procedure, to keep track that certain facts happen (strictly) before others, so that a lemma proved by induction can be used to refine a clause only when the ordering is compatible (hence guaranteeing soundness). More precisely, facts are now annotated with temporal variables and clauses include ordering constraints on these variables. We provide a sound procedure to resolve such clauses. Thanks to these temporal variables, it is then easy to support temporal queries.

The introduction of natural numbers with addition and comparison follows (and generalizes) the approach initiated in [14]: our saturation procedure relies on the Pratt algorithm [30] to solve inequality constraints that appear in clauses.

While revisiting the core procedure of ProVerif, we have considerably improved its efficiency at several steps of the algorithm, as detailed in Section 4. For example, clause generation has been turned into a more lazy approach in order to generate fewer clauses. Moreover, we have introduced techniques from automated deduction [32] to speed up checking whether a clause is more general than another. We have also optimized the detection and removal of redundant clauses.

We prove the correctness of the new procedure, for the entire syntax and semantics of ProVerif. We cover

M, N ::=	terms
x	variable ($x \in \mathcal{V}$)
n	name ($n \in \mathcal{N}$)
$f(M_1, \ldots, M_k)$	applied $f \in \mathcal{F}_c$
ev ::=	events
$e(M_1, \ldots, M_k)$	($e \in \mathcal{F}_e$)
D ::=	expressions
M	term
$h(D_1, \ldots, D_k)$	applied $h \in \mathcal{F}_d \cup \mathcal{F}_c$
fail	failure
P, Q ::=	processes
0	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x); P$	input
$P \mid Q$	parallel composition
$!P$	replication
new $a; P$	restriction
let $x = D$ in P else Q	assignment
$\mathsf{event}(ev); P$	event

Figure 2. Syntax of the core language of ProVerif.

optimizations and features that were never formally defined in previous papers. For instance, we generalize the definition of correspondence queries, which were previously defined only with events in their conclusion. The full proof can be found in a technical report [11].

2. Syntax and semantics

The syntax and semantics of ProVerif is mostly unchanged. In terms of syntax, there are only two additions. First, messages can now contain natural numbers, which is useful for example to model counters or to specify finer properties. Secondly, we introduce lemmas, axioms, and restrictions, which are simply a particular form of queries. We introduce here the main constructs of the syntax and the semantics. A complete specification can be found in [11].

2.1. Syntax and informal semantics

The syntax for *terms*, *events*, *expressions*, and *processes* is displayed in Figure 2. ProVerif actually also supports tables and phases but they are omitted here for simplicity. Our results hold for the full language of ProVerif.

We assume a set of variables $\mathcal{V}$ and a set of names $\mathcal{N}$. We consider a finite signature Σ of function symbols with their arity (i.e. their number of arguments) partitioned into three sets $\mathcal{F}_c$, $\mathcal{F}_d$, and $\mathcal{F}_e$ representing respectively the *constructor*, *destructor*, and *event* function symbols.

Terms represent data, such as messages, and can be built as a variable, a name, or the application of a constructor function symbol to terms.

Destructor function symbols can manipulate terms and only appear in expressions. They represent functions from terms to terms defined by rewrite rules. More specifically, the behavior of a destructor function symbol g is defined by an ordered list $\mathsf{def}(g)$ of rewrite rules of the form $g(U_1, \ldots, U_n) \to U$ where $U_1, \ldots, U_n, U$ are either terms M or the constant fail.

As usual, we say that a term, expression, ... is *ground* when it contains no variable. A *value* V is either a ground term or fail. Expressions represent computations on values. Their semantics is defined by an evaluation relation $D \Downarrow V$, which means that the ground expression D evaluates to the value V. We evaluate a ground expression D by rewriting it as follows, until we obtain the value V. Each subexpression $g(V_1, \ldots, V_n)$, where g is a destructor and $V_1, \ldots, V_n$ are values, is rewritten by trying the rewrite rules of g in the order given in the list $\mathsf{def}(g)$, and applying the first applicable rewrite rule. When no rewrite rule of $\mathsf{def}(g)$ can be applied, $g(V_1, \ldots, V_n)$ rewrites to fail. Furthermore, when f is a constructor and $V_i = \mathsf{fail}$ for some i, $f(V_1, \ldots, V_n)$ rewrites to fail. A more formal definition of $D \Downarrow V$ is given in Appendix A and examples are given in Examples 1 and 2.

We use the constructors $true$ and $false$ for boolean constants. We also consider constructor symbols for tuples of any arity that the attacker can always deconstruct. Thus, for all tuple constructors f of arity n, we assume the existence of a destructor π_i^f, for $i = 1 \ldots n$, that is the i-th projection of f. Formally, $\mathsf{def}(\pi_i^f) = [\pi_i^f(f(x_1, \ldots, x_n)) \to x_i]$. We omit f for canonical tuples.

Example 1. *The standard asymmetric encryption primitives can be modeled by considering a constructor $aenc$ of arity 3, taking as arguments the cleartext, some randomness, and the public encryption key; a constructor pk of arity 1, which maps secret keys to public keys; and a destructor $adec$ of arity 2 with the following rewrite rule:* $\mathsf{def}(adec) = [adec(aenc(x, y, pk(z)), z) \to x]$. *The evaluation of the ground expression $adec(aenc((a, b), r, pk(k)), k)$ applies the rewrite rule, yielding $adec(aenc((a, b), r, pk(k)), k) \Downarrow (a, b)$. The function symbol $adec$ needs to be a destructor for this evaluation to happen; if it were a constructor, the term $adec(aenc((a, b), r, pk(k)), k)$ would remain unchanged and be different from (a, b).*

Similarly, signatures can be modeled with the rule $\mathsf{def}(checksign) = [checksign(sign(x, y), vk(y)) \to x]$. *This rule combines the verification of the signature and the retrieval of the message whose signature has been checked.*

Example 2. *Defining the behavior of a destructor with a sequence of rewrite rules allows a simple definition of the "if then else" construction. Consider the destructor $ifelse$ of arity 3 such that $ifelse(x, y, z)$ should be rewritten in y when x is $true$, and in z otherwise. This can be expressed in our formalism with the sequence of rewrite rules* $\mathsf{def}(ifelse) = [ifelse(true, y, z) \to y; ifelse(x, y, z) \to z]$. *For instance, $ifelse(false, a, b) \Downarrow b$ because the first rewrite rule cannot be applied and the second one can.*

Event function symbols $e \in \mathcal{F}_e$ can be applied to terms

to build events $e(M_1, \ldots, M_n)$. Events can be executed by processes with the construct $\mathsf{event}(e(M_1, \ldots, M_n)); P$. Events record that a certain program point has been reached in the process, with certain values of their arguments, but otherwise do not affect the execution of the process. They are used to express correspondence properties. The other constructs in processes are standard. The process $\mathsf{in}(N, x); P$ models the input on the channel N of a term that is bound to x when executing P. The process $\mathsf{out}(N, M); P$ represents the output of the term M on the channel N. The process $P \mid Q$ models the concurrent execution of P and Q. The replication $!P$ represents an unbounded number of copies of P. The restriction $\mathsf{new}\ a; P$ generates a fresh name a that can be used in P. Finally, the construct $\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ evaluates the expression D, executing P with x bound to M when D evaluates to a term M, and otherwise executing Q.

Natural numbers. Following the work of [14], we also consider natural numbers using the Peano representation. We consider $zero/0$ and $succ/1$ two function symbols in $\mathcal{F}_c$, and $minus_one/1$ a function symbol in $\mathcal{F}_d$. For simplicity, we denote by $x + n$ the term $succ^n(x)$, and we denote by $n \in \mathbb{N}$ the term $succ^n(zero)$. We say that a ground term M is a natural number if $M = n$ for some $n \in \mathbb{N}$. The destructor $minus_one$ represents the subtraction by 1 defined by $\mathsf{def}(minus_one) = [minus_one(succ(x)) \to x]$.

Moreover, we consider two special additional functions $nat/1$ and $geq/2$ whose evaluation on ground expressions is defined as follows:

- $nat(D) \Downarrow true$ (resp. $false$) if $D \Downarrow M \in \mathbb{N}$ (resp. $M \notin \mathbb{N}$). Otherwise $nat(D) \Downarrow \mathsf{fail}$.
- $geq(D_1, D_2) \Downarrow true$ (resp. $false$) if for $i = 1, 2$, $D_i \Downarrow M_i \in \mathbb{N}$ and $M_1 \geq M_2$ (resp. $M_1 < M_2$). Otherwise $geq(D_1, D_2) \Downarrow \mathsf{fail}$.

The representation of natural numbers using $zero$ and $succ$ is obviously inefficient for large numbers. In practice, this is not problematic in the examples we consider because, even if counters may take large values during the execution of a protocol, the constants that occur in the protocol are typically small. It would obviously be possible to modify the tool to use a more efficient representation of natural numbers if needed.

2.2. Formal semantics

A *configuration* $\mathcal{E}, \mathcal{P}, \mathcal{A}$ consists of a multiset $\mathcal{P}$ of processes, representing the current state of the process, a set of names $\mathcal{E}$ representing the free names of $\mathcal{P}$ and the names created by the adversary, and a set $\mathcal{A}$ of terms known by the adversary. The semantics of processes is defined through a reduction relation $\xrightarrow{\ell}$ between configurations, where ℓ is either the empty label or a label of the form $\mathrm{msg}(N, M)$ or $\mathrm{event}(ev)$ with N, M being terms and ev being an event. An initial configuration is a configuration of the form $\mathcal{E}, \{\!|P|\!\}, \mathcal{A}$, also denoted $\mathcal{E}, P, \mathcal{A}$ for simplicity, where $\mathcal{A}$ contains only names and $\mathcal{E}$ is the union of $\mathcal{A}$ with the free names of P.

As usual, we define substitutions as functions from variables to terms, and the application of a substitution σ to a process Q is denoted $Q\sigma$. We write $\{^M/_x\}$ for the substitution that maps x to M.

The semantics is defined as usual and is provided in Appendix A. For example, an adversary can send any message of her knowledge through the IN rule:

$$\mathcal{E}, \mathcal{P} \cup \{\!|\mathsf{in}(N, x); Q|\!\}, \mathcal{A} \xrightarrow{\mathrm{msg}(N,M)} \mathcal{E}, \mathcal{P} \cup \{\!|Q\{^M/_x\}|\!\}, \mathcal{A}$$

if $N, M \in \mathcal{A}$. The attacker may enrich her knowledge by generating fresh names (NEW rule) or by applying function symbols through the APP rule:

$$\mathcal{E}, \mathcal{P}, \mathcal{A} \to \mathcal{E}, \mathcal{P}, \mathcal{A} \cup \{M\}$$

if $M_1, \ldots, M_n \in \mathcal{A}$, $h/n \in \mathcal{F}_c \cup \mathcal{F}_d$ and $h(M_1, \ldots, M_n) \Downarrow M$. The attacker may also learn outputted messages:

$$\mathcal{E}, \mathcal{P} \cup \{\!|\mathsf{out}(N, M); P|\!\}, \mathcal{A} \xrightarrow{\mathrm{msg}(N,M)} \mathcal{E}, \mathcal{P} \cup \{\!|P|\!\}, \mathcal{A} \cup \{M\}$$

if $N \in \mathcal{A}$. Finally, events simply annotate the trace:

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

A trace T of a configuration $\mathcal{C}_0$ is a finite sequence $\mathcal{C}_0 \xrightarrow{\ell_1} \mathcal{C}_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_n} \mathcal{C}_n$. A *step* of T is an integer τ such that $1 \leq \tau \leq n$. Then the configuration at step τ in trace T is the configuration $T[\tau] = \mathcal{C}_\tau$. We denote $\mathsf{trace}(\mathcal{C})$ the set of traces from $\mathcal{C}$.

Example 3. *Belenios [18] is a simple voting protocol used in more than 1400 elections in 2020. Each voter sends her vote encrypted with the public key of the election, and signed with a credential. The encrypted vote is then published on a bulletin board. At the end of the election, ballots are shuffled (using mixnets) and votes are published in a random order. Other variants of Belenios exist with homomorphic encryption but we present here only a simplified version, omitting for example the zero-knowledge proofs of correct decryption.*

The behavior of a voter can be represented by the following process:

$$\begin{array}{l} V(vote, sk) = \mathsf{in}(c, x_r); \mathsf{new}\ r; \\ \quad \mathsf{out}(c, sign(aenc(vote, (r, x_r), pk(sk_e)), sk)) \end{array}$$

This process models that the voting client uses both its own randomness r and some external randomness x_r (typically from the server) to encrypt the vote.

For simplicity, we model the voting server together with the tally phase. The server receives votes, checks the validity of signatures and once the election is over, publishes the decrypted ballots in a random order.

$$\begin{array}{l} Board = \\ \quad \mathsf{in}(c, x_1); \mathsf{let}\ y_1 = checksign(x_1, vk(sk_a))\ \mathsf{in} \\ \quad \mathsf{in}(c, x_2); \mathsf{let}\ y_2 = checksign(x_2, vk(sk_b))\ \mathsf{in} \\ \quad \mathsf{in}(c, x_3); \mathsf{let}\ y_3 = checksign(x_3, vk(sk_c))\ \mathsf{in} \\ \qquad \mathsf{out}(c, adec(y_1, sk_e)) \\ \qquad \mid \mathsf{out}(c, adec(y_2, sk_e)) \\ \qquad \mid \mathsf{out}(c, adec(y_3, sk_e)) \end{array}$$

The fact that the decrypted ballots are sent in a random order is modeled here by considering three outputs in parallel: the adversary cannot distinguish the order of the outputs.

