A little more conversation, a little less action, a lot more satisfaction: Global states in ProVerif.

Vincent Cheval
INRIA, LORIA, France

Véronique Cortier
CNRS, LORIA, France

Mathieu Turuani
INRIA, LORIA, France

***Abstract*—ProVerif is a popular tool for the fully automatic analysis of security protocols, offering very good support to detect flaws or prove security. One exception is the case of protocols with global states such as counters, tables, or more generally, memory cells. ProVerif fails to analyse such protocols, due to its internal abstraction.**

Our key idea is to devise a generic transformation of the security properties queried to ProVerif. We prove the soundness of our transformation and implement it into a front-end GSVerif. Our experiments show that our front-end (combined with ProVerif) outperforms the few existing tools, both in terms of efficiency and protocol coverage. We successfully apply our tool to a dozen of protocols of the literature, yielding the first fully automatic proof of a security API and a payment protocol of the literature.

1. Introduction

Formal methods have been successful in the analysis of security protocols. They provide a nice level of abstraction, that allow good automation while being sufficiently precise to detect logical flaws. ProVerif [9] is a popular tool for the automatic analysis of security protocols. It has been successfully applied to hundreds of protocols ranging from TLS [8], web services [7], secure messaging [26], to voting protocols [19], [17].

The reasons of success are the flexibility and efficiency of ProVerif: ProVerif can cover a wide class of cryptographic primitives and various protocols structures (with else branches, private channels, etc.), yielding a very flexible tool that can be used to analyse various encoding of protocols and security properties. ProVerif is also one of the only tools that can analyse an unbounded number of sessions, together with Scyther [18], Maude-NPA [21], and Tamarin [33].

While ProVerif can handle a wide range of primitives and complex protocols, one well identified limitation is the case of protocols with global states. Global states appear in many examples. For example, in contract signing, an authority would issue a contract only if it has not previously aborted the transaction [23]. Secure device APIs (eg PKCS#11 [32] or TPM [2]) may or may not execute a command depending on the current internal state (status of a key, previous history). Several protocols include counters (e.g. Yubikey [35], avionic protocols [11]) and their security rely on the fact that a counter cannot take twice the same value. In voting protocols, the voting server typically maintains a table that contains the list of voters that have voted so far, which can be crucial when revotes are forbidden [22]. Unfortunately, in most of these cases, ProVerif immediately finds false attacks and therefore fails to prove security.

Why does ProVerif fail to handle global states? This is due to its internal abstraction: ProVerif takes as input a protocol specified in (a dialect of) the applied-pi calculus [4] and translates it into Horn clauses. This yields several over-approximations. In particular, Horn clauses can be applied an arbitrary number of times, for arbitrary instantiations.

This issue is well known and there have been several attempts to add global states to ProVerif or to other tools of the literature.

- StatVerif [5] introduces an extension of the applied pi-calculus to specify protocols with states and automatically translates it into Horn Clauses. The main idea is that the predicate $\mathsf{attacker}(M)$, which models that the attacker knows M, is replaced by $\mathsf{attacker}(v_1, \dots, v_k, M)$, which models that the attacker knows M when cell s_1 has value v_1, ..., and cell s_k has value v_k. It is limited to a finite number of cells and runs very quickly into state explosion.
- SetPi [13] proposes another extension of the applied-pi calculus to define sets and set membership. Such sets can be used to store values, provided the values are atomic. SetPi again translates the input language into Horn Clauses. It assumes that protocol messages follow a strict format (possibly ruling out type flaw attacks) and often requires several protocol abstractions.
- AIF-ω [30] is a recent tool that follows the approach developped by SetPi. Compared to SetPi, it typically handles better cases where operations on sets are not locked by the protocol (for example when several processes may read or write a state at the same time).
- SAPIC [27] relies on a different tool, Tamarin, that offers both an automatic and interactive mode. SAPIC takes as input an extension of the applied pi-calculus, extended to global states and translates it into Tamarin, such that the resulting model is well suited for Tamarin. In particular, part of the semantics is passed directly to the security property (instead of the protocol rules).

Our contribution. The contribution of the paper is to significantly enhance ProVerif in order to handle both global states and natural numbers. Our technique is flexible and covers various flavours of global states: for example private channels, cells, tables, or counters.

Our first idea is simple and therefore easy to use and adapt. Instead of querying ProVerif whether a protocol P satisfies a property ϕ, we query instead "$\phi \vee$ *some action has been taken twice*". Provided that we can ensure (typically through simple syntactic checks) that this action is actually unique, we can immediately deduce that ϕ holds.

More formally, we devise several formulas ϕ_{act}, ϕ_{com}, ϕ_{cell}, $\phi_{counter}$, ϕ_{table} corresponding respectively to "fresh actions" (when an action is guarded by a fresh nonce/key), private channels, cells, counters, and tables. Then we automatically annotate protocols with events that record for example when a channel is used, with which message, and possibly with some freshness indicator. We then formally prove the soundness of our transformation. In other words, we prove that whenever $\overline{P}^{act} \models \phi \vee \phi_{\mathsf{act}}$ then $P \models \phi$, where $\overline{P}^{act}$ is the protocol P annotated with some events (and similarly for the other formulas).

Maybe surprisingly, ProVerif can very efficiently prove properties like $\phi \vee \phi_{\mathsf{act}}$. This is due to the fact that, after saturating the set of Horn clauses, ProVerif can show that any trace (derivation) where ϕ is not satisfied is such that two "unique" actions have taken place, hence the conclusion.

Our second and main contribution is to enrich ProVerif with natural numbers together with equalities and inequalities. Formally, we introduce a new type `nat` together with predicates $=, \neq, \geq$ with the expected semantics. Our motivation is twofold. First natural numbers arise naturally in the case of counters. For example, a server would typically accept a request containing a counter only if this one is "fresh", that is greater than the current value of the counter stored on the server. Second, this allows us to express finer properties, useful for some of our transformations. For example, we can characterize more precisely when an event occurs *before* another one. And of course, natural numbers may be used in other contexts.