Then the process for modeling Belenios altogether is:

$$P_{Bel} = V(v_a, sk_a) \mid V(v_b, sk_b) \mid Board \mid Setup$$

where Setup is defined as:

$$\mathsf{out}(c, vk(sk_a)) \mid \mathsf{out}(c, vk(sk_b)) \mid \mathsf{out}(c, pk(sk_e))$$

P_{Bel} models a system with two honest voters, Alice and Bob with secret keys sk_a and sk_b respectively, and a dishonest one, whose secret key is sk_c. The initial configuration is $\mathcal{C}_0 = \mathcal{E}_0, P_{Bel}, \mathcal{A}_0$ with $\mathcal{E}_0 = \{c, sk_a, sk_b, sk_c, sk_e, v_a, v_b\}$ and $\mathcal{A}_0 = \{c, sk_c, v_a, v_b\}$ modeling the fact that the key sk_c is known to the attacker while the other ones are initially secret. The vote values v_a, v_b are also given to the attacker.

Then a possible execution trace is

$$\mathcal{C}_0 \xrightarrow{\mathrm{msg}(c,vk(sk_a))} \xrightarrow{\mathrm{msg}(c,vk(sk_b))} \xrightarrow{\mathrm{msg}(c,pk(sk_e))} \mathcal{C}_1 \xrightarrow{\mathrm{msg}(c,r'_a)} \xrightarrow{\mathrm{msg}(c,M_a)} \xrightarrow{\mathrm{msg}(c,r'_b)} \xrightarrow{\mathrm{msg}(c,M_b)} \mathcal{C}_2$$

where we omit steps with no labels and $M_\alpha = sign(aenc(v_\alpha, (r_\alpha, r'_\alpha), pk(sk_e)), sk_\alpha)$, $\alpha \in \{a, b\}$ and $\mathcal{C}_2 = \mathcal{E}_2, P_2, \mathcal{A}_2$ with $\mathcal{E}_2 = \mathcal{E}_0 \cup \{r_a, r'_a, r_b, r'_b\}$, $\mathcal{A}_2 = \mathcal{A}_0 \cup \{vk(sk_a), vk(sk_b), pk(sk_e), r'_a, M_a, r'_b, M_b\}$, and $P_2 = Board$. This trace corresponds to the emission of the initial messages of the Setup process, followed by the execution of the processes of the two honest voters.

Then more interestingly, instead of casting a standard ballot, the attacker may copy Alice's ballot and sends it on behalf of the dishonest voter. This behavior is reflected by the trace $\mathcal{C}_2 \xrightarrow{\mathrm{msg}(c,M_a)} \xrightarrow{\mathrm{msg}(c,M_b)} \xrightarrow{\mathrm{msg}(c,M_c)} \mathcal{C}_3$. The board first receives the two honest ballots, and then the adversarial one: $sign(checksign(M_a, vk(sk_a)), sk_c) \Downarrow sign(aenc(v_1, (r_a, r'_a), pk(k))), sk_c) = M_c$.

As noticed in [19] in the context of the Helios protocol, this attack yields a privacy attack. Indeed, the outcome of the election will be $\{v_a, v_a, v_b\}$, hence the attacker can deduce that Alice voted v_a.

To avoid this attack, the bulletin board should never accept two identical encrypted votes. This can be modeled in the process Board by checking that y_1, y_2, and y_3 are pairwise distinct, yielding a process Board′.

2.3. Security properties

ProVerif includes two main ways to express security properties: correspondence and equivalence properties.

2.3.1. Correspondence properties. To express correspondence properties, we consider *atomic formulas*, defined as *facts* or *test formulas*, whose syntax is given by the following grammar:

F ::=	fact
$\mathrm{event}(ev)$	event ev is executed
$\mathrm{att}(M)$	attacker knows M
$\mathrm{msg}(M, N)$	message N was sent on channel M

ϕ ::=	test formula
$M = N$	equality
$M \neq N$	disequality
$M \geq N$	inequality
$\mathrm{isnat}(M)$	M is a natural number
$\neg\mathrm{isnat}(M)$	M is not a natural number

Given a trace T, the satisfaction relation $T, \tau \vdash \psi$ means that the trace T satisfies the formula ψ at step τ. We define the satisfaction relation on ground facts as follows:

- $T, \tau \vdash \mathrm{att}(M)$ if and only if $T[\tau] \rightarrow^* \mathcal{E}, \mathcal{P}, \mathcal{A}$ by only the rules APP and NEW such that $M \in \mathcal{A}$.
- $T, \tau \vdash \mathrm{event}(ev)$ if and only if $T[\tau - 1] \xrightarrow{\mathrm{event}(ev)} T[\tau]$.
- $T, \tau \vdash \mathrm{msg}(N, M)$ if and only if $T[\tau - 1] \xrightarrow{\mathrm{msg}(N,M)} T[\tau]$ with $T[\tau] = \mathcal{E}, \mathcal{P}, \mathcal{A}$.

The satisfaction of a test formula ϕ is in fact independent of T, τ and is defined as expected.

Note that $T, \tau \vdash \mathrm{att}(M)$ holds even if M is not in the attacker's knowledge $\mathcal{A}$ of the configuration $T[\tau]$. Indeed, we wish $T, \tau \vdash \mathrm{att}(M)$ to hold as soon as the attacker *may deduce* M and even if M is not already explicitly in his knowledge. This is why we consider any evolution $T[\tau] \rightarrow^* \mathcal{E}, \mathcal{P}, \mathcal{A}$ that only uses the attacker rules function application (APP) and nonce generation (NEW).

A *correspondence query* ϱ is a formula of the form $F_1 \wedge \ldots \wedge F_n \Rightarrow \psi$ where:

$$\psi, \psi' ::= \top \mid \bot \mid F \mid \phi \mid \psi \wedge \psi' \mid \psi \vee \psi'$$

Intuitively, a trace T satisfies the correspondence query ϱ ($T \vdash \varrho$) if whenever T satisfies an instance of F_i at τ_i for each $1 \leq i \leq n$, then T also satisfies ψ where each satisfied atomic formula in ψ is satisfied at some step $\tau \leq \max_i(\tau_i)$.

We write $\mathcal{T} \vdash \varrho$ when all traces T of the set $\mathcal{T}$ satisfy the query, that is, $T \vdash \varrho$ for all $T \in \mathcal{T}$. Then a configuration $\mathcal{C}$ satisfies ϱ when $\mathsf{trace}(\mathcal{C}) \vdash \varrho$. A process P_0 satisfies ϱ when all traces of P_0 satisfy ϱ.

Note that the secrecy of a ground term M can be expressed as the correspondence query $\mathrm{att}(M) \Rightarrow \bot$.

ProVerif also includes *nested queries*, in which ψ may itself contain correspondence queries $\mathrm{event}(ev) \rightsquigarrow \psi$, and injective events $\mathrm{inj\text{-}event}(ev)$ but we omit them here. Their formal definition is provided in [11].

Example 4. *In the context of voting protocols, correspondence queries can be used for example to specify verifiability properties. Continuing Example 3, we may express that whenever a vote is registered in the name of an honest voter Alice, then the content of the ballot indeed corresponds to Alice's intention. For this, we add events in the specification of the process Board as follows.*

$Board$=
  $\mathsf{in}(c, x_1); \mathsf{let}\ y_1 = checksign(x_1, vk(sk_a))\ \mathsf{in}$
    $\mathsf{event}(Record(y_1, vk(sk_a)));$

$\mathsf{in}(c, x_2)$; let $y_2 = \mathit{checksign}(x_2, \mathit{vk}(\mathit{sk}_b))$ in
$\quad \mathsf{event}(\mathit{Record}(y_2, \mathit{vk}(\mathit{sk}_b)))$;
$\mathsf{in}(c, x_3); \cdots$

and the rest of the Board process is left unchanged. Similarly, the event $\mathit{Voted}(\mathit{vote}, \mathit{vk}(\mathit{sk}))$ *is added at the beginning of the process* $V(\mathit{vote}, \mathit{sk})$.

Then we can request that any recorded ballot corresponds to a cast vote, for the same (honest) voter:

$$\mathrm{event}(\mathit{Record}(x_b, x_{vk})) \Rightarrow \\ \quad \mathrm{event}(\mathit{Voted}(x_v, x_{vk})) \wedge x_b = \mathit{aenc}(x_v, x_r, \mathit{pk}(k))$$

2.3.2. Temporal correspondence properties. One of our contributions is the introduction of temporal properties. Traditional correspondence queries only allow to order the facts of the premise of the query w.r.t. the facts of its conclusion. *Nested queries* permit a limited comparison between facts in the conclusion of the query. Temporal queries generalize correspondence queries by allowing to order facts occurring anywhere in the query.

Facts F in correspondence queries can now be labeled with temporal variables t, to denote that F is satisfied by the trace at the step t. Such facts are written $F@t$ and called *temporal facts*, *e.g.* $\mathrm{event}(\mathit{Voted}(x_v, x_{vk}))@t$. Temporal variables can only be instantiated by natural numbers and thus can be compared in the query using equalities, disequalities, and inequalities. Note that even though temporal variables are instantiated by natural numbers, they should not be mixed with terms in the query and we only allow a temporal variable to be compared with another temporal variable (*e.g.* $t + 3 \geq t'$, $t > 5$ and $\mathrm{event}(\mathit{Counter}(t))@t$ are not valid atomic formulae when t, t' are temporal variables).

The satisfaction relation $\vdash$ is extended as expected to ground temporal facts and temporal queries. For example, $T, \tau \vdash \mathsf{event}(ev)@\tau'$ if and only if $T[\tau - 1] \xrightarrow{\mathrm{event}(ev)} T[\tau]$ and $\tau = \tau'$.

We do not impose restrictions on comparisons of temporal variables associated to events. However, an attacker or message temporal fact can only be requested to occur before another temporal fact of the query.

2.3.3. Equivalence properties. Privacy properties are often modeled using equivalence properties. In particular, observational equivalence, a weak bisimulation stable by evaluation context, intuitively guarantees that an attacker cannot see any difference between observationally equivalent processes. As shown in [10], ProVerif can prove equivalence between two processes P_1 and P_2 that differ only by the terms and expressions they contain. To do so, ProVerif represents P_1 and P_2 as a single process P, called *biprocess*. The grammar of biprocesses is the same as in Figure 2 with the addition of $\mathsf{diff}[M, M']$ for terms and $\mathsf{diff}[D, D']$ for expressions. Given a biprocess P, we define $\mathsf{fst}(P)$ (resp. $\mathsf{snd}(P)$) as the process obtained from P by replacing all instances of $\mathsf{diff}[M, M']$ with M (resp. M') and all instances of $\mathsf{diff}[D, D']$ with D (resp. D'). For instance, the processes $P_1 = \mathsf{out}(c, a)$ and $P_2 = \mathsf{out}(c, b)$ are represented by the biprocess $P = \mathsf{out}(c, \mathsf{diff}[a, b])$, such that $P_1 = \mathsf{fst}(P)$ and $P_2 = \mathsf{snd}(P)$.

ProVerif proves observational equivalence between $P_1 = \mathsf{fst}(P)$ and $P_2 = \mathsf{snd}(P)$ by proving a trace property on the biprocess P as we shall explain here.

The semantics of biprocesses is defined by a relation $\rightarrow_b$ and is an easy adaptation of the relation $\rightarrow$: a biprocess can take a step only if both the corresponding left and right processes can take the same step. For example,

$$\mathcal{E}, \mathcal{P} \cup \{\!|\mathsf{out}(N, M); P, \mathsf{in}(N', x); Q|\!\}, \mathcal{A} \\ \xrightarrow{\mathrm{msg}(N,M)}_b \mathcal{E}, \mathcal{P} \cup \{\!|P, Q\{^M/_x\}|\!\}, \mathcal{A}$$

if $\mathsf{fst}(N) = \mathsf{fst}(N')$ and $\mathsf{snd}(N) = \mathsf{snd}(N')$.

Similarly, an else branch is executed if both left and right processes execute the else branch:

$$\mathcal{E}, \mathcal{P} \cup \{\!|\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q|\!\}, \mathcal{A} \rightarrow_b \mathcal{E}, \mathcal{P} \cup \{\!|Q|\!\}, \mathcal{A}$$

if $\mathsf{fst}(D) \Downarrow \mathsf{fail}$ and $\mathsf{snd}(D) \Downarrow \mathsf{fail}$.

A trace of a biprocess in this semantics is called a *bitrace*. We recall the notion of *convergent bitrace*, which allows to prove observational equivalence.

Definition 1. *A bitrace* T converges*, denoted* $T{\downarrow\uparrow}$*, when for all steps* τ*,* $T[\tau] = \mathcal{E}, \mathcal{P}, \mathcal{A}$ *implies:*

- *if* $\mathcal{P} = \mathcal{P}' \cup \{\!|\mathsf{out}(N, M); P, \mathsf{in}(N', x); Q|\!\}$*,*
  or $\mathcal{P} = \mathcal{P}' \cup \{\!|\mathsf{out}(N, M); P|\!\}$ *and* $N' \in \mathcal{A}$*,*
  or $\mathcal{P} = \mathcal{P}' \cup \{\!|\mathsf{in}(N, x); P|\!\}$ *and* $N' \in \mathcal{A}$ *then*
  $\mathsf{fst}(N) = \mathsf{fst}(N')$ *iff* $\mathsf{snd}(N) = \mathsf{snd}(N')$.
  Any communication that can be done by the left process can also be done by the right one, and conversely.
- *if* $M_1, \ldots, M_n \in \mathcal{A}$ *and* $g \in \mathcal{F}_d$ *then*
  $\mathsf{fst}(g(M_1, \ldots, M_n)) \Downarrow \mathsf{fail}$
  iff $\mathsf{snd}(g(M_1, \ldots, M_n)) \Downarrow \mathsf{fail}$.
  The attacker does not observe any difference between the left and right sides. This is similar to static equivalence as defined in [1].
- *if* $\mathcal{P} = \mathcal{P}' \cup \{\!|\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q|\!\}$ *then* $\mathsf{fst}(D) \Downarrow \mathsf{fail}$ *iff* $\mathsf{snd}(D) \Downarrow \mathsf{fail}$*. The left process cannot take the "else" branch if the right process takes the "in" branch, and conversely.*

We extend this notion to sets of bitraces as expected, i.e. $\mathcal{T}{\downarrow\uparrow}$ *if and only if for all* $T \in \mathcal{T}$*,* $T{\downarrow\uparrow}$.

Bitrace convergence implies observational equivalence.

Proposition 1 ([7, Theorem 3.5])**.** *For all initial biconfigurations* $\mathcal{C}$*, if* $\mathsf{trace}(\mathcal{C}) {\downarrow\uparrow}$ *then* $\mathsf{fst}(\mathcal{C})$ *and* $\mathsf{snd}(\mathcal{C})$ *are observationally equivalent (as defined in [7, Definition 3.6]).*

Example 5. *Vote privacy is usually expressed as an equivalence property [20]: an attacker that observes Alice voting* v_a *and Bob voting* v_b*, should not see any difference with a scenario where Alice is voting* v_b *and Bob is voting* v_a*.*

Continuing Example 3, this is formalized as the observational equivalence of

$$V(v_a, \mathit{sk}_a) \mid V(v_b, \mathit{sk}_b) \mid \mathit{Board}' \mid \mathit{Setup} \\ \textit{and } V(v_b, \mathit{sk}_a) \mid V(v_a, \mathit{sk}_b) \mid \mathit{Board}' \mid \mathit{Setup}\,.$$

By Proposition 1, this can be proved by showing bitrace convergence of the process P_{diff} defined as:

$$V(\mathsf{diff}[v_a, v_b], sk_a) \mid V(\mathsf{diff}[v_b, v_a], sk_b) \mid Board'' \mid Setup$$

with $Board''$ being the process $Board'$ where we swap the output of the votes:

$Board''$=
$\mathsf{in}(c, x_1); \mathsf{let}\ y_1 = checksign(x_1, vk(sk_a))\ \mathsf{in}$
...
$\mathsf{out}(c, \mathsf{diff}[adec(y_1, sk_e), adec(y_2, sk_e)])$
$\mid \mathsf{out}(c, \mathsf{diff}[adec(y_2, sk_e), adec(y_1, sk_e)])$
$\mid \mathsf{out}(c, adec(y_3, sk_e))$

The output of the votes being in parallel composition, the swap preserves observational equivalence and is in fact the basis of an automatic transformation of processes in [13] to help ProVerif prove equivalence.

2.3.4. Correspondence queries on bitraces. We introduce the notion of correspondence queries on convergent bitraces. They may be used to help prove equivalence by the means of lemmas or axioms, or help prove a lemma by the means of other lemmas or axioms.

A correspondence query on convergent bitraces is the same as a correspondence query on standard traces except that facts are replaced by bifacts. Formally, we consider the algebra for queries defined in Section 2.3.1 where we replace:

- events $\mathrm{event}(e(M_1, \ldots, M_n))$ by $\mathrm{event}'(e(M_1, \ldots, M_n), e(M'_1, \ldots, M'_n))$;
- facts $\mathrm{att}(M)$ by $\mathrm{att}'(M, M')$;
- facts $\mathrm{msg}(M, N)$ by $\mathrm{msg}'(M, N, M', N')$.

The non-primed arguments deal with the left process, while the primed ones deal with the right process. We adapt the satisfaction relation $\vdash$ accordingly.

Then a configuration $\mathcal{C}$ satisfies ϱ if all its bitraces satisfy the query, that is, $\mathsf{trace}(\mathcal{C}) \vdash \varrho$.

2.4. Axioms, lemmas, and restrictions

One of the main contributions of this paper is the introduction of axioms, lemmas, and restrictions in ProVerif.

- axioms are properties that are true but do not need to be proved. However, if ProVerif happens to find a contradiction w.r.t. an axiom, it will trigger an error;
- lemmas are properties that ProVerif proves and then reuses while proving other queries or lemmas;
- restrictions model assumptions: only traces that satisfy the restrictions will be considered. They can be used *e.g.* to specify that a resource can be accessed by at most one process, without a heavy encoding with private channels.

This idea of restrictions and lemmas is not novel in the field of automatic verification: it already appears in Tamarin [33].

Definition 2. *A* ProVerif lemma, axiom, or restriction *is a correspondence query $F_1 \wedge \ldots \wedge F_n \Rightarrow \psi$ such that ψ only contains events and test formulas (i.e. no injective event, nested query, attacker fact, and message fact).*

Given a set $\mathcal{Q}$ of queries, $\mathcal{A}x$ of axioms, $\mathcal{L}$ of lemmas, and $\mathcal{R}$ of restrictions, a configuration $\mathcal{C}$ satisfies $\mathcal{Q}$ w.r.t. $\mathcal{A}x$, $\mathcal{L}$, and $\mathcal{R}$ if for all $\varrho \in \mathcal{A}x \cup \mathcal{L} \cup \mathcal{Q}$:

$$\mathsf{trace}(\mathcal{C}) \cap \{T \mid T \vdash \rho, \forall \rho \in \mathcal{R}\} \vdash \varrho$$

The only difference between $\mathcal{A}x$, $\mathcal{L}$, and $\mathcal{Q}$ is that ProVerif will not prove queries in $\mathcal{A}x$ and may use lemmas in $\mathcal{L}$ and axioms in $\mathcal{A}x$ to prove queries in $\mathcal{Q}$.

Example 6. *ProVerif cannot prove trace convergence of the process P_{diff} as defined in Example 5. The reason is that in case the voter's process could be executed several times, then a variant of the attack remains: the attacker could simply copy another ballot produced by Alice that did not reach the ballot box. P_{diff} specifies a system where each voter votes at most once but unfortunately, the internal behavior of ProVerif considers the case where several instances of the voter process are executed. To help ProVerif, we can add an axiom that says that the voter process cannot use two different inputs. To do so, we modify the voter process by adding an event $Uniq(st, r)$ that records that the randomness r was received by the first input at time st.*

$V'(vote, sk)$ =
$\mathsf{new}\ stamp;\ \mathsf{in}(c, x_r);\ \mathsf{event}\, Uniq(stamp, x_r);\ \mathsf{new}\ r;$
$\mathsf{out}(c, sign(aenc(vote, (r, x_r), pk(sk_e), sk))$

Then the following axiom states that no two distinct randomness can be received for the same input.

$$\mathrm{event}(Uniq(st, r_1)) \wedge \mathrm{event}(Uniq(st, r_2)) \Rightarrow r_1 = r_2$$

While such a property cannot be proved by ProVerif, it is easy to show (by hand) that it holds in any process containing V' provided that the event $Uniq$ only occurs in V': intuitively, the input is executed at most once for each creation of a fresh stamp, so if the stamp st is the same, then the message r_1 or r_2 received by the input must be the same as well.

The axiom above is suitable for correspondence queries. For equivalence queries, we need to use correspondence queries on bitraces. Then we use the following axiom, which considers the left and right sides separately:

$$\begin{aligned}&\mathrm{event}'(Uniq(st, r_1), Uniq(st, r'_1)) \wedge \\ &\mathrm{event}'(Uniq(st, r_2), Uniq(st, r'_2)) \Rightarrow r_1 = r_2 \wedge r'_1 = r'_2\end{aligned} \tag{1}$$

3. New procedure in ProVerif

In this section, we revisit the main steps of the algorithm of ProVerif outlined in the introduction: generation of clauses (Section 3.1), saturation (Section 3.2), and verification of security properties (Section 3.3). The generation of clauses has been described in [5], [10], [13] and is left mostly unchanged here, apart from the addition of inequality constraints on natural numbers and some novel optimizations discussed in Section 4. The saturation is extended with new transformations on clauses in order to support lemmas

and natural numbers. The verification is also extended with ordering constraints expressing that certain facts happen before others; this is essential to support proofs by induction and temporal queries.

3.1. Clauses generation

ProVerif generates clauses of the form $\phi \wedge \bigwedge_{j=1}^{m} F_i \to C$ where $F_1, \ldots, F_n, C$ are facts and $\phi = \bigwedge_{i=1}^{n} M_i \; op_i \; N_i \wedge \bigwedge_{i'=1}^{n'} p_{i'}(M'_i) \wedge \bigwedge_{i''=1}^{n''} \forall \tilde{x}_i . M''_i \neq N''_i$ with $op_i \in \{\geq, =\}$, $p_{i'} \in \{\mathrm{isnat}, \neg\mathrm{isnat}\}$. We call ϕ a *constraint formula*.

Attacker clauses. Given an initial configuration $\mathcal{C}_0 = \mathcal{E}_0, P_0, \mathcal{A}_0$, ProVerif generates a set of attacker clauses $\mathbb{C}_{\mathcal{A}}(\mathcal{C}_0)$ containing in particular the following clauses.

$$\to \mathrm{att}(a[]) \qquad a \in \mathcal{A}_0 \quad \text{(Ri)}$$

$$\mathrm{att}(x_1) \wedge \ldots \wedge \mathrm{att}(x_n) \to \mathrm{att}(f(x_1, \ldots, x_n)) \qquad f \in \mathcal{F}_c \quad \text{(Rf)}$$

$$\mathrm{att}(U_{i,1}) \wedge \ldots \wedge \mathrm{att}(U_{i,n}) \wedge \phi_i \to \mathrm{att}(U_i) \quad \text{(Rg)}$$
$$\mathsf{def}(g) = [g(U_{i,1}, \ldots, U_{i,n}) \to U_i]_{i=1}^{k}$$
$$\phi_i = \textstyle\bigwedge_{j<i}(U_{i,1}, \ldots, U_{i,n}) \neq (U_{j,1}, \ldots, U_{j,n})$$

$$\mathrm{msg}(x, y) \wedge \mathrm{att}(x) \to \mathrm{att}(y) \quad \text{(Rl)}$$

$$\mathrm{att}(x) \wedge \mathrm{att}(y) \to \mathrm{msg}(x, y) \quad \text{(Rs)}$$

$$\mathrm{att}(x) \to \mathrm{att}(succ(x)) \quad \text{(R+)}$$

$$\to \mathrm{att}(zero) \quad \text{(R0)}$$

The attacker clauses reflect that the attacker initially knows some names (Ri), may apply any function ((Rf) and (Rg)), learn a message sent over the network when she knows the corresponding channel (Rl), and conversely may send a message of her knowledge on a channel she knows (Rs). The last two clauses let the attacker build any natural number.

Protocol clauses. Similarly, a set of protocol clauses, denoted $\mathbb{C}_{\mathcal{P}}(\mathcal{C}_0)$, is generated from the processes in $\mathcal{C}_0$. Intuitively, each output of a process triggers a new clause whose hypotheses reflect previous inputs and the conclusion reflects the output. Fresh values corresponding to new n do not exist in clauses hence they are emulated by a function symbol that depends on previously met variables, denoted $n[x_1, \ldots, x_k]$. These variables include a replication index for each replication above the new, representing intuitively the copy number of the replicated process, so that a different fresh name is generated in each copy. They generally include messages received at inputs as well, to express that the same fresh name cannot be generated after receiving different inputs. That often improves the precision of the analysis, but is optional. Additionally, ProVerif first renames bound names to distinct names before translating the process into clauses, guaranteeing that no new n occurs syntactically twice, thus avoiding that two generations of names be emulated by the same function symbol.

We provide more intuition on an example.

Example 7. *The process $V'(vote, sk)$ defined in Example 6 yields the generation of two clauses:*

$$\mathrm{att}(x) \to \text{m-event}(\mathit{Uniq}(\mathit{stamp}[], x)) \quad (2)$$

$$\mathrm{att}(x) \wedge \text{s-event}(\mathit{Uniq}(\mathit{stamp}[], x)) \to \mathrm{att}(u) \quad (3)$$

where $u = sign(aenc(vote, (r[x], x), pk(sk_e)), sk)$. The first clause corresponds to the fact that an event $Uniq(stamp[], x)$ may be triggered as soon as the process has inputted x from the attacker. The randomness new $stamp$ *is replaced by a constant $stamp[]$ as the declaration is not preceded by any input or replication. The second clause corresponds to the output of the voter. The random r is replaced by a function that depends on the inputs $r[x]$. The output can occur only if the event $Uniq(stamp[], x)$ has been triggered already. (We explain below why we use two predicates for events,* m-event *and* s-event.*)*

It has been shown [5, Theorem 1] that the generated clauses $\mathbb{C}_{\mathcal{A}}(\mathcal{C}_0) \cup \mathbb{C}_{\mathcal{P}}(\mathcal{C}_0)$ are sufficient to generate any fact that can be executed in a trace of the initial configuration. More formally, for any fact F that holds in some trace T (at some step), F is derivable from the generated clauses. We prove a refined version of this result, showing that clause derivations precisely mimic execution traces, obeying the same order. More precisely, we show that not only F is derivable but moreover, for any clause $F_1 \wedge \cdots \wedge F_n \to C$ used in the derivation, F_i is satisfied in T at some step τ_i, C is satisfied in T at some step τ, and we have that $\tau_i < \tau$ if F_i is an event, $\tau_i \leq \tau$ otherwise (and we also show the strict inequality under some conditions). Obtaining a strict inequality $\tau_i < \tau$ is a key property to prove lemmas by induction. We refer to this property as our *main invariant*.

In order to reason on correspondence queries, we distinguish between events that may be produced (m-event; "m" is for "may") and events we know for sure to be in the trace (s-event; "s" is for "sure"). Indeed, our main invariant implies that, if some event ev is executed in a trace T, then ev is derivable from the clauses, so ev is in the conclusion of some clause. However, the converse is not true. Hence, events in the conclusion of clauses *may* be executed. We use m-event for these events. In contrast, events that occur in the hypothesis of a clause are required for the conclusion to be derived. More precisely, when some fact F holds in a trace T, F is derivable from the clauses, so there is a clause that concludes F and whose hypotheses are satisfied; in particular, the events in the hypothesis of that clause are *for sure* in the trace T, so we use s-event for these events. This is the main argument for proving correspondence queries.