Running directly ProVerif on protocols with naturals quickly yields false attacks again. Therefore, we enrich ProVerif's procedure with the algorithm of Pratt [31], for checking satisfiability of inequalities between naturals. This allows ProVerif to detect that many clauses are actually unsatisfiable. We also improve the behaviour of ProVerif when proving disjunctions. Indeed, ProVerif typically fails to prove a query of the form $E(x) \Rightarrow x = a \vee x \neq a$ (where E is some event). This is due to the fact that ProVerif actually tries to prove $E(x) \Rightarrow x = a$ or $E(x) \Rightarrow x \neq a$. We therefore introduce a more precise treatment of disjunctions. These two improvements are of independent interest and could be added to the main development of ProVerif.

Implementation and experimentation. We have implemented our approach into an extension of ProVerif, GSVerif, that given a protocol P, automatically annotates it with events whenever applicable and tries to prove $\phi \vee \phi'$ instead of ϕ. We have successfully tested GSVerif on various protocols of the literature, yielding the first fully automatic proof of a security API [27] and a payment protocol [15], two protocols previously analysed in the literature. GSVerif demonstrates a major improvement compared to StatVerif, SetPi, or SAPIC, in terms of efficiency or simply covering examples that could not be handled so far. In previous studies [27], [5], [13], only a few simple protocols (2 to 4) were analysed. We conduct a systematic comparison of the existing tools on a dozen protocols of the literature, including a voting protocol (for verifiability properties) and a payment protocol. This extended study offers a better understanding of the scope of existing approaches. To our knowledge, we provide the first automatic proof in ProVerif of protocols with a true representation of counters. In previous approaches, counters were abstracted by fresh nonces or by arbitrary values controlled by the attacker (avionic protocol [11]).

During our study, we also discovered two new attacks against the Key Registration protocol of [13] and a recent mobile payment protocol [15] (for some choice of implementation).

Interestingly, when GSVerif fails to prove the security of a protocol, it is still possible to apply our technique by hand, by designing another formula ψ well adapted to the protocol. This is for example the case of the YubiKey protocol. This authentication protocol strongly relies on counters to ensure that the server will never accept the same authentication twice. Despite our transformations, GSVerif cannot automatically prove its security because it requires some inductive reasoning. So instead of querying ProVerif whether some authentication property ϕ_{auth} holds, we query the following three properties: $\psi(0)$, $\forall n\ \psi(n) \Rightarrow \psi(n+1)$, and $\phi_{auth} \vee (\exists n \neg\psi(n))$. These three properties can be automatically proved by ProVerif. We can then straightforwardly conclude that ϕ_{auth} holds. Similarly, it is always possible to add a succession of intermediate formulas, e.g. ϕ_1, $\phi_2 \vee \neg\phi_1$, $\dots$, $\phi \vee \neg\phi_n$ instead of ϕ. Therefore our approach not only yields a major improvement over the tools StatVerif, SetPi, and SAPIC, for global states but also adds a flavour of interactivity to the ProVerif tool.

Related Work. Several tools have been developed for analyzing protocols for a bounded number of sessions (e.g. Avispa [6] or Scyther [18]). When the number of sessions is bounded, it is easy to model global states by simply enumerating all possible cases. However, these tools suffer from a state-explosion issue and cannot prove security in the general case. Scyther [18] can also prove protocols for an unbounded number of sessions as well as Maude-NPA [21] but we are not aware of any attempt to use them to prove protocols with global states (for an unbounded number of sessions). Tamarin [33] is a recent tool that allows the user to enter an interactive mode when Tamarin fails to prove security automatically. In the interactive mode, Tamarin can in theory prove almost any protocol (possibly at the cost of heavy user interactions) so we focus here on the automatic mode. There are two main approaches to use Tamarin with global states. The first one is a direct encoding in Tamarin, which supports built-in memory cells (through linear facts) and counters may be directly encoded using multisets [29], [1]. A second approach is the tool SAPIC [27], that automatically provides an appropriate encoding for Tamarin, as already discussed. The two approaches closest to our work are StatVerif and SetPi that we have discussed in details above. Another advantage of our approach is that we do not impose any particular encoding for states: the user is free to encode states at her will since the input language remains the applied-pi calculus, allowing a simple integration into ProVerif, a tool already well understood by many users.

2. Overview

We overview here our main transformations. We leave the ones relying on natural numbers to Section 6. The corresponding ProVerif files (of the initial and transformed examples) can be found here [3].

2.1. Unique action

A first simple example where ProVerif fails due to states is when the security of a protocol relies on the fact that some rule is executed at most once. The issue is well illustrated by the following mock example.

$$\begin{aligned} A = \;& \mathsf{out}(c, \mathsf{enc}(s, (k_1, k_2)));\\ & \mathsf{out}(c, \mathsf{enc}(k_1, k));\\ & \mathsf{out}(c, \mathsf{enc}(k_2, k))\\ B = \;& \mathsf{in}(c, x);\\ & \mathsf{let}\ y = \mathsf{dec}(x, k)\ \mathsf{in}\\ & \mathsf{out}(c, y) \end{aligned}$$

Alice sends a secret s, encrypted with the pair of k_1 and k_2 and she also sends both k_1 and k_2 encrypted by a fresh key k. Bob will decrypt any message encrypted by k but *at most once* for this key k. This corresponds to the case where e.g. a server will answer some particular request at most once. When we specify this protocol in ProVerif, it is internally translated into the following clauses.

$$\begin{aligned} &\rightarrow \mathsf{attacker}(\mathsf{enc}(s, (k_1, k_2)))\\ &\rightarrow \mathsf{attacker}(\mathsf{enc}(k_1, k))\\ &\rightarrow \mathsf{attacker}(\mathsf{enc}(k_2, k))\\ \mathsf{attacker}(\mathsf{enc}(x, k)) &\rightarrow \mathsf{attacker}(x) \end{aligned}$$

ProVerif also includes clauses for the attacker, in particular the ability to concatenate messages and to decrypt.

$$\begin{aligned} \mathsf{attacker}(x) \wedge \mathsf{attacker}(y) &\rightarrow \mathsf{attacker}((x, y))\\ \mathsf{attacker}(\mathsf{enc}(x, y)) \wedge \mathsf{attacker}(y) &\rightarrow \mathsf{attacker}(x) \end{aligned}$$

Now the question is whether ProVerif can prove the secrecy of s ? The answer is no: s *is* deducible since clauses do entail $\mathsf{attacker}(s)$. Therefore ProVerif finds false attacks on such examples, and thus fails to prove security.