Extension to biprocesses. Clauses $\mathbb{C}'_{\mathcal{A}}(\mathcal{C}_0)$ and $\mathbb{C}'_{\mathcal{P}}(\mathcal{C}_0)$ are similarly generated for configurations with biprocesses. The previous clauses are adapted to bifacts. For instance, the following clauses adapt (Ri), (Rf), and (Rl) to bifacts:

$$\to \mathrm{att}'(a[], a[]) \qquad a \in \mathcal{A}_0 \quad \text{(Ri')}$$

$$\mathrm{att}'(x_1, x'_1) \wedge \ldots \wedge \mathrm{att}'(x_n, x'_n) \to \mathrm{att}'(f(x_1, \ldots, x_n), f(x'_1, \ldots, x'_n)) \quad \text{(Rf')}$$

$$\mathrm{msg}'(x, y, x', y') \wedge \mathrm{att}'(x, x') \to \mathrm{att}'(y, y') \quad \text{(Rl')}$$

Moreover, additional clauses conclude the special predicate bad when the attacker can observe a difference between the two sides. For instance, attacker clauses conclude bad when a destructor can successfully be applied on one side and not on the other. For the equality test, we obtain the following

clauses, which conclude bad when we have equality on one side and not on the other:

$$\begin{array}{l}\mathrm{att}'(x,y) \wedge \mathrm{att}'(x,y') \wedge y \neq y' \rightarrow \mathrm{bad}\\ \mathrm{att}'(x,y) \wedge \mathrm{att}'(x',y) \wedge x \neq x' \rightarrow \mathrm{bad}\end{array} \quad (4)$$

Similarly, protocol clauses conclude bad if some protocol step can happen on one side and not on the other.

Precise actions. The generated clauses do not perfectly model that actions cannot be repeated arbitrarily. Instead, a clause can be "used" arbitrarily during the resolution procedure. In particular, one cannot say that the two clauses of Example 7 should be used at most once while they do correspond to a process that can be executed only once.

Following the initial work of [14] on global states, it is now possible to tell ProVerif that an input should be taken into account as precisely as possible, by annotating the input with `[precise]`. This yields the generation of the axioms defined in Example 6. The fresh name *stamp* used in the axioms is represented internally as a function of the replication indices of the replications above the input (if any). Therefore, the axioms express that, when we are in the same copy of the process (same replication indices, that is, same stamp), the received message must be the same.

3.2. Clauses saturation

The core step in ProVerif is the saturation of clauses. The idea is to deduce clauses that are consequences of the initial ones, such that queries can then be evaluated directly on the resulting clauses.

3.2.1. Resolution rule and selection function. The core idea of the saturation procedure consists in combining two clauses to produce a new one. This is the resolution rule:

$$\frac{H \rightarrow C \qquad F \wedge H' \rightarrow C' \qquad \sigma = mgu(F, C)}{H\sigma \wedge H'\sigma \rightarrow C'\sigma}\ (\mathrm{Res})$$

where $mgu(F, F')$ is the most general unifier of the two facts F and F'.

To avoid too many resolutions that would immediately yield termination issues, we assume a *selection function*, that is a function sel from clauses to sets of facts. The resolution rule is applied only on selected facts, that is, when $\mathrm{sel}(H \rightarrow C) = \emptyset$ and $F \in \mathrm{sel}(F \wedge H' \rightarrow C')$.

After generating a new clause by resolution, ProVerif applies multiple simplification rules. For example, tautologies are suppressed:

$$\frac{\mathbb{C} \cup \{F \wedge H \rightarrow F\}}{\mathbb{C}}\ (\mathrm{Taut})$$

The complete set of simplification rules is denoted (Simpl) and has not been changed. Hence we refer the reader to [8], [10] for its complete description.

3.2.2. Natural numbers. Formulae now contain predicates on natural numbers, such as $M \geq N$, $\neg\mathrm{isnat}(M)$ and $\mathrm{isnat}(M)$. Natural numbers are handled by a special procedure. Following the work in [14], we rely on the algorithm of Pratt [30] that computes the set of solutions of a system of inequalities. Note that compared to [14], we can consider a fully untyped attacker, as done by default in ProVerif. For example, in a process $\mathsf{in}(c, x); P$ where x is e.g. used for comparison, the work of [14] assumes that x can only be instantiated by a natural number while we let here the attacker freely choose any value. Another key difference with [14] is that we process clauses with natural numbers from the saturation phase while [14] only changed the verification phase. This earlier treatment yields a better integration and more simplification rules, hence fewer generated clauses.

Proposition 2 ([14])**.** *There is a polynomial time algorithm* $\mathsf{checkeq}$ *that given a conjunction ϕ of inequalities between terms returns:*

- $\bot$ *if ϕ has no solution*
- *a substitution σ' such that for all solutions σ of ϕ, there exists a substitution δ such that $\sigma = \sigma'\delta$.*

Relying on the algorithm $\mathsf{checkeq}$, we can consider the following simplification rules specific to natural number predicates. A formula is simplified when it admits solutions.

$$\frac{\mathbb{C} \cup \{H \wedge \phi \rightarrow C\} \quad \phi = \bigwedge_i M_i \geq N_i \quad \mathsf{checkeq}(\phi) = \sigma}{\mathbb{C} \cup \{H\sigma \wedge \phi\sigma \rightarrow C\sigma\}}$$

The clause $H \wedge \phi \rightarrow C$ is removed when $\mathsf{checkeq}(\phi) = \bot$.

We also consider a few simple rules that detect when the predicate isnat is satisfied or not. All this set of rules for natural numbers is denoted (Nat). Finally, any clause that only allows an attacker to derive a natural number is useless:

$$\frac{\mathbb{C} \cup \{R = (H \wedge \phi \rightarrow \mathrm{att}(M))\} \qquad \phi \models \mathrm{isnat}(M)}{\mathbb{C}}(\mathrm{NatCl})$$

provided R is not one of the attacker clauses (R+) or (R0) themselves. A similar rule (NatCl') is defined for bifacts.

3.2.3. Reasoning with lemmas. The key idea of lemmas is that they can be used during the saturation procedure in order to simplify clauses by applying lemmas as soon as possible, inside clauses. Since protocol clauses do not use directly events but may- and sure- events instead, we reflect this transformation in lemmas as well. Formally, given a conjunction of atomic formulas ψ, we denote by $\lceil\psi\rceil^s$ the formula obtained from ψ by replacing all events $\mathrm{event}(ev)$ and $\mathrm{event}'(ev_1, ev_2)$ with respectively sure-events $\text{s-event}(ev)$ and $\text{s-event}'(ev_1, ev_2)$. Similarly, given a query conclusion ψ, we denote by $\lceil\psi\rceil^m$ the query conclusion obtained from ψ by replacing all events $\mathrm{event}(ev)$ with may-events $\text{m-event}(ev)$ and similarly for $\mathrm{event}'(ev_1, ev_2)$.

A lemma is a correspondence query $F_1 \wedge \ldots \wedge F_n \Rightarrow \bigvee_{j=1}^{m} \psi_j$. We apply it to a clause $H \rightarrow C$ by replacing H with $H \wedge \lceil\psi_j\sigma\rceil^s$ as soon as the hypotheses of the lemma are satisfied, that is, $\lceil F_i\sigma\rceil^s \in H$. The resulting clause has more assumptions, hence is more precise. It will allow to

derive fewer clauses, enhancing termination. The full rule reflecting the application of lemmas is the following one.

$$\frac{\mathbb{C} \cup \{H \to C\} \qquad (\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j) \in \mathcal{L} \quad \text{for all } i, \text{ either } \lceil F_i\sigma \rceil^s \in H \text{ or } (\lceil F_i\sigma \rceil^m = C \text{ and } \forall j, \forall F \in \psi_j,\ mgu(\lceil F\sigma \rceil^m, C) = \bot)}{\mathbb{C} \cup \{H \wedge \lceil \psi_j\sigma \rceil^s \to C\}_{j=1}^{m}} \quad (\text{Lem}(\mathcal{L}))$$

The last line in the hypotheses corresponds to a more subtle application of the lemma: it can be applied also when the assumptions of the lemma are satisfied by the conclusion of the clause. This still preserves our *main invariant* for the following reason. We consider a trace T and a fact F that holds in T and a derivation of F with the clauses obtained so far. When $H \to C$ is applied in the derivation, then C holds in T at some step τ and the facts of H hold at some steps earlier than τ (or at τ). Since the lemma $(\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j)$ is true and since the $F_i\sigma$ hold in T (at a step smaller or equal to τ), we know that every ψ_j holds too. Hence given a fact $F \in \psi_j$, $F\sigma$ holds at τ' with $\tau' \leq \tau$. Now, since we know that $F\sigma$ cannot correspond to the same event as C (since $mgu(\lceil F\sigma \rceil^m, C) = \bot$), it must be the case that $\tau' < \tau$, hence our main invariant is satisfied.

Example 8. *During the saturation procedure for proving equivalence on the process P_{diff} as defined in Example 6, ProVerif generates the following clause:*

$$\begin{aligned} &\text{s-event}'(\mathit{Uniq}(\mathit{stamp}_1[], x_1), \mathit{Uniq}(\mathit{stamp}_1[], x_1')) \\ &\quad \wedge \text{s-event}'(\mathit{Uniq}(\mathit{stamp}_1[], x_2), \mathit{Uniq}(\mathit{stamp}_1[], x_2')) \\ &\quad \wedge \text{s-event}'(\mathit{Uniq}(\mathit{stamp}[], x_3), \mathit{Uniq}(\mathit{stamp}[], x_3')) \\ &\quad \wedge \text{att}'(x_1, x_1') \wedge \text{att}'(x_2, x_2') \wedge \text{att}'(x_3, x_3') \\ &\quad \wedge (x_1, x_1') \neq (x_2, x_2') \\ &\quad \to \text{att}'(v_2[], v_1[]) \end{aligned}$$

This clause corresponds to a scenario where Alice would receive a random coin ($\text{att}'(x_3, x_3')$) to encrypt her vote while Bob would encrypt twice his vote with different received random coins ($\text{att}'(x_1, x_1')$ and $\text{att}'(x_2, x_2')$). The attacker then signs Bob's second ballot with her own key and sends the three ballots to the board. This clause does not correspond to a real trace since Bob cannot vote twice.

Moreover, this clause is problematic for proving equivalence as it concludes the fact $\text{att}'(v_2[], v_1[])$ where v_1 and v_2 are names in the initial knowledge of the attacker, i.e. $v_1, v_2 \in \mathcal{A}_0$. Thus, the clause $\to \text{att}'(v_1[], v_1[])$ is included in the attacker clauses. With Clause (4), we obtain a derivation of bad*, hence ProVerif cannot prove equivalence.*

However, the hypotheses of the clause contain the events $\text{s-event}'(\mathit{Uniq}(\mathit{stamp}_1[], x_1), \mathit{Uniq}(\mathit{stamp}_1[], x_1'))$ *and* $\text{s-event}'(\mathit{Uniq}(\mathit{stamp}_1[], x_2), \mathit{Uniq}(\mathit{stamp}_1[], x_2'))$ *that match the premise of lemma (1) with $\sigma = \{^{\mathit{stamp}_1[]}/_{st}\} \cup \{^{x_i}/_{r_i}, ^{x_i'}/_{r_i'}\}_{i=1,2}$. Hence, the application of the rule (Lem($\mathcal{L}$)) adds the formula $x_1 = x_2 \wedge x_1' = x_2'$ to the hypothesis of the clause. Since this formula contradicts the formula $(x_1, x_1') \neq (x_2, x_2')$ already in the hypothesis of the clause, the clause is discarded by ProVerif.*

Inductive lemmas. The rule Lem($\mathcal{L}$) can only be applied when lemmas in $\mathcal{L}$ are already proved. An inductive lemma is not yet proved but we still wish to use it during the saturation procedure. In particular, an inductive lemma $(\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j)$ can be used to prove itself provided it is applied on smaller instances. More formally, during the saturation procedure, we know that it holds until steps $\tau' < \tau$. Hence we can only apply it provided the facts $F_i\sigma$ hold at a step $\tau'' < \tau$. So we simply allow the application of an inductive lemma to hypotheses of a clause, but not its conclusion.

$$\frac{\mathbb{C} \cup \{H \to C\} \quad (\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j) \in \mathcal{L} \quad \text{for all } i,\ \lceil F_i\sigma \rceil^s \in H}{\mathbb{C} \cup \{H \wedge \lceil \psi_j\sigma \rceil^s \to C\}_{j=1}^{m}} \quad (\text{Ind}(\mathcal{L}))$$

3.2.4. Subsumption. A clause C should be removed when there exists another clause C' that is more general than C. This is called subsumption. Formally, we say that $C = H_1 \wedge \phi_1 \to C_1$ subsumes $C' = H_2 \wedge \phi_2 \to C_2$, denoted $C \sqsupseteq C'$, if there exists σ such that

(i) either $C_1\sigma = C_2$ or $C_1 = \text{bad}$
(ii) $H_1\sigma \subseteq H_2$ (where H_1 and H_2 are seen as multisets of facts and $\subseteq$ is the multiset inclusion)
(iii) $\phi_2 \models \phi_1\sigma$.

Removing subsumed clauses is crucial in order to avoid the generation of too many clauses and hence termination issues.

3.2.5. Saturation procedure. The saturation procedure applies the resolution rule, followed by simplification rules as much as possible. We consider a set $\mathcal{L}$ of axioms, restrictions, and already proved lemmas, and a set $\mathcal{L}_i$ of lemmas to be proved by induction. The simplification of a set of clauses $\mathbb{C}$ is defined as follows:

- $\mathsf{simplify}_{\mathcal{L},\mathcal{L}_i}(\mathbb{C})$ repeatedly applies on $\mathbb{C}$ the rules Simpl, Nat, NatCl, NatCl', Lem($\mathcal{L}$), Ind($\mathcal{L}_i$) until a fixpoint is reached.
- then $\mathsf{condense}(\mathbb{C})$ eliminates from $\mathbb{C}$ the clauses that are subsumed by other clauses from $\mathbb{C}$. It also removes clauses that are redundant in the sense that their conclusion can already be derived from other clauses. This redundancy rule is applied with care to preserve soundness.

Finally, the saturation algorithm $\mathsf{saturate}_{\mathcal{L},\mathcal{L}_i}(\mathbb{C})$ combines simplification and resolution:

(i) $\mathbb{C}_{\mathsf{current}} := \mathsf{condense}(\mathsf{simplify}_{\mathcal{L},\mathcal{L}_i}(\mathbb{C}))$,
(ii) generate a new clause R by applying the resolution rule (Res) to some clauses of $\mathbb{C}_{\mathsf{current}}$,
(iii) $\mathbb{C}_{\mathsf{current}} := \mathsf{condense}(\mathsf{simplify}_{\mathcal{L},\mathcal{L}_i}(\{R\}) \cup \mathbb{C}_{\mathsf{current}})$,
(iv) repeat the steps (ii) and (iii) until a fixpoint is reached,
(v) return $\{R \in \mathbb{C}_{\mathsf{current}} \mid \text{sel}(R) = \emptyset\}$.

3.3. Verification

Once the set of clauses corresponding to a protocol is saturated, it is time to check whether the properties are satisfied. We explain the procedure for equivalence properties and correspondence queries.

3.3.1. Equivalence queries. The simplest case is to check equivalence queries: if all clauses concluding bad have been removed, then equivalence is satisfied. The algorithm for verifying equivalence queries is given in Algorithm 1.

Algorithm 1: $\texttt{prove'}\ (\mathcal{C}, \mathcal{A}x, \mathcal{L}, \mathcal{R})$: Verification procedure for equivalence queries

input : An initial biconfiguration $\mathcal{C}$, sets of axioms $\mathcal{A}x$, proved lemmas $\mathcal{L}$, and restrictions $\mathcal{R}$ on bitraces

$\mathbb{C} := \mathbb{C}'_{\mathcal{P}}(\mathcal{C}) \cup \mathbb{C}'_{\mathcal{A}}(\mathcal{C})$

$\mathbb{C}_{sat} := \mathsf{saturate}_{\mathcal{A}x \cup \mathcal{L} \cup \mathcal{R}, \emptyset}(\mathbb{C})$

return $\neg\exists\ (H \to \text{bad}) \in \mathbb{C}_{sat}$

The following theorem, proved in [11, Theorem 5], shows that this algorithm is sound.

Theorem 1. *Let $\mathcal{C}$ be an initial biconfiguration, $\mathcal{A}x$, $\mathcal{L}$, $\mathcal{R}$ be respectively sets of axioms, lemmas, and restrictions. If*

- *for all $\varrho \in \mathcal{A}x \cup \mathcal{L}$, $\mathsf{trace}(\mathcal{C}) \cap \{T \mid T \vdash \rho, \forall \rho \in \mathcal{R}\} \vdash \varrho$*
- *and $\texttt{prove'}\ (\mathcal{C}_I, \mathcal{A}x, \mathcal{L}, \mathcal{R})$ returns true*

then $\mathsf{trace}(\mathcal{C}) \cap \{T \mid T \vdash \rho, \forall \rho \in \mathcal{R}\}{\downarrow\uparrow}$.

Without restrictions, Theorem 1 shows that $\mathsf{fst}(\mathcal{C})$ and $\mathsf{snd}(\mathcal{C})$ are observationally equivalent by Proposition 1. However, with restrictions, it does not directly show observational equivalence (for traces satisfying the restrictions), unless the restrictions hold on one side of the bitrace if and only if they hold on the other. In the future, we plan to study sufficient conditions for observational equivalence with restrictions.

3.3.2. Correspondence queries. The procedure to check correspondence queries is more complex. We describe here only the main principles, summarized in Algorithm 2. Due to space constraints, several notations and sub-algorithms are explained informally in text here.

The set of clauses corresponding to the initial configuration is first saturated, yielding a set $\mathbb{C}_{sat}$, assuming axioms and lemmas to hold (lemmas should be proved in a previous round). Finally, the key step is to check whether a query ϱ holds. For the sake of clarity, assume that ϱ is a simple query of the form $F_1 \wedge F_2 \Rightarrow \psi$ where $F_1 = \text{event}(A_1(M_1, \ldots, M_{k_1}))$ and $F_2 = \text{event}(A_2(N_1, \ldots, N_{k_2}))$. The goal is to check that whenever F_1 and F_2 hold then ψ holds as well. Hence we would like to compute all possible clauses that can derive an instance of F_1 and F_2 and check whether ψ is indeed satisfied. To do so, an ingenious idea consists in saturating the clause $\lceil F_1 \rceil^m \wedge \lceil F_2 \rceil^m \to F_1 \wedge F_2$ with the clauses of $\mathbb{C}_{sat}$ in order to find all possible ways of producing an instance of $F_1 \wedge F_2$. However, a conjunction $F_1 \wedge F_2$ cannot appear as a conclusion of a clause. Therefore, we introduce the fact and_{F_1,F_2} defined as $conj_{A_1,A_2}(M_1, \ldots, M_{k_1}, N_1, \ldots, N_{k_2})$. In ProVerif 2.00, it is sufficient to saturate the clause

$$\text{m-event}(A_1(M_1, \ldots, M_{k_1})) \wedge \text{m-event}(A_2(N_1, \ldots, N_{k_2})) \to conj_{A_1,A_2}(M_1, \ldots, M_{k_1}, N_1, \ldots, N_{k_2}) \quad (*)$$

with $\mathbb{C}_{sat}$. Due to inductive lemmas, this is no longer possible. Indeed, if the query ϱ is an inductive lemma, it has already been used to generate the clauses in $\mathbb{C}_{sat}$ and we now need to prove ϱ. Hence intuitively, while saturating (*) we should only apply clauses at "earlier steps", which cannot be controlled. More formally, the clause (*) immediately breaks our main invariant. Indeed it is not true that $\text{m-event}(A_1(M_1, \ldots, M_{k_1}))$ holds at a (strictly) earlier step than $and_{A_1,A_2}(M_1, \ldots, M_{k_1}, N_1, \ldots, N_{k_2})$. Therefore, we had to completely rework the saturation procedure $\mathsf{saturateS}$ used in the verification phase. We now annotate facts with temporal variables ($F@t$) and we collect explicit ordering constraints on the temporal variables t. Continuing our example, we consider the clause R_q defined as:

$$\text{m-event}(A_1(M_1, \ldots, M_{k_1}))@t_1 \wedge \text{m-event}(A_2(N_1, \ldots, N_{k_2}))@t_2 \to conj_{A_1,A_2}(M_1, \ldots, M_{k_1}, N_1, \ldots, N_{k_2})@(t_1, t_2)$$

We then generalize the saturation procedure to clauses with temporal annotations in order to keep track when an event is generated. For example, assume we have a clause $F_1 \wedge F_2 \to \text{m-event}(A_1(M_1, \ldots, M_{k_1}))$ in $\mathbb{C}_{sat}$. Then the resolution will produce the clause

$$F_1@t'_1 \wedge F_2@t''_1 \wedge \text{m-event}(A_2(N_1, \ldots, N_{k_2}))@t_2 \wedge t'_1 < t_1 \wedge t''_1 < t_1 \to conj_{A_1,A_2}(M_1, \ldots, M_{k_1}, N_1, \ldots, N_{k_2})@(t_1, t_2)$$

In such a case, while saturating R_q, we can use the inductive lemma ϱ on F_1 and $\text{m-event}(A_2(N_1, \ldots, N_{k_2}))$ for instance since $\{t'_1, t_2\} < \{t_1, t_2\}$ as multiset ordering. More generally, we define order conditions to state when saturation rules can be applied.

Finally, we check whether the resulting set of clauses entails the query ϱ under verification. Intuitively, we check that all possible instantiations of a clause $R = H \to C$ satisfy ϱ. This is done by finding a substitution σ that first matches the facts in the premises of ϱ with C and second guarantees that the atomic formulas in the conclusion of $\varrho\sigma$ are entailed by H. For example, for an event F, we check that $F\sigma$ occurs directly in H. For a test formula $M \neq N$, we check that the disequalities in H implies $M\sigma \neq N\sigma$. We write $R \models \varrho$ when we can find such substitution σ for R and ϱ.

Example 9. *Let $\varrho = \text{event}(A(x))@t_1 \wedge \text{event}(B(y))@t_2 \Rightarrow \text{event}(C(x, y))@t_3 \wedge x \neq y \wedge t_1 > t_3$ and consider the clause $R = \text{s-event}(C(a[], y_1))@t'_3 \wedge y_1 \neq a[] \wedge t'_3 < t'_1 \wedge \text{att}(y_1) \to conj_{A,B}(a[], y_1)@(t'_1, t'_2)$. Recall that the fact $conj_{A,B}(a[], y_1)@(t'_1, t'_2)$ corresponds to the conjunction of $\text{event}(A(a[]))@t'_1$ and $\text{event}(B(y_1))@t'_2$. By taking $\sigma = \{^{a[]}/_x; ^{y_1}/_y; ^{t'_1}/_{t_1}; ^{t'_2}/_{t_2}; ^{t'_3}/_{t_3}\}$, we have that $\text{event}(A(x\sigma))@t_1\sigma$ and $\text{event}(B(y\sigma))@t_2\sigma$ match the conclusion of R. Moreover, $\text{event}(C(x, y)\sigma)@t_3\sigma$ occurs in the hypotheses, $x\sigma \neq y\sigma$ is entailed by $y_1 \neq a[]$, and $t_1\sigma > t_3\sigma$ is entailed by $t'_3 < t'_1$. So $R \models \varrho$. If R contained $t'_3 < t'_2$ instead of $t'_3 < t'_1$, then $t_1\sigma > t_3\sigma$ would not be entailed and we would have $R \not\models \varrho$.*

The case of injective events is detailed in [11]. It follows similar ideas but requires a more subtle treatment.

Algorithm 2: `prove` $(\mathcal{C}, \mathcal{A}x, \mathcal{L}, \mathcal{L}_i, \mathcal{R}, \mathcal{Q})$: Verification procedure for correspondence queries

input : An initial configuration $\mathcal{C}$, sets of axioms $\mathcal{A}x$, proved lemmas $\mathcal{L}$, inductive lemmas $\mathcal{L}_i$, restrictions $\mathcal{R}$ and queries $\mathcal{Q}$.

$\mathbb{C} := \mathbb{C}_{\mathcal{P}}(\mathcal{C}) \cup \mathbb{C}_{\mathcal{A}}(\mathcal{C})$
$\mathbb{C}_{sat} := \mathsf{saturate}_{\mathcal{L}\cup\mathcal{A}x\cup\mathcal{R},\mathcal{L}_i}(\mathbb{C})$
return $\forall \varrho \in \mathcal{Q} \cup \mathcal{L}_i$
  suppose $\varrho = F_1 \wedge \cdots \wedge F_n \Rightarrow \psi$
  $G_1 := \lceil F_1 \rceil^m, \ldots, G_n = \lceil F_n \rceil^m$
  $R_q := G_1@t_1 \wedge \ldots \wedge G_n@t_n \rightarrow and_{G_1,\ldots,G_n}@(t_1, \ldots, t_n)$
  if $\varrho \in \mathcal{L}_i$ **then**
    $\mathbb{C}_s := \mathsf{saturateS}_{\mathcal{L}\cup\mathcal{A}x\cup\mathcal{R},\mathcal{L}_i}(\{R_q\}, \mathbb{C}_{sat})$
  else
    $\mathbb{C}_s := \mathsf{saturateS}_{\mathcal{L}\cup\mathcal{L}_i\cup\mathcal{A}x\cup\mathcal{R},\emptyset}(\{R_q\}, \mathbb{C}_{sat})$
  $\forall R \in \mathbb{C}_s.\ R \models \varrho$

We can prove soundness of the ProVerif procedure.

Theorem 2. *Let $\mathcal{C}$ be an initial configuration. Let $\mathcal{Q}$, $\mathcal{A}x$, $\mathcal{L}$, $\mathcal{L}_i$, $\mathcal{R}$ be respectively a set of correspondence queries, axioms, lemmas, inductive lemmas, and restrictions. If*

- *for all $\varrho \in \mathcal{A}x \cup \mathcal{L}$, $\mathsf{trace}(\mathcal{C}) \cap \{T \mid T \vdash \rho, \forall \rho \in \mathcal{R}\} \vdash \varrho$*
- *and* `prove` $(\mathcal{C}, \mathcal{A}x, \mathcal{L}, \mathcal{L}_i, \mathcal{R}, \mathcal{Q})$ *returns true*

then $\mathcal{C}$ satisfies $\mathcal{Q} \cup \mathcal{L}_i$ w.r.t. $\mathcal{A}x$, $\mathcal{L}$, and $\mathcal{R}$.

A generalization of this result is proved in [11, Theorem 6] and a brief proof sketch appears in Appendix C. Using Theorem 2, ProVerif proves lemmas in round, using the previous lemmas to possibly help proving the next ones, and finally proves the requested queries.

3.4. Experiments

Starting from the source code of ProVerif 2.00, we have implemented the new saturation and verification procedure that has been included in the mainstream version of ProVerif. Our benchmarks and source code are available in [11]. In order to evaluate its ability to prove new protocols, we have considered ProVerif models provided in the distribution or in various papers, for which ProVerif 2.00 was unable to conclude (answered "cannot be proved"). This is the case in particular of several protocols coming from models in CryptoVerif and also of the Helios protocol, identified as problematic for ProVerif 2.00. Among the ProVerif files provided in its distribution, 58 queries resulted in a "cannot be proved". Our new version can now conclude (with a proof or an attack) for 21 of them. The remaining inconclusive queries are mostly either queries where trace reconstruction has been deactivated (13 in total) or equivalence queries (15 in total): when there is an attack, the best ProVerif can do is find a trace that does not converge, but this trace does not prove that the observational equivalence is false.