Continuing our example, instead of querying $\mathsf{attacker}(s)$, we can query $\mathsf{attacker}(s) \vee \phi_{\mathsf{act}}$ where

$$\phi_{\mathsf{act}} = \mathsf{UAction}(x, y) \wedge \mathsf{UAction}(x, y') \wedge y \neq y'$$

where $\mathsf{UAction}(\mathsf{st}, m)$ is an event added in our protocol that records that some input rule has received message m at some (fresh) step st. So process B is enriched with additional events.

$$\begin{aligned} B_{\mathsf{act}} = \;& \mathsf{new}\ \mathsf{st} : \mathsf{stamp};\\ & \mathsf{in}(c, x);\\ & \mathsf{event}(\mathsf{UAction}(st, x));\\ & \mathsf{let}\ y = \mathsf{dec}(x, k)\ \mathsf{in}\\ & \mathsf{out}(c, y) \end{aligned}$$

Note that it does not change its execution (besides the additional events). The fact that stamp st is fresh guarantees that $\mathsf{UAction}(x, y) \wedge \mathsf{UAction}(x, y') \wedge y \neq y'$ is always false and therefore $\mathsf{attacker}(s) \vee \phi_{\mathsf{act}}$ actually guarantees $\mathsf{attacker}(s)$.

More generally, querying $\phi \vee \phi_{\mathsf{act}}$ instead of ϕ is sound as soon as we can guarantee that st is fresh each time $\mathsf{UAction}(\mathsf{st}, t)$ is issued. This is formalized and proved in Section 4.1.

2.2. Private channels

Our transformation on the unicity of an input action may not be sufficient, in particular in the presence of private channels.

Continuing the previous example, we can write a process similar to A, using a private channel as a token.

$$\begin{aligned} A' = \;& \mathsf{new}\ d : \mathsf{channel}; (\mathsf{out}(d, k)\\ & \mid \mathsf{in}(d, x); \mathsf{out}(c, k_1)\\ & \mid \mathsf{in}(d, x); \mathsf{out}(c, k_2)\\ & \mid \mathsf{out}(c, \mathsf{enc}(s, (k_1, k_2)))) \end{aligned}$$

A' emits once on a private channel d. Both keys k_1 and k_2 can be released but they each require to receive something on channel d. Therefore the attacker can obtain at most one of the two keys, which protects the secrecy of s. This is a mock example for the sake of the presentation but the same kind of behaviour happens when private channels are used as tokens, to prevent some action to occur before another one.

Once again ProVerif is not able to prove secrecy of s, even with the event annotation presented in Section 2.1 and ϕ_{act}. This is due to the fact that the input message does not vary. Here, we need to express that each input may correspond to at most one output. Therefore, we introduce fresh identifiers for any input and output. The identifier st of an output is also sent together with the message. Then for any input (of identifier st′), the association between st and st′ is recorded through the event $\mathsf{UComm}(\mathsf{st}', \mathsf{st})$. On our example, this results into the following process.

$$\begin{aligned} A'_{\mathsf{com}} = \;& \mathsf{new}\ d : \mathsf{channel}; (\\ & \quad \mathsf{new}\ \mathsf{st}_0 : \mathsf{stamp}; \mathsf{out}(d, (\mathsf{st}_0, k))\\ & \mid \mathsf{new}\ \mathsf{st}_1 : \mathsf{stamp}; \mathsf{in}(d, (x_1 : \mathsf{stamp}, x));\\ & \qquad \mathsf{event}(\mathsf{UComm}(x_1, \mathsf{st}_1)); \mathsf{out}(c, k_1)\\ & \mid \mathsf{new}\ \mathsf{st}_2 : \mathsf{stamp}; \mathsf{in}(d, (x_2 : \mathsf{stamp}, x));\\ & \qquad \mathsf{event}(\mathsf{UComm}(x_2, \mathsf{st}_2)); \mathsf{out}(c, k_2)\\ & \mid \mathsf{out}(c, \mathsf{enc}(s, (k_1, k_2)))) \end{aligned}$$

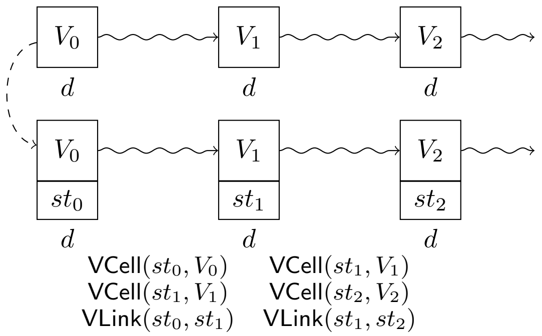

V_0
V_1
V_2
d
d
d
V_0
V_1
V_2
st_0
st_1
st_2
d
d
d
$\mathsf{VCell}(st_0, V_0)$
$\mathsf{VCell}(st_1, V_1)$
$\mathsf{VLink}(st_0, st_1)$
$\mathsf{VCell}(st_1, V_1)$
$\mathsf{VCell}(st_2, V_2)$
$\mathsf{VLink}(st_1, st_2)$

Figure 1. Transformation for cells

For any two events $\mathsf{event}(\mathsf{UComm}(\mathsf{st}_1, \mathsf{st}_2))$ and $\mathsf{event}(\mathsf{UComm}(\mathsf{st}'_1, \mathsf{st}'_2)$, we have $\mathsf{st}_1 = \mathsf{st}'_1$ iff $\mathsf{st}_2 = \mathsf{st}'_2$ $(*)$. This means that we can query $\mathsf{attacker}(s) \vee \phi_{\mathsf{com}}$ for A'_{com} instead of querying $\mathsf{attacker}(s)$ for A', where ϕ_{com} is the following formula:

$$\begin{aligned} &(\mathsf{event}(\mathsf{UComm}(x, y)) \wedge \mathsf{event}(\mathsf{UComm}(x, z)) \wedge y \neq z) \\ \vee\ &(\mathsf{event}(\mathsf{UComm}(y, x)) \wedge \mathsf{event}(\mathsf{UComm}(z, x)) \wedge y \neq z) \end{aligned}$$

More generally, querying $\phi \vee \phi_{\mathsf{com}}$ instead of ϕ is sound as soon as we can guarantee that $(*)$ holds, provided that channel d is *strongly private*, that is (intuitively) a channel that can never be deduced by the attacker.