Officially published ProVerif files do not contain a lot of failures however. Indeed, authors publish files for which they succeeded to prove the desired properties and typically do not show all the intermediate files for which they had to fine tune the model. So to further test our new version of ProVerif, we used files sent by authors to ProVerif's development team when they needed help with ProVerif. Some other files have been obtained through teaching classes, where students had to model their newly invented protocols and faced a "cannot be proved". This shows that our new version will also help users that are beginners in protocol modeling.

The result of our experiments is reported in Table 1. In some cases, we had to use some options that we have introduced (columns P, I, R). Column "P" indicates that the option `set preciseActions = true` has been used. This corresponds to adding the option precise to all in actions. Column "I" corresponds to a technique that can be (optionally) used in the verification phase. On each generated clause, ProVerif 2.00 tests if the resolution of some facts in the hypotheses of the clause could trivially lead to non-termination. In such a case, ProVerif usually prevents such a resolution. Though it helps for termination, it can be detrimental for proving the query. Our new technique consists in allowing these facts to be resolved but at most once (or any - small - number indicated in option).

Column "R" corresponds to another new option. Lemmas and axioms usually require to add events in the original process (*e.g.* when setting the `precise` option to true). These events complicate the generated clauses, which could lead to non-termination. The option `set removeEventsForLemma = true` strips these events from clauses during the saturation procedure when it determines that they would most probably not be useful anymore (*e.g.* if they have already been used to apply some lemma). The simpler clauses are more likely to provide termination, at the cost of a loss of precision.

For some files, the new result is due to a change not documented here. In particular, we have considerably improved the proof of injective queries as exemplified by the file Student2. This part is fully detailed in [11]. Moreover, we have reworked attack reconstruction in particular for nested or injective queries, yielding a better treatment of attacks.

As indicated in column A, we have added axioms in some examples. The axioms are based on the GSVerif [14] tool that, given a protocol model, automatically generates formulas ϕ proved to hold in [14]. There, the authors request queries of the form $\varrho \vee \neg\phi$ instead of a query ϱ. Here, we add the formulas ϕ as axioms. The option `precise` is a special case where axioms are automatically generated. Interestingly, our more integrated treatment of precise actions allows to prove many more protocols. Actually, none of the protocols presented in Table 1 could already be handled by the GSVerif [14] tool. To further compare our new version

1. The official distribution contains a weaker injective query on which ProVerif 2.00 can conclude but cannot conclude for the full property.

Protocol	Q	O	#	N	P	I	R	ℕ	A
published files									
PCV Otway-Rees	eq	✗	1	✓	x				
PCV Needham-Schroeder	inj	✗	6 3	✓ ↯	x	x			
PCV Denning-Sacco	inj	✗	1	↯					
JFK	cor inj	✗[1]	2 2	↯ ✓		x			
Arinc823	cor	✗	6	↯				x	
Helios-norevote	eq	✗	4	✓	x				
Signal	cor	✗	2	↯					
TLS12-TLS13-draft18	cor	✗	1	↯					
unpublished files									
QBC_4qubits	cor	✗	1 1	✓ ↯	x				
voting-draft	eq	✗	1	✓	x				
LAK-simplified	cor	⏱	1	✓		x			
PACE_v3_sequence	cor	✗	1 3	✓ ↯	x			x	x
DP-3T-simpl-draft	cor	⏱	1 2	✓ ↯	x	x	x	x	x
Student1	cor inj	⏱	2 1	✓ ↯	x		x		
Student2	inj	✗	1	✓					
Student3	cor	⏱	1	↯	x		x		
Student4	cor	✗	2	↯	x		x		
Student5	cor	⏱	1	↯					

TABLE 1. NEW PROOFS OR ATTACKS FOUND

Q: type of query *cor*: correspondence query *eq*: equivalence query *inj*: correspondence query with injective events
O: old result ✗: "cannot be proved" ⏱: out of time (>24h)
#: number of queries N: new result ✓: proof ↯: attack
ℕ: use of natural numbers A: use of axioms, lemmas
P, I, R: use of newly introduced options, see text.

of ProVerif with GSVerif, we considered the files proposed in [14]. We showed that all proofs were preserved when using the generated formulas as axioms. Moreover, one of the protocols (Yubikey [34]) was partly handled by hand in [14] and can now be covered using a proof by induction.

4. Efficiency

The second main contribution of the paper is a major improvement of the efficiency of ProVerif. Part of the gain in execution time is due to engineering optimizations that are hard to describe, like better sharing of data that spare both memory and time. But we also perform several algorithmic changes, that yield major gains in terms of efficiency.

4.1. New algorithms

(a) Subsumption. As explained in Section 3.2.4, during the saturation procedures, subsumed clauses are removed, in order to avoid immediate non termination issues. In particular, ProVerif checks for subsumption between each newly generated clause and all previously generated clauses. For complex protocols, we identified that the subsumption test could take up more than 80% of the total execution time. This does not come as a complete surprise since checking for subsumption between two clauses is NP-complete [24]. A few years ago, Schulz [32] introduced a novel technique based on feature vector indexing allowing efficient set-to-clause subsumption. A feature is a function f from clauses to natural numbers such that whenever $C_1 \sqsupseteq C_2$ then $f(C_1) \geq f(C_2)$ (*). In particular, if $f(C_2) > f(C_1)$ then we know that C_1 cannot subsume C_2 and we do not need to run the subsumption test. The technique allows to considerably reduce the number of subsumption tests that need to be performed.

(b) Clause generation. During the generation of clauses, for each protocol step (*e.g.* a condition), ProVerif evaluates the terms it contains. In case some destructors are defined by several rewrite rules, ProVerif has to consider all the possible cases, and repeatedly for the next steps of the protocol. This may grow quickly. In order to avoid this explosion as much as possible, we evaluate an argument of a function only when it is still needed in order to determine the result, knowing the value of the previous arguments.

(c) Set-to-clause resolution. During the saturation procedure, when a new clause $H \to C$ is generated, ProVerif attempts to apply the resolution rule (Res) between this clause and any other existing clauses. These resolutions were previously computed independently, leading to the computation of a large number of most general unifiers. However, in the application of the rule (Res) between $H \to C$ and other clauses, the same selected fact of $H \to C$ is always used to compute the different most general unifiers (e.g. C when $\mathrm{sel}(H \to C) = \emptyset$). Furthermore, when performing resolution between $H \to C$ and a set of clauses, we only need to consider clauses whose selected fact has the same function symbols (looking from the root until we meet variables) as the selected fact of $H \to C$, so that unification can succeed. Inspired by *substitution tree indexing* techniques [22], [23], we have implemented a set-to-clause resolution algorithm that stores the existing clauses in a tree indexed by the function symbols of the selected fact, starting from the root. Hence, clauses with a selected fact that has certain symbols at the root can be quickly retrieved.

(d) Global redundancy. During the verification phase, clauses are removed when they are "globally redundant". Intuitively, a clause is globally redundant if its conclusion can already be obtained by resolving existing clauses (where only the conclusion is selected). While this is useful for efficiency, this check can itself cause efficiency issues. Hence we have characterized cases where it is useless to check for global redundancy and cases where the procedure can be simplified (*e.g.* subsumption tests are useless here).

(e) Pre-treatment of processes. ProVerif 2.00 sometimes optimizes sequences of let such as 'let $x_1 = M_1$ in ... let $x_n = M_n$ in P' into a process that first evaluates all $M_1, \ldots, M_n$ and then executes P if none of them failed. This is particularly useful when proving equivalence queries as the third condition of Definition 1 (bitrace convergence) is satisfied more often. Typically, a trace can converge even if the evaluation of one term M_i fails on the left side and the evaluation of another $M_{i'}$ fails

Protocol	# files	2.00	(a)	(b)	(c)	(d)	(e)	gain ($\frac{t_{2.00}}{t_{(e)}}$)
Distrib	134	4min 39s	2min 05s	1min 55s	1min 51s	1min 52s	1min 14s	×3.8
Noise	42	≥ 170h 32min	≥ 124h 36min	29h 45min	13h 47min	3h 37min	20min	×516
TLS	3	8h 39min	47min	47min	44min	39min	32min	×16
Arinc823	18	11h 35min	24min	23min	20min	18min	17min	×42
Signal	13	30h 52min	23h 45min	16h 11min	3h 35min	1h 04min	58min	×32
Neuchâtel	9	≥ 73h 33min	≥ 24h 24min	6min	6min	5min	5min	×945

TABLE 2. TIME GAIN

Protocol	# files	2.00	(e)	gain ($\frac{m_{2.00}}{m_{(e)}}$)
Distrib	134	5.0GB	4.3GB	×1.2
Noise	40	58.3GB	7.4GB	×7.9
TLS	3	11.2GB	9.1GB	×1.2
Arinc823	18	8.2GB	8.4GB	×1
Signal	13	29.9GB	23.9GB	×1.2
Neuchâtel	6	0.3GB	0.2GB	×1.7

TABLE 3. MEMORY GAIN

on the right side. However, during the clause generation, evaluating the M_i when the evaluation of M_j failed leads to useless case distinctions and so to a loss of efficiency. We have improved the encoding by ensuring the terms M_i are not evaluated when the evaluation of a previous M_j fails while preserving the gain in precision for equivalence queries.

4.2. Time and memory gain

To evaluate the efficiency of our new ProVerif version, we considered all the example files provided in the distribution of ProVerif. Since our gain is important for files that already take time, we have separated the protocol models for the avionic protocol Arinc823 [9] from the other models in the distribution aggregated in a repository Distrib. For all the 134 files in Distrib, ProVerif 2.00 needs a total of 4min39s while the new ProVerif takes 1min14s. This means that the user experience remains the same for all these easy files: the new ProVerif answers almost immediately. We also considered major case studies of the literature, for which the execution time of ProVerif 2.00 was of several hours. Namely, we considered the following protocol models: protocols from the Noise Protocol Framework [26], [29], TLS [4], Signal [25], and the Neuchâtel voting protocol [17].

Our experiments have been run on an Intel Xeon 3.10GHz, with 340Gb of memory. Table 2 shows the gain in terms of execution time. A cell in the table corresponds to the cumulative execution time of all files for the considered protocol. The operator ≥ indicates that at least one of the files reached a timeout of 24h. To highlight the effect of each technique, we display the gain obtained for each algorithm.

- (a) includes the new subsumption test, based on feature vector indexing;
- (b) further adds a more efficient clause generation;
- (c) additionally contains the set-to-clause resolution;
- (d) also includes the modified algorithm for checking global redundancy;
- (e) finally contains all new algorithms.

We have also worked to reduce ProVerif's memory consumption by physically sharing more data and applying more greedily the simplifications and subsumption checks during the generation of initial clauses. Table 3 shows the gain in memory consumption. Once again, a cell in the table corresponds to the cumulative memory consumption of all files for the considered protocol. For fair representation, we only considered files that did not reach the 24h timeout since ProVerif's memory consumption usually increases with time. Note that, for the Neuchâtel protocol, we thus discarded in the table the three most time and memory consuming variants. We still highlight the following result: As mentioned in the introduction, the verification of ballot privacy for 6 voting options times out for ProVerif 2.00. ProVerif (b) is the first version that can verify it under 24h taking 5h 51min but consuming 104GB of memory, while ProVerif (e) completes the verification in 4h 35min and only consumes 0.4GB.

5. Related Work

Many verification tools exist for analysing the security of protocols, such as Avispa [3], DeepSec [15], Akiss [12], or Maude-NPA [21]. However, for the analysis of large scale modern protocols, the two main competitors are Tamarin [31] and ProVerif [6], two complementary tools. Both tools cover a large class of protocols, cryptographic primitives, and security properties. The main advantages of Tamarin are that it covers some primitives with associative and commutative properties (such as the exclusive or - XOR) and it provides a high level of interactions: a user may help the tool with lemmas but may also perform proofs "manually" in the interactive mode. On the other hand, Tamarin has a lower level of automation. For example, we are not aware of major protocols like Signal or TLS that have been treated without adding at least several well-adapted lemmas. Even when Tamarin can work automatically, it is significantly slower. For example, our new version of ProVerif can analyse the 134 example files of the distribution in 1min14s while Tamarin, in a recent improvement of its automation [16], takes 7min07s to analyse 22 files that encode simple protocols of the literature (also encoded in the distribution of ProVerif).

The development of ProVerif started more than 20 years ago, and several results have gradually improved the tool. Recently, a new approach [28] has been proposed to cover more complex equational theories (such as the one of XOR) in ProVerif, using an external solver to saturate the clauses modulo the theory. This approach applies to the secrecy property only. Natural numbers were first introduced in ProVerif in [14] but with a strong typing policy w.r.t. integers. We generalized this approach to allow a fully untyped attacker: an attacker may send an arbitrary term even when

an integer is expected. This may possibly yield attacks, for example if the integer (maybe a counter value) is transmitted to another agent without further checks, yielding possibly a confusion with another message. Several attempts have been performed to handle global states in ProVerif. The first one is StatVerif [2] that automatically translates a protocol with states into a set of Horn clauses that can be passed to ProVerif. In StatVerif, the number of states needs to remain finite (and small). More recently, GSVerif [14] introduced a different approach and automatically generates properties that are guaranteed to hold, helping ProVerif to avoid false attacks. The comparison performed in [14] shows that this approach can cover many more protocols than StatVerif and also performs better than SAPIC [27] that uses Tamarin as a back-end to prove protocols with global states. Our work enables to use GSVerif in a more native way by adding the generated properties as axioms. As a result, we now even cover the few examples (Yubikey, CANauth) that were left out by GSVerif.

Acknowledgements: This work has been partly supported by the ANR Research and teaching chair in AI ASAP and by ANR TECAP (decision number ANR-17-CE39-0004-03). This work was partly done while Vincent Cheval was at *Inria Nancy - Grand Est*.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, January 2001.