2.3. Cells

Another common example of states is the use of cells. A memory cell d stores a value (true, false, init, a key, etc.) that evolves with time: the cell contains v_0, then v_1, then v_2, etc. Two processes may not access a cell at the same time. The most standard way to model cells in the applied pi-calculus is through private channels (e.g. in [5]) but of course, ProVerif very quickly runs into false attacks.

To avoid such false attacks, we can add a stamp st for each new value of the cell. Then for each process that may access the cell, it typically reads the value v of the cell, together with its stamp st, does some computation and possibly other input/output actions and finally write a new value v' to the cell, associated to a new stamp st'. We annotate such a round of read/process/write actions with the events $\mathsf{VCell}(v, \mathsf{st})$, $\mathsf{VCell}(v', \mathsf{st}')$, and $\mathsf{VLink}(\mathsf{st}, \mathsf{st}')$, as illustrated in Figure 1.

If the private channel d behaves indeed as a cell (at most one output after an input, which can be easily checked syntactically), then we can prove that

- for any two events $\mathsf{event}(\mathsf{VLink}(\mathsf{st}_1, \mathsf{st}_2))$ and $\mathsf{event}(\mathsf{VLink}(\mathsf{st}'_1, \mathsf{st}'_2)$ then $\mathsf{st}_1 = \mathsf{st}'_1$ iff $\mathsf{st}_2 = \mathsf{st}'_2$;
- for any two events $\mathsf{event}(\mathsf{VCell}(\mathsf{st}_1, M))$ and $\mathsf{event}(\mathsf{VCell}(\mathsf{st}_1, N))$ then $N = M$.

Therefore, we can safely query $\phi \vee \phi_{\mathsf{cell}}$ instead of ϕ where ϕ_{cell} is defined as

$$\begin{aligned} &(\mathsf{event}(\mathsf{VLink}(x, y)) \wedge \mathsf{event}(\mathsf{VLink}(x, z)) \wedge y \neq z) \\ \vee\ &(\mathsf{event}(\mathsf{VLink}(y, x)) \wedge \mathsf{event}(\mathsf{VLink}(z, x)) \wedge y \neq z) \\ \vee &(\mathsf{event}(\mathsf{VCell}(x, y)) \wedge \mathsf{event}(\mathsf{VCell}(x, y')) \wedge y \neq y') \end{aligned}$$

This can greatly help ProVerif to prove security of protocols with memory cells (e.g. TPM or PKCS#11) as we shall see in our experimentation section (Section 7). Some protocols may require the introduction of natural numbers, either because they make use of counters or because we need to express more precise relations between old and new values. This will be presented in Section 5 and 6.

3. ProVerif syntax and semantics

For the sake of readability, we only present parts of the syntax and semantics of ProVerif that are relevant to our work. A complete presentation of the syntax and semantics of ProVerif can be found in [10], [12].

3.1. Syntax

We assume a set $\mathcal{V}$ of variables, a set $\mathcal{N}$ of names, a set $\mathcal{T}$ of types. By default in ProVerif, types include channel for channel's names, bitstring for bitstrings and bool for booleans. The syntax for *terms*, *expressions*, and *processes* is displayed in Figure 2.

M, N ::=	terms
x	variable ($x \in \mathcal{V}$)
n	name ($n \in \mathcal{N}$)
$f(M_1, \ldots, M_k)$	applied $f \in \mathcal{C}$
D::=	expressions
M	term
$h(D_1, \ldots, D_k)$	applied $h \in \mathcal{C} \cup \mathcal{D}$
fail	failure
P, Q::=	processes
0	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P$	assignment
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
$\mathsf{event}(ev(M_1, \ldots, M_n)); P$	event
$\mathsf{get}\ tbl(x_1 : T_1, \ldots, x_n : T_n)$ $\mathsf{suchthat}\ D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	table lookup
$\mathsf{insert}\ tbl(M_1, \ldots, M_n); P$	table insertion

Figure 2. Syntax of the core language of ProVerif.

Terms and expressions. Crytographic primitives are represented by function symbols, split into two sets of constructors $\mathcal{C}$ (e.g. encryption) and destructors $\mathcal{D}$ (e.g. decryption) respectively. Terms are built over names, variables, and constructors and represent actual messages sent over the network, while expressions may also contain destructors and represent cryptographic computations. Function symbols are given with their types: $g(T_1, \ldots, T_n) : T$ means that the function g takes n arguments as input of types respectively $T_1, \ldots, T_n$ and returns a result of type T. A substitution is a mapping from variables to terms, denoted $\{U_1/x_1, \ldots, U_n/x_n\}$. The application of a substitution σ to a term U, denoted $U\sigma$, is obtained by replacing variables by the corresponding terms and is defined as usual. We only consider well typed substitutions.

The evaluation of an expression is defined through rewrite rules. Specifically, each destructor g is associated with a list of rewrite rules $\mathsf{def}(g) = [g(M_{i,1}, \ldots, M_{i,n}) \to M_i]_{i=1}^k$, over terms. The evaluation of an expression is as follows: $g(D_1, \ldots, D_n)$ *evaluates* to U, denoted $g(D_1, \ldots, D_n) \Downarrow U$, when

- $\forall i,\ D_i \Downarrow M_i$, and g is a constructor ($g \in \mathcal{C}$) and $U = g(M_1, \ldots, M_n)$; or g is a destructor ($g \in \mathcal{D}$) with $\mathsf{def}(g) = [g(M'_{i,1}, \ldots, M'_{i,n}) \to M'_i]_{i=1}^k$ and there exist a substitution σ and $1 \leq i \leq k$ such that $M_j = M'_{i,j}\sigma$, $U = M'_i\sigma$ and for all $i' < i$, for all σ', $(M_1, \ldots, M_n) \neq (M_{i',1}, \ldots, M_{i',n})\sigma'$.
- $U = \mathsf{fail}$ otherwise, i.e. the evaluation failed.

Example 1. The standard symmetric encryption primitives can be easily modeled by considering a constructor $\mathsf{enc}(\mathsf{bitstring}, \mathsf{bitstring}) : \mathsf{bitstring}$ in $\mathcal{C}$ for encryption and a destructor $\mathsf{dec}(\mathsf{bitstring}, \mathsf{bitstring}) : \mathsf{bitstring}$ in $\mathcal{D}$ for decryption with the following rewrite rule:

$$\mathsf{def}(\mathsf{dec}) = [\mathsf{dec}(\mathsf{enc}(x, y), y) \to x].$$

Similarly, pair and projections are represented by the constructor $\mathsf{pair}(\mathsf{bitstring}, \mathsf{bitstring}) : \mathsf{bitstring} \in \mathcal{C}$ and the destructors $\mathsf{proj}_i : \mathsf{bitstring} : \mathsf{bitstring} \in D$, $i \in \{1, 2\}$, with the following rewrite rules:

$$\begin{aligned} \mathsf{proj}_1(\mathsf{pair}(x, y)) &\to x \\ \mathsf{proj}_2(\mathsf{pair}(x, y)) &\to y \end{aligned}$$

In ProVerif, the pair operator is actually built in and $\mathsf{pair}(m_1, m_2)$ is denoted (m_1, m_2). ▶

Processes. Most of the syntax of processes used by ProVerif comes from the applied pi calculus [4]. For instance, the output of a message M on channel N is represented by $\mathsf{out}(N, M); P$ while $\mathsf{in}(N, x : T); P$ represents an input on channel N, stored in variable x. Note that in both cases, N must have the type channel. Process $P \mid Q$ models the parallel composition of P and Q, while $!P$ represents P replicated an arbitrary number of times. $\mathsf{new}\ \ a : T; P$ generates a fresh name of type T and behaves like P. The conditional process $\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$ executes P if M is the boolean true and executes Q otherwise. The process $\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ evaluates D, stores it in x and then behaves like P unless the evaluation fails, in which case it behaves like Q. The process $\mathsf{event}(M); P$ is used to specify security properties: the process emits an *event* (not observable by an attacker) to reflect that it reaches some specific state, with some values, stored in M. Finally, ProVerif supports user defined tables declared by their name and the types of their elements, i.e. table $\mathit{tbl}(T_1, \ldots, T_n)$. The process insert $\mathit{tbl}(M_1, \ldots, M_n); P$ corresponds to the insertion in the table tbl of the entry $(M_1, \ldots, M_n)$. The process get $\mathit{tbl}(x_1 : T_1, \ldots, x_n : T_n)$ suchthat D in P else Q looks for an entry $(M_1, \ldots, M_n)$ in the table tbl such that $D\sigma$ evaluates to true with $\sigma = \{M_i/x_i\}_{i=1}^n$. If such an entry exists then $P\sigma$ is executed otherwise Q is executed. Note that the expression D is required to have the type bool.

The set of free names of a process P is denoted $\mathit{fn}(P)$. A *closed* process is a process with no free variables.

Example 2. The processes A and B defined in Section 2.1 are processes in the ProVerif syntax, where the type bitstring has been omitted. The composition of the two processes can be written

$$P_{\mathsf{enc}} = \mathsf{new}\ \ k; (\mathsf{new}\ k_1; \mathsf{new}\ k_2; A \mid B)$$

where k, k_1, k_2 are freshly generated. ▶

We may use pattern matching to ease readability. For example, $\mathsf{in}(c, (x : T, = M)); Q$ represents the process:

$$\begin{aligned} &\mathsf{in}(c, y : \mathsf{bitstring}); \\ &\mathsf{let}\ x : T = \mathsf{proj}_1(y)\ \mathsf{in} \\ &\mathsf{if}\ M = \mathsf{proj}_2(y)\ \mathsf{then}\ Q \end{aligned}$$

In the rest of the paper, we will assume that processes are written without pattern but we will use patterns to define our transformations for sake of readability. We may also omit the type bitstring when it is clear from the context.

3.2. Semantics

A *configuration* $E, \mathcal{S}, \mathcal{P}, \Phi$ is given by a multiset $\mathcal{P}$ of processes, representing the current state of the processes, a set $E = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}})$ representing respectively the public and private names used so far, a set $\mathcal{S}$ of elements of the form $(\mathit{tbl}, M_1, \ldots, M_n)$ representing the entries of user declared tables and finally a substitution Φ representing the knowledge of the attacker.

The semantics of processes is defined through a reduction relation $\to$ between configuration, defined as expected. For example, the rule corresponding to the reception of a message is defined as follows.

$$(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{in}(N, x : T); P\}\!\}, \Phi \to \\ (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{S}, \mathcal{P} \cup \{\!\{P\{M/x\}\}\!\}, \Phi$$

if there exist D_1, D_2 such that $\mathit{fv}(D_1, D_2) \subseteq \mathit{dom}(\Phi)$, $\mathit{fn}(D_1, D_2) \subseteq \mathcal{N}_{\text{pub}}$, $D_1\Phi \Downarrow N$, $D_2\Phi \Downarrow M$ and M is of type T. Intuitively, an attacker may inject any deducible message on a deducible channel.

A *trace* is a sequence of reductions between configurations $E_0, \mathcal{S}_0, \mathcal{P}_0, \Phi_0 \to \cdots \to E_n, \mathcal{S}_n, \mathcal{P}_n, \Phi_n$.

For the rest of this paper, we will say that a E, P is a *valid initial configuration* if P is a well-typed closed process and $E = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}})$ is a pair of sets of names such that $\mathcal{N}_{\text{pub}} \cap \mathcal{N}_{\text{priv}} = \emptyset$ and $\mathit{fn}(P) \subseteq \mathcal{N}_{\text{pub}}$. For sake of readability, we will write E, P instead of $E, \emptyset, \{\!\{P\}\!\}, \emptyset$. Moreover, we may write $a \in E$ instead of $a \in \mathcal{N}_{\mathrm{pub}} \cup \mathcal{N}_{\mathrm{priv}}$.

3.3. Security properties

ProVerif is able to verify three different kinds of security properties, namely secrecy properties, correspondence properties, and equivalence properties. In this paper, we only focus on secrecy and correspondence properties (equivalence properties is left as future work). We provide their formal definitions in this section.

Definition 1. *Let (E, P) be a valid initial configuration. Let M be a ground term such that $fn(M) \subseteq E$.*

We say that (E, P) preserves the secrecy of M iff for all traces $E, P \to^* (\mathcal{N}'_{\mathrm{pub}}, \mathcal{N}'_{\mathrm{priv}}), \mathcal{S}', \mathcal{P}', \Phi'$, for all expressions D, if $fn(D) \subseteq \mathcal{N}'_{\mathrm{pub}}$ and $fv(D) \subseteq dom(\Phi')$ then $D\Phi' \not\Downarrow M$.