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

[3] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In *17th International Conference on Computer Aided Verification, CAV'2005*, volume 3576 of *LNCS*, pages 281–285. Springer, 2005.

[4] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P'17)*, pages 483–503, May 2017.

[5] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[6] B. Blanchet. Automatic verification of security protocols in the symbolic model: The verifier proverif. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *LNCS*, pages 54–87. Springer, 2014. https//proverif.inria.fr.

[7] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, Oct. 2016.

[8] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, 2016.

[9] B. Blanchet. Symbolic and computational mechanized verification of the ARINC823 avionic protocols. In *30th IEEE Computer Security Foundations Symposium (CSF'17)*, pages 68–82. IEEE, Aug. 2017.

[10] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *J. Log. Algebr. Program.*, 75(1):3–51, 2008.

[11] B. Blanchet, V. Cheval, and V. Cortier. Proverif with lemmas, induction, fast subsumption, and much more - technical report, sources and benchmarks. https://proverif.inria.fr/snp22/, 2021.

[12] R. Chadha, Ş. Ciobâcă, and S. Kremer. Automated verification of equivalence properties of cryptographic protocols. In *Programming Languages and Systems — 21th European Symposium on Programming (ESOP'12)*, volume 7211, pages 108–127. Springer, 2012.

[13] V. Cheval and B. Blanchet. Proving more observational equivalences with proverif. In *2nd International Conference on Principles of Security and Trust (POST'13)*, volume 7796 of *LNCS*, pages 226–246. Springer, Mar. 2013.

[14] V. Cheval, V. Cortier, and M. Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in proverif. In *31st IEEE Computer Security Foundations Symposium (CSF'18)*. IEEE Computer Society Press, July 2018.

[15] V. Cheval, S. Kremer, and I. Rakotonirina. Deepsec: Deciding equivalence properties in security protocols - theory and practice. In *39th IEEE Symposium on Security and Privacy (S&P'18)*, pages 525–542. IEEE Computer Society Press, May 2018.

[16] V. Cortier, S. Delaune, and J. Dreier. Automatic generation of sources lemmas in Tamarin: towards automatic proofs of security protocols. In *25th European Symposium on Research in Computer Security (ESORICS 2020)*, 2020.

[17] V. Cortier, D. Galindo, and M. Turuani. A formal analysis of the neuchâtel e-voting protocol. In *3rd IEEE European Symposium on Security and Privacy (EuroSP'18)*, pages 430–442, April 2018.

[18] V. Cortier, P. Gaudry, and S. Glondu. *Belenios: A Simple Private and Verifiable Electronic Voting System*. Springer, 2019.

[19] V. Cortier and B. Smyth. Attacking and fixing Helios: An analysis of ballot secrecy. *Journal of Computer Security*, 21(1):89–148, 2013.

[20] S. Delaune, S. Kremer, and M. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[21] S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1-2):162–20, 2006.

[22] P. Graf. Substitution tree indexing. In *Rewriting Techniques and Applications*, pages 117–131. Springer, 1995.

[23] K. Hoder and A. Voronkov. Comparing unification algorithms in first-order theorem proving. In *KI 2009: Advances in Artificial Intelligence*, pages 435–443. Springer, 2009.

[24] D. Kapur and P. Narendran. Np-completeness of the set unification and matching problems. In *Proc. of the 8th International Conference on Automated Deduction*, page 489–495. Springer, 1986.

[25] 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 2017, Paris, France, April 26-28, 2017*, pages 435–450. IEEE, 2017.

[26] N. Kobeissi, G. Nicolas, and K. Bhargavan. Noise explorer: Fully automated modeling and verification for arbitrary noise protocols. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, pages 356–370. IEEE, 2019.

[27] S. Kremer and R. Künnemann. Automated analysis of security protocols with global state. *Journal of Computer Security*, 24(5):583–616, 2016.

[28] D. L. Li and A. Tiu. Combining ProVerif and Automated Theorem Provers for Security Protocol Verification. In *CADE 2019*, pages 354–365, 2019.

[29] T. Perrin. The noise protocol framework, 2016.

[30] V. R. Pratt. Two easy theories whose combination is hard. Technical report, 1977.

[31] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 78–94. IEEE, 2012.

[32] S. Schulz. Simple and efficient clause subsumption with feature vector indexing. In *Automated Reasoning and Mathematics 2013*, pages 45–67, 2013.

[33] The Tamarin Team. Tamarin-prover manual. Available at https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf, 2019.

[34] Yubico AB. *The YubiKey Manual - Usage, configuration and introduction of basic concepts (Version 2.2)*, 2010.

Appendix A. Semantics

Let us first define the evaluation of expressions. Formally, let the ordered list of rewrite rules associated to destructor g be $\mathsf{def}(g) = [g(U_{i,1}, \ldots, U_{i,n}) \to U_i]_{i=1}^{k}$. We define the evaluation of ground expressions with one destructor at the root as follows: $g(V_1, \ldots, V_n)$ *evaluates* to V, denoted $g(V_1, \ldots, V_n) \Downarrow V$, where $V_1, \ldots, V_n, V$ are ground terms or fail, when:

- either there exists a substitution σ and $1 \leq i \leq k$ such that $U_i\sigma = V$, $(U_{i,1}, \ldots, U_{i,n})\sigma = (V_1, \ldots, V_n)$ and no previous rule could match, that is, for all $i' < i$, for all σ', $(U_{i',1}, \ldots, U_{i',n})\sigma' \neq (V_1, \ldots, V_n)$.
- or else $V = \mathsf{fail}$.

It is then extended to arbitrary ground expressions as expected: $\mathsf{fail} \Downarrow \mathsf{fail}$ and $h(D_1, \ldots, D_n) \Downarrow U$ if $D_1 \Downarrow U_1$, …, $D_1 \Downarrow U_n$ and

- if $h \in \mathcal{F}_d$ and $h(U_1, \ldots, U_n) \Downarrow U$;
- or if $h \in \mathcal{F}_c$, $U_1, \ldots, U_n$ are terms, and $U = h(M_1, \ldots, M_n)$;
- otherwise $U = \mathsf{fail}$.

The rules of the semantics are presented in Figure 3. The reduction rule NIL removes the process 0 from the multiset since it does nothing, the rule PAR applies the parallel composition and the rule REPL duplicates the process P hence modeling replication. The rule RESTR generates a new private name a' hence the condition $a' \notin \mathcal{E}$. The rule I/O allows communication between an output and input. The rule MSG indicates that the attacker is sending the message M on the channel N without interacting with the honest process. The rules LET1 and LET2 define the semantics of the evaluation of an expression D. Note that the condition $D \Downarrow M$ expresses that the evaluation of D succeeded since M is a term hence not the expression constant fail. The rule EVENT executes the event M. The other rules are explained in the body of the paper.

The syntax of processes does not explicitly include a conditional of the form $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$. This is because it can be modeled by an assignment $\mathsf{let}\ x = \mathit{equals}(M, N)\ \mathsf{in}\ P\ \mathsf{else}\ Q$ where x is fresh and $\mathit{equals}/2$ is a destructor function symbol defined by $\mathsf{def}(\mathit{equals}) = [\mathit{equals}(x, x) \to x]$. We therefore assume that $\mathcal{F}_d$ contains $\mathit{equals}/2$.

Appendix B. Satisfaction of a correspondence query

We formalize in the next definition the satisfaction of a correspondence query by a trace of the process.

Definition 3. *Let ψ be a query conclusion. Let $\tilde{\tau} = (\tau_1, \ldots, \tau_n)$ be a tuple of steps. Let T be a trace. We say that the trace T satisfies ψ at steps $\tilde{\tau}$ or before, denoted $T, \tilde{\tau} \vdash_{\leq} \psi$, when:*

- $\psi = \top$
- $\psi = \psi_1 \wedge \psi_2$, *$T, \tilde{\tau} \vdash_{\leq} \psi_1$ and $T, \tilde{\tau} \vdash_{\leq} \psi_1$*
- $\psi = \psi_1 \vee \psi_2$ *and either $T, \tilde{\tau} \vdash_{\leq} \psi_1$ or $T, \tilde{\tau} \vdash_{\leq} \psi_2$*
- $\psi = F$ *and there exists $\tau \leq \max_i(\tau_i)$ such that $T, \tau \vdash F$*
- $\psi = \phi$ *and $T, 0 \vdash \phi$*

Let $\varrho = F_1 \wedge \ldots \wedge F_n \Rightarrow \psi$ be a correspondence query. Let T be a trace. We have $T \vdash \varrho$ when for all tuples of steps $\tilde{\tau} = (\tau_1, \ldots, \tau_n)$, for all substitutions σ, if $T, \tau_i \vdash F_i\sigma$ for $i = 1 \ldots n$ then there exists σ' such that $F_i\sigma = F_i\sigma'$ for $i = 1 \ldots n$ and $T, \tilde{\tau} \vdash_{\leq} \psi\sigma'$.

Appendix C. More details on the verification procedure

The saturation procedure saturateS**.** As mentioned in Section 3.3.2, we reworked the saturation procedure used in the verification phase due to the introduction of temporal facts and temporal inequalities within the clauses.

The general structure of the saturation procedure saturateS is similar to saturate, that is, it follows the same five steps as described in Section 3.2.5; but the resolution rule and the simplification rules used in $\mathsf{simplify}_{\mathcal{L},\mathcal{L}_i}(\mathbb{C})$ are modified to handle temporal facts. Note that the definition of subsumption given Section 3.2.4 also holds with temporal facts.

In $\mathsf{saturateS}_{\mathcal{L},\mathcal{L}_i}(\mathbb{C}, \mathbb{C}_{sat})$, the resolution rule is always applied between a clause from $\mathbb{C}_{sat}$ and a clause from $\mathbb{C}$. (Recall that $\mathbb{C}_{sat}$ is generated from saturate, hence for all clauses R in $\mathbb{C}_{sat}$, $\mathrm{sel}(R) = \emptyset$.) However, clauses in $\mathbb{C}$ contain temporal facts, while clauses in $\mathbb{C}_{sat}$ do not. To perform resolution, we first add temporal annotations to clauses in $\mathbb{C}_{sat}$, relying on our main invariant introduced in Section 3.1, which guarantees strict or non-strict ordering constraints. More precisely, there exists a set of predicates $\mathcal{S}_p$ such that if a predicate of F_k is in $\mathcal{S}_p$, i.e. $\mathit{pred}(F_k) \in \mathcal{S}_p$, then the fact F_k occurs strictly before C in the trace, otherwise it only occurs before C or at the same time as C. The event predicate and all predicates occurring in lemmas, axioms, restrictions and in the conclusion of the query are in $\mathcal{S}_p$. Assuming a clause $\phi \wedge \bigwedge_{k=1}^{m} F_k \to C$ in $\mathbb{C}_{sat}$, we can then add temporal annotations to this clause as follows. We generate fresh temporal variables t_k for each fact F_k and t for C and require $t_k < t$ (resp. $t_k \leq t$) when $\mathit{pred}(F_k) \in \mathcal{S}_p$ (resp. $\mathit{pred}(F_k) \notin \mathcal{S}_p$). Formally, the clause is transformed as follows:

$$\mathcal{E}, \mathcal{P} \cup \{\!\{0\}\!\}, \mathcal{A} \to \mathcal{E}, \mathcal{P}, \mathcal{A} \quad \text{(Nil)}$$

$$\mathcal{E}, \mathcal{P} \cup \{\!\{P \mid Q\}\!\}, \mathcal{A} \to \mathcal{E}, \mathcal{P} \cup \{\!\{P, Q\}\!\}, \mathcal{A} \quad \text{(Par)}$$

$$\mathcal{E}, \mathcal{P} \cup \{\!\{!P\}\!\}, \mathcal{A} \to \mathcal{E}, \mathcal{P} \cup \{\!\{P, !P\}\!\}, \mathcal{A} \quad \text{(Repl)}$$

$$\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{new}\ a; P\}\!\}, \mathcal{A} \to \mathcal{E} \cup \{a'\}, \mathcal{P} \cup \{\!\{P\{^{a'}/_{a}\}\}\!\}, \mathcal{A} \quad \text{if } a' \notin \mathcal{E} \quad \text{(Restr)}$$

$$\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{out}(N, M); P, \mathsf{in}(N, x); Q\}\!\}, \mathcal{A} \xrightarrow{\mathrm{msg}(N,M)} \mathcal{E}, \mathcal{P} \cup \{\!\{P, Q\{^{M}/_{x}\}\}\!\}, \mathcal{A} \quad \text{(I/O)}$$

$$\mathcal{E}, \mathcal{P}, \mathcal{A} \xrightarrow{\mathrm{msg}(N,M)} \mathcal{E}, \mathcal{P}, \mathcal{A} \quad \text{if } N, M \in \mathcal{A} \quad \text{(Msg)}$$

$$\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\}, \mathcal{A} \to \mathcal{E}, \mathcal{P} \cup \{\!\{P\{^{M}/_{x}\}\}\!\}, \mathcal{A} \quad \text{if } D \Downarrow M \quad \text{(Let1)}$$

$$\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\}, \mathcal{A} \to \mathcal{E}, \mathcal{P} \cup \{\!\{Q\}\!\}, \mathcal{A} \quad \text{if } D \Downarrow \mathsf{fail} \quad \text{(Let2)}$$

$$\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{out}(N, M); P\}\!\}, \mathcal{A} \xrightarrow{\mathrm{msg}(N,M)} \mathcal{E}, \mathcal{P} \cup \{\!\{P\}\!\}, \mathcal{A} \cup \{M\} \quad \text{if } N \in \mathcal{A} \quad \text{(Out)}$$

$$\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{in}(N, x); Q\}\!\}, \mathcal{A} \xrightarrow{\mathrm{msg}(N,M)} \mathcal{E}, \mathcal{P} \cup \{\!\{Q\{^{M}/_{x}\}\}\!\}, \mathcal{A} \quad \text{if } N, M \in \mathcal{A} \quad \text{(In)}$$

$$\mathcal{E}, \mathcal{P}, \mathcal{A} \to \mathcal{E}, \mathcal{P}, \mathcal{A} \cup \{M\} \quad \text{(App)}$$
$$\text{if } M_1, \ldots, M_n \in \mathcal{A},\ f/n \in \mathcal{F}_c \cup \mathcal{F}_d \text{ and } f(M_1, \ldots, M_n) \Downarrow M$$

$$\mathcal{E}, \mathcal{P}, \mathcal{A} \to \mathcal{E} \cup \{a'\}, \mathcal{P}, \mathcal{A} \cup \{a'\} \quad \text{if } a' \notin \mathcal{E} \quad \text{(New)}$$

$$\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{event}(ev); P\}\!\}, \mathcal{A} \xrightarrow{\mathrm{event}(ev)} \mathcal{E}, \mathcal{P} \cup \{\!\{P\}\!\}, \mathcal{A} \quad \text{(Event)}$$

Figure 3. Transitions between configurations.

$$\frac{\phi \wedge \bigwedge_{k=1}^{m} F_k \to C \qquad t_1, \ldots, t_m, t \text{ fresh} \qquad \text{if } pred(F_k) \in \mathcal{S}_p \text{ then } \sim_k = < \text{ else } \sim_k = \leq}{\phi \wedge \bigwedge_{k=1}^{m} F_k@t_k \wedge t_k \sim_k t \to C@t}$$

Then, we apply the standard resolution rule (Res) between the obtained clause and a clause in $\mathbb{C}$.