Intuitively, the secrecy of M is preserved if the attacker is not able to deduce M for any trace of the process P.

Correspondence properties are very useful to express authentication properties such as "if Alice reaches some state (e.g. finishes her session) then Bob must have engaged a conversation with her". Such authentication queries are typically expressed through events, e.g. $\mathsf{event}(A) \rightsquigarrow \mathsf{event}(B)$ which requires that for all traces tr of the process, if tr has executed the event A then tr has also executed B. To define queries formally, we consider *facts* whose syntax is given by the following grammar:

F ::=	fact
$\mathsf{attacker}(M)$	the attacker knows M
$\mathsf{event}(ev(M_1, \dots, M_n))$	the event is executed
$M = N$	equality
$M \neq N$	disequality

A correspondence query is a formula of the form.

$$F \rightsquigarrow \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} F_{i,j}$$

where F is either an attacker or an event fact and the $F_{i,j}$s are either event, equality, or disequality facts.

We say that a trace $\mathsf{tr} = E, \mathcal{S}, \mathcal{P}, \Phi \to^* (\mathcal{N}'_{\mathrm{pub}}, \mathcal{N}'_{\mathrm{priv}}), \mathcal{S}', \mathcal{P}', \Phi'$ *executes* a ground fact F if

- either $F = \mathsf{attacker}(M)$ and there exists D such that $fv(D) \subseteq dom(\Phi')$, $fn(D) \subseteq \mathcal{N}'_{\mathrm{pub}}$ and $D\Phi' \Downarrow M$;
- or $F = \mathsf{event}(ev(M_1, \dots, M_n))$ and tr contains a reduction $E'', \mathcal{S}'', \mathcal{P}'' \cup \{\mathsf{event}(ev(M_1, \dots, M_n)); P\}, \Phi'' \to E'', \mathcal{S}'', \mathcal{P}'' \cup \{P\}, \Phi''$;
- or $F = (M = N)$ (resp $F = (M \neq N)$) and $M = N$ (resp. $M \neq N$).

Satisfiability of a correspondence query can now be formally defined.

Definition 2. *Let (E, P) be a valid initial configuration.*

A correspondence query $F \rightsquigarrow \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} F_{i,j}$ holds for (E, P) iff for all term substitutions σ closing for F, for all traces $\mathsf{tr} = E, P \to^* E', \mathcal{S}', \mathcal{P}', \Phi'$, if tr executes $F\sigma$ then there exist $i \in \{1, \dots, n\}$ and a term substitution σ_i such that $F\sigma = F\sigma_i$ and for all $j \in \{1, \dots, m_i\}$, σ_i is closing for $F_{i,j}$ and tr executes $F_{i,j}\sigma_j$.

Note that in the formula $F \rightsquigarrow \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} F_{i,j}$, the variables of $fv(F)$ are implicitly quantified universally while the other ones are implicitly quantified existentially. Given a correspondence query $F \rightsquigarrow \phi$ and a configuration (E, P), we write $E, P \models F \rightsquigarrow \phi$ if $F \rightsquigarrow \phi$ holds for (E, P).

Example 3. Continuing Example 2, the secrecy of s is preserved if $((\emptyset, \{s\}), P_{\mathsf{enc}})$ preserves the secrecy of s. The fact that s is a private name models the fact that s is initially unknown to the attacker.

Alternatively, we may require that the key received by the process B is one of the keys sent by A. This can be expressed by annotating A and B by the following events:

$$\begin{array}{rl} A' = & \mathsf{out}(c, \mathsf{enc}(s, (k_1, k_2))); \\ & \mathsf{event}(\mathsf{begin}(k_1)); \mathsf{event}(\mathsf{begin}(k_2)); \\ & \mathsf{out}(c, \mathsf{enc}(k_1, k)); \\ & \mathsf{out}(c, \mathsf{enc}(k_2, k)); 0 \\ & \\ B' = & \mathsf{in}(c, x); \\ & \mathsf{let}\ y = \mathsf{dec}(x, k)\ \mathsf{in} \\ & \mathsf{out}(c, y) \\ & \mathsf{event}(\mathsf{end}(y)); 0 \end{array}$$

and requiring the following correspondence property:

$$\mathsf{event}(\mathsf{end}(z)) \rightsquigarrow \mathsf{event}(\mathsf{begin}(z)). \qquad \blacktriangleright$$

Remark 1. In a correspondence query, a fact $\mathsf{attacker}(M)$ represents that the attacker can deduce M. Therefore the secrecy of M can actually be modeled by the correspondence property $\mathsf{attacker}(M) \rightsquigarrow \mathsf{false}$. ▶

Thanks to the previous remark, it is sufficient to consider correspondence queries in the rest of the paper.

4. ProVerif with global states

ProVerif is a very efficient tool that has been successfully used to analyse many protocols. However, as explained in Section 2, in case of stateful protocols, the number of false attacks raises dramatically. Here, we formalize the transformations sketched in Section 2 and prove their soundness.

In this section, we introduce a new type stamp, used to model fresh nonces used as stamps. We also introduce several function symbols for events, implicitly assuming that these function symbols do no already appear in the protocols under consideration.

4.1. Unique action

As explained in Section 2.1, ProVerif fails to prove protocols whose security relies on the fact that some rules may be used as most once. To avoid these false attacks, we introduce a new function symbol $\mathsf{UAction}(\mathsf{bitstring}, T) : \mathsf{bitstring}$. Then we annotate each input action $\mathsf{in}(c, m)$ with an event $\mathsf{UAction}(\mathsf{st}, m)$ that records that message m has been received at step st where st is a fresh name. This is formally defined as follows.

Definition 3. *Let P be a process. We denote by $[P]_{act}$ the process obtained from P by replacing any occurence of $\mathsf{in}(N, x : T); Q$ in P by*

$$\mathsf{new}\ \mathsf{st} : \mathsf{stamp}; \mathsf{in}(N, x : T); \mathsf{event}\ \mathsf{UAction}(\mathsf{st}, x); Q$$

where st *is a fresh name.*

Example 4. Consider B and B_{act} as defined in Section 2. We have $[B]_{act} = B_{\mathsf{act}}$. Then continuing Example 2, $[P_{\mathsf{enc}}]_{act} = \mathsf{new}\ \ k; (\mathsf{new}\ k_1; \mathsf{new}\ k_2; A \mid B_{\mathsf{act}})$. ▶

This transformation does not modify the behavior of the protocol since events do not interfere with the process.

Lemma 1. *Let (E, P) be a valid configuration. For all correspondence queries $F \rightsquigarrow \phi$,*

$$E, P \models F \rightsquigarrow \phi \quad \textit{iff} \quad E, [P]_{act} \models F \rightsquigarrow \phi$$

The freshness of the stamps guarantees the unicity of each event (for a given stamp).

Lemma 2. *Let (E, P) be a valid initial configuration. For all names* st*, for all ground terms M_1, M_2, for all traces* $\mathsf{tr} = E, [P]_{act} \rightarrow^* E', \mathcal{S}', \mathcal{P}', \Phi'$*, if* tr *executes* $\mathsf{event}(\mathsf{UAction}(\mathsf{st}, M_1))$ *and* $\mathsf{event}(\mathsf{UAction}(\mathsf{st}, M_2))$ *then* $M_1 = M_2$.

Let $\phi_{\mathsf{act}} = \mathsf{event}(\mathsf{UAction}(x_{st}, y)) \wedge \mathsf{event}(\mathsf{UAction}(x_{st}, z)) \wedge y \neq z$. It is sound to query $\phi \vee \phi_{\mathsf{act}}$ instead of ϕ.

Theorem 1. *Let (E, P) be a valid initial configuration. Let $F \rightsquigarrow \phi$ be a correspondance query. We have*

$$E, P \models F \rightsquigarrow \phi \textit{ if and only if } E, [P]_{act} \models F \rightsquigarrow (\phi \vee \phi_{\mathsf{act}})$$

Example 5. Continuing Examples 3 and 4, to show that $((\emptyset, \{s\}), P_{\mathsf{enc}})$ preserves the secrecy of s, it is sufficient to check whether $((\emptyset, \{s\}), [P_{\mathsf{enc}}]_{act}) \models \mathsf{attacker}(s) \rightsquigarrow \phi_{\mathsf{act}}$, thanks to Theorem 1. The most interesting part of this transformation is that ProVerif can indeed check the latter property, hence automatically proving that P_{enc} preserves the secrecy of s. ▶

The case of private channels, presented in Section 2.2 is handled similarly and formally stated in our technical report [14].

4.2. Cells

As seen in Section 2.3, secure hardwares typically have global states: a key is associated with a (mutable) status, a TPM stores keys and locks values in its registers. There is no construction in the applied-pi calculus to denote such global states (neither in most security protocols models). Instead, the most common way is to encode the storage of keys using private channels. Therefore, our first task is to detect when a private channel is used as a cell.

We say that a channel d is a *cell* w.r.t. a valid configuration $((\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}), P)$ if *(i)* $d \in \mathcal{N}_{\mathrm{priv}}$ or d is bound (once) in P; *(ii)* d only occurs as first argument of input or output in P; *(iii)* any output (unlock operation) on d is preceded by an input (lock operation) on d, possibly after arbitrary many other actions that do not involve d; there might be at most one exception (one output on d without an input on d, with no replication), corresponding to the initialization operation. A more formal definition is provided in our technical report [14].

Example 6. We consider the protocol of a simplified security device, as described in [5]. The configurable hardware device generates a public key $\mathsf{pk}(k)$. Alice encrypts a pair of secret (s_l, s_r). Bob can either configure the hardware to "left" or "right". Once the hardware is configured, it will always decrypt the left or right part of a pair, according to its setting. The device cannot be reconfigured.

To model this protocol, we consider a private name k and a private channel d. The process Conf models the setup of the device:

$$\begin{array}{l}
!\ \mathsf{in}(c, x); \mathsf{in}(d, y); \\
\quad \mathsf{let}\ t : \mathsf{bool} = (y = \mathsf{init}\ \&\&\ (x = \mathsf{left} \mid\mid x = \mathsf{right}))\ \mathsf{in} \\
\quad \mathsf{if}\ t\ \mathsf{then}\ \mathsf{out}(d, x)\ \mathsf{else}\ \mathsf{out}(d, y)
\end{array}$$

The process $\mathsf{Decrypt}$ models how the device decrypts pairs of secret depending on its configuration:

$$\begin{array}{l}
!\ \mathsf{in}(c, x); \\
\quad \mathsf{let}\ (x_l, x_r) = \mathsf{adec}(x, k)\ \mathsf{in} \\
\quad \mathsf{in}(d, y); \\
\quad \mathsf{if}\ y = \mathsf{left}\ \mathsf{then}\ \mathsf{out}(c, x_l); \mathsf{out}(d, y) \\
\quad \mathsf{else}\ \mathsf{if}\ y = \mathsf{right}\ \mathsf{then}\ \mathsf{out}(c, x_r); \mathsf{out}(d, y) \\
\quad \mathsf{else}\ \mathsf{out}(d, y)
\end{array}$$

The complete process P_{cell} can then modeled as follows:

$$\begin{array}{ll}
\mathsf{out}(c, \mathsf{pk}(k)) \mid & \\
\mathsf{out}(d, \mathsf{init}) \mid & \text{cell initialization} \\
\mathsf{Conf} \mid \mathsf{Decrypt} \mid & \\
\mathsf{out}(c, \mathsf{aenc}((s_l, s_r), \mathsf{pk}(k))) & \text{Alice's role}
\end{array}$$

Then d is a cell w.r.t. (E, P_{cell}) where $E = (\{c, \mathsf{left}, \mathsf{right}, \mathsf{init}\}, \{k, d, s_l, s_r\})$. The goal is to prove that the attacker cannot obtain both s_l and s_r, which is expressed by the query: $\mathsf{query\ attacker}((s_l, s_r))$.

Due to its overapproximations, ProVerif again fails to prove the query for P_{cell}. ▶

Our key idea is to link the values stored in cells, using stamps and events $\mathsf{VLink}(\mathsf{stamp}, \mathsf{stamp})$ and $\mathsf{VCell}(\mathsf{stamp}, \mathsf{bitstring})$, as illustrated in Figure 1.