The rules Lem($\mathcal{L}$) and Ind($\mathcal{L}_i$) are modified as follows. Consider a lemma $\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j$ in $\mathcal{L}$ and a clause $R = F'_1@t'_1 \wedge \ldots \wedge F'_n@t'_n \wedge H \wedge \phi \to C@(t_1, \ldots, t_\ell)$. When applying the lemma on the temporal facts $F'_1@t'_1, \ldots, F'_n@t'_n$ in R, we want to generate clauses in which we add ψ_j in the hypothesis of the clause R. Hence we need to order the events in ψ_j w.r.t. $C@(t_1, \ldots, t_\ell)$. However, the satisfaction of correspondence queries, and in particular lemmas, only guarantees that the events in ψ_j occur before at least one of the facts $F_1, \ldots, F_n$. Therefore, we consider a formula $\Phi[t]$ that is guaranteed by this constraint, that is, $\Phi[t]$ is such that $\phi \wedge t \leq \max(t'_1, \ldots, t'_n) \models \Phi[t]$. The formula $\Phi[t]$ is then added in the hypotheses of R for all events in ψ_j. This is formalized by the transformation $\lceil \psi_j\sigma \rceil^{s}_{\Phi[\cdot]}$ that replaces in $\psi_j\sigma$ every fact F by $\lceil F \rceil^{s}@t' \wedge \Phi[t']$ where t' is a fresh temporal variable. The complete rule Lem($\mathcal{L}$) for saturateS is given as follows:

$$\frac{\mathbb{C} \cup \{F'_1@t'_1 \wedge \ldots \wedge F'_n@t'_n \wedge H \wedge \phi \to C@(t_1, \ldots, t_\ell)\} \qquad (\textstyle\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j) \in \mathcal{L} \qquad \forall k,\ F'_k = \lceil F_k\sigma \rceil^{s} \qquad \phi \wedge t \leq \max(t'_1, \ldots, t'_n) \models \Phi[t]}{\mathbb{C} \cup \{H \wedge \lceil \psi_j\sigma \rceil^{s}_{\Phi[\cdot]} \to C\}_{j=1}^{m}}$$

The rule is sound for any choice of $\Phi[t]$ that satisfies the constraint $\phi \wedge t \leq \max(t'_1, \ldots, t'_n) \models \Phi[t]$ but of course, a more precise $\Phi[t]$ yields a more precise clause. In our implementation, we simply define $\Phi[t]$ as the conjunction of inequalities $t \leq t'$ for any t' such that ϕ implies $t'_1 \leq t' \wedge \ldots \wedge t'_n \leq t'$ (and we consider the strict inequality when all inequalities are strict).

When proving a query by induction, we rely on a multiset ordering of the steps of the trace on which the facts of query's premise are satisfied. In a clause $R = F'_1@t'_1 \wedge \ldots \wedge F'_n@t'_n \wedge H \wedge \phi \to C@(t_1, \ldots, t_\ell)$ obtained during the saturation procedure saturateS, these steps are represented by the temporal variables $t_1, \ldots, t_\ell$. Thus, to apply an inductive hypothesis represented by an inductive lemma $\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j$ in $\mathcal{L}_i$ on the facts $F'_1, \ldots, F'_n$ of R, we need to verify that ϕ ensures that the facts $F'_1@t'_1, \ldots, F'_n@t'_n$ occur strictly before the facts represented by $C@(t_1, \ldots, t_\ell)$, denoted $\phi \models \{t'_1, \ldots, t'_n\} < \{t_1, \ldots, t_\ell\}$ (for the multiset ordering). Formally, this can be checked by verifying that

1) either for all $k \in \{1, \ldots, n\}$, there exists $k' \in \{1, \ldots, \ell\}$ such that $\phi \models t'_k < t_{k'}$
2) or $n \leq \ell$ and there exist distinct $r_1, \ldots, r_n$ in $\{1, \ldots, \ell\}$ such that:
   a) for all $k \in \{1, \ldots, n\}$, $\phi \models t'_k \leq t_{r_k}$;
   b) if $n = \ell$ then there exists $k \in \{1, \ldots, n\}$ such that $\phi \models t'_k < t_{r_k}$.

The complete rule Ind($\mathcal{L}_i$) for saturateS is given as follows:

$$\frac{\mathbb{C} \cup \{F'_1@t'_1 \wedge \ldots \wedge F'_n@t'_n \wedge H \wedge \phi \to C@(t_1, \ldots, t_\ell)\} \qquad (\textstyle\bigwedge_{i=1}^{n} F_i \Rightarrow \bigvee_{j=1}^{m} \psi_j) \in \mathcal{L}_i \qquad \forall k,\ F'_k = \lceil F_k\sigma \rceil^{s} \qquad \phi \models \{t'_1, \ldots, t'_n\} < \{t_1, \ldots, t_\ell\} \qquad \phi \wedge t \leq \max(t'_1, \ldots, t'_n) \models \Phi[t]}{\mathbb{C} \cup \{H \wedge \lceil \psi_j\sigma \rceil^{s}_{\Phi[\cdot]} \to C\}_{j=1}^{m}}$$

Verifying that a clause R entails a query ϱ. As mentioned in Section 3.3.2, we need to check that all possible instantiations of the clause $R = \phi \wedge H \to$

$and_{F'_1,\ldots,F'_n}@(t'_1,\ldots,t'_n)$ satisfy the query ϱ. To check that an event or a message fact of the conclusion of ϱ is satisfied, we verify that it either occurs in the hypotheses of R or in its conclusion. However, for an attacker fact att, the satisfaction relation $T,\tau \vdash \mathrm{att}(M)$, defined in Section 2.3.1, holds if the attacker is able to build M from its knowledge at step τ using only the semantic rules APP and NEW. These two semantics rules are in fact represented by the four first Horn clauses of the attacker clauses defined in Section 3.1, that we denote $\mathbb{C}_{\text{APP,NEW}}$. Thus, to show that an attacker fact $\mathrm{att}(M)$ occurring in the conclusion of the query is satisfied by the clause R, we verify that the fact att is derivable from the hypotheses and conclusion of R using only the Horn clauses in $\mathbb{C}_{\text{APP,NEW}}$.

Considering the query in its disjunctive normal form, that is $\varrho = \bigwedge_{k=1}^{n} F_k@t_k \Rightarrow \bigvee_{k'=1}^{m} \phi_{k'} \wedge H_{k'}$ where $H_{k'}$ is a conjunction of facts, we can formally define $R \models \varrho$ to hold when there exist σ a substitution and $k' \in \{1,\ldots,m\}$ such that, denoting $H_{k'} = G_1@t''_1 \wedge \ldots \wedge G_\ell@t''_\ell$, there exist $H'_1,\ldots,H'_\ell$ such that:

- $(F_1@t_1)\sigma = F'_1@t'_1,\ldots,(F_n@t_n)\sigma = F'_n@t'_n$;
- for all $r \in \{1,\ldots\ell\}$, $G_r\sigma$ is derivable from $\mathbb{C}_{\text{APP,NEW}} \cup \{\rightarrow F' \mid F'@t' \in H'_r\}$ with $H'_r \subseteq H \cup \{F'_1@t'_1, \ldots, F'_\eta@jt'_n\}$;
- $\phi \wedge \bigwedge_{r=1}^{\ell} t''_r = \max(t' \mid F'@t' \in H'_r) \models \phi_{k'}\sigma$.

Complete coverage. Proving that $R \models \varrho$, as presented above, is not always sufficient in particular when the query contains disequalities or inequalities combined with disjunction. Consider the query $\mathsf{event}(A(x)) \Rightarrow \mathsf{event}(B(a)) \vee x \neq a$ and assume the saturation performed by the verification procedure generates the clause $\text{s-event}(B(x)) \rightarrow \text{m-event}(A(x))$. (We omit temporal variables $@t$, which are not necessary here.) In ProVerif 2.00, the verification would fail since the hypothesis of the clause does not imply $x \neq a$ and $\mathsf{event}(B(a))$ does not match $\mathsf{event}(B(x))$. In this paper, based on the algorithm presented in [14], we prove the query on a *complete coverage* of the clauses, intuitively corresponding to a partition of all instantiations of the clauses. In our example, the clause $\text{s-event}(B(x)) \rightarrow \text{m-event}(A(x))$ is split into $\text{s-event}(B(x)) \wedge x = a \rightarrow \text{m-event}(A(x))$ and $\text{s-event}(B(x)) \wedge x \neq a \rightarrow \text{m-event}(A(x))$. The latter implies $x \neq a$ trivially, whereas the former is simplified into $\text{s-event}(B(a)) \rightarrow \text{m-event}(A(a))$ and so also satisfies the query.

Formally, a complete coverage of a clause $R = H \rightarrow C$ is a set of clauses $\mathbb{C} = \{H \wedge \phi_i \rightarrow C\}_{i=1}^{n}$ such that the variables of ϕ_i occur in H or C, and $\phi_1 \vee \ldots \vee \phi_n \equiv \top$. Note that $\{R\}$ is always a completely coverage of R. We can therefore extend the notation $R \models \varrho$ to hold when there exists a complete coverage $\mathbb{C}$ of R such that for all $R' \in \mathbb{C}$, $R' \models \varrho$.

Temporal queries. Example 9 illustrates the proof of a temporal correspondence query. Note that the clauses obtained by the saturation procedure saturateS can only contain inequalities between time variables that express that some hypothesis of the clause happens before (or strictly before) some conclusion. Hence, we can directly prove that some conclusion of a query happens before some premise of the query. Moreover, we can also prove that some premise of the query happens before another premise by taking advantage of the fact that the event corresponding to the premise will appear not only in the conclusion of the clause, but also in the hypothesis of the clause if its execution is guaranteed before another premise. The temporal inequalities in the clause then allow us to prove the desired ordering of events.

However, the constraint inequalities in a clause are not sufficient to directly order two conclusions of a query as in $\mathrm{event}(A(x))@t_1 \Rightarrow \mathrm{event}(B(x))@t_2 \wedge \mathrm{event}(C(x))@t_3 \wedge t_2 < t_3$. We need to go further. Essentially, we first prove the query without the ordering constraint $t_2 < t_3$: $\mathrm{event}(A(x))@t_1 \Rightarrow \mathrm{event}(B(x))@t_2 \wedge \mathrm{event}(C(x))@t_3$. Then we prove a modified query in which the conclusion that should happen last, here $\mathrm{event}(C(x))@t_3$, is added in the premise of the query: $\mathrm{event}(A(x))@t_1 \wedge \mathrm{event}(C(x))@t_3 \Rightarrow \mathrm{event}(B(x))@t_2 \wedge \mathrm{event}(C(x))@t_3 \wedge t_2 < t_3$. On this query, the temporal constraints of the clause can allow us to prove $t_2 < t_3$ because $\text{m-event}(C(x))$ will appear in the conclusion of the clause and $\text{s-event}(B(x))$ in its hypothesis. The combination of the two queries allows us to prove the initial query: if A happens, then B and C happen; and if A and C happen, then B happens before C; so if A happens, then B and C happen and B happens before C. This is the technique we use to prove the *nested queries* of ProVerif, which also provide this kind of ordering constraints. It improves over the proof of nested queries in ProVerif 2.00. This aspect is omitted in Algorithm 2 for simplicity.

Proof sketch of Theorem 2 If a fact holds in the trace, then it is derivable from the generated clauses, by a derivation that matches the trace: the intermediate facts in the derivation also hold in the trace [11, Theorem 1]. Assuming axioms, lemmas, and restrictions hold on the trace and inductive lemmas hold on a strict prefix, derivability is preserved by saturation [11, Theorem 2]. Consider the query $\varrho = \bigwedge F_i \Rightarrow \psi$ and suppose that for all clauses R in $\mathbb{C}_s$, $R \models \varrho$. If an instance of the premise $\bigwedge F_i$ of ϱ holds in a trace, then this instance is derivable from the clauses in $\mathbb{C}_s$ by a derivation that matches the trace. In particular, the last clause of the derivation is in $\mathbb{C}_s$. Since the derivation matches the trace, the hypothesis of R holds in the trace. The conclusion of R corresponds to an instance of $\bigwedge F_i$ and because $R \models \varrho$, the hypotheses of this clause entail the conclusion ψ of ϱ. Hence ψ holds in the trace. Therefore, the query ϱ is satisfied. The algorithm `prove` uses this technique for proving each query and inductive lemma.