Definition 4. *Let (E, P) be a valid configuration and d be a cell in (E, P). We denote by $[P]^d_{cell}$ the process obtained from P by replacing any subprocess $P' = \mathsf{in}(d, x : T); C[\mathsf{out}(d, M_1); Q_1, \ldots, \mathsf{out}(d, M_n); Q_n]$ (where C does not contain inputs nor outputs on d) by*

$$\begin{array}{l}
\mathsf{in}(d, (x_{st} : \mathsf{stamp}, x : T)); \\
\mathsf{event}\ \mathsf{VCell}_T(x_{st}, x); \\
C[Q'_1, \ldots, Q'_n]
\end{array}$$

where Q'_i is defined as follows:

- *if $M_i = x$ then $Q'_i = \mathsf{out}(d, (x_{st}, x)); Q_i$, this corresponds to the case where the value of the cell does not change so we do not need to annotate this action;*
- *otherwise*

$$\begin{aligned} Q'_i = {} & \mathsf{new}\ st : \mathsf{stamp}; \\ & \mathsf{event}\ \mathsf{VCell}_T(st, M_i); \\ & \mathsf{event}\ \mathsf{VLink}(x_{st}, st); \\ & \mathsf{out}(d, (\mathsf{st}, M_i)); Q_i \end{aligned}$$

Moreover, if P contains a subprocess $\mathsf{out}(d, M); Q$ *that is not preceeded by an input on d (initialization case), then it is replaced by* $\mathsf{new}\ st : \mathsf{stamp}; \mathsf{out}(d, (st, M)); Q$.

Example 7. Continuing Example 6, process $[P_{\mathsf{cell}}]^d_{cell}$ is:

$$\begin{array}{ll} \mathsf{out}(c, \mathsf{pk}(k)) \mid & \\ \mathsf{new}\ st_0 : \mathsf{stamp}; \mathsf{out}(d, (st_0, \mathsf{init})) \mid & \text{cell initialization} \\ \mathsf{Conf}' \mid \mathsf{Decrypt}' \mid & \\ \mathsf{out}(c, \mathsf{aenc}((s_l, s_r), \mathsf{pk}(k))) & \text{Alice's role} \end{array}$$

where the process Conf' is defined as follows:

$$\begin{array}{l} !\ \mathsf{in}(c, x); \mathsf{in}(d, (x_{st} : \mathsf{stamp}, y)); \\ \quad \mathsf{event}\ \mathsf{VCell}(x_{st}, y); \\ \quad \mathsf{let}\ t : \mathsf{bool} = (y = \mathsf{init}\ \&\&\ (x = \mathsf{left} \mid\mid x = \mathsf{right}))\ \mathsf{in} \\ \quad \mathsf{if}\ t\ \mathsf{then} \\ \qquad \mathsf{new}\ \mathsf{st} : \mathsf{stamp}; \mathsf{event}\ \mathsf{VLink}(x_{st}, \mathsf{st}); \mathsf{out}(d, (\mathsf{st}, x)) \\ \quad \mathsf{else}\ \mathsf{out}(d, (x_{st}, y)) \end{array}$$

Note that a new stamp is added only in the *then* branch of the condition. Finally the process $\mathsf{Decrypt}'$ is defined as follows:

$$\begin{array}{l} !\ \mathsf{in}(c, x); \\ \quad \mathsf{let}\ (x_l, x_r) = \mathsf{adec}(x, k)\ \mathsf{in} \\ \quad \mathsf{in}(d, (x_{st} : \mathsf{stamp}, y)); \\ \quad \mathsf{event}\ \mathsf{VCell}(x_{st}, y); \\ \quad \mathsf{if}\ y = \mathsf{left}\ \mathsf{then}\ \mathsf{out}(c, x_l); \mathsf{out}(d, (x_{st}, y)) \\ \quad \mathsf{else\ if}\ y = \mathsf{right}\ \mathsf{then}\ \mathsf{out}(c, x_r); \mathsf{out}(d, (x_{st}, y)) \\ \quad \mathsf{else}\ \mathsf{out}(d, (x_{st}, y)) \end{array}$$ ▶

Following the techniques of the previous transformations, we can show that it is safe to query $\phi \vee \phi_{\mathsf{cell}}$ on $[P]^d_{cell}$ instead of ϕ on P, where ϕ_{cell} has been defined in Section 2.3.

Theorem 2. *Let (E, P) be a valid configuration. Let d be a cell in (E, P). Let $F \rightsquigarrow \phi$ be a correspondence query. We have:*

$$E, P \models F \rightsquigarrow \phi \text{ iff } E, [P]^d_{cell} \models F \rightsquigarrow (\phi \vee \phi_{cell})$$

Example 8. Continuing Example 7, ProVerif can prove that $(E, [P_{\mathsf{cell}}]^d_{cell}) \models \mathsf{attacker}((s_l, s_r)) \rightsquigarrow \phi_{com}$. Thanks to Theorem 2, we deduce $(E, P_{\mathsf{cell}}) \models \mathsf{attacker}((s_l, s_r))$. ▶

5. Natural numbers

Our second and main contribution is to enrich ProVerif with natural numbers together with equalities and inequalities, as well as a limited addition (no addition of two variables). We adapt ProVerif procedure to cope with inequalities, using the simple polynomial time algorithm from Pratt [31] that converts inequalities between naturals into the existence of a cycle in a weighted graph.

5.1. Syntax for natural numbers

We consider a new built-in type nat, a public constant zero of type nat, a public constructor $\mathsf{succ}(\mathsf{nat}) : \mathsf{nat}$ and a public destructor $\mathsf{prev}(\mathsf{nat}) : \mathsf{nat}$ whose behaviour is defined by the rewrite rule $\mathsf{prev}(\mathsf{succ}(i)) \to i$. We write $f^n(t)$ to represent n applications of the function f to t. Therefore, a natural n is represented by the term $\mathsf{succ}^n(\mathsf{zero})$, an addition $x + n$ (with x a variable of type nat) is represented by the term $\mathsf{succ}^n(x)$. To ease reading, we may write $x+n$ instead of $\mathsf{succ}^n(x)$. Similarly to the case of projection of pairing in ProVerif, the subtraction $x = y-n$ is implicitly represented by the construction $\mathsf{let}\ x + n = y\ \mathsf{in} \cdots$.

To ensure that a term of type nat is always of the form $\mathsf{succ}^n(\mathsf{zero})$ or $\mathsf{succ}^n(x)$, we assume that names cannot be declared with the type nat and that constructor function symbols cannot be declared with nat as output type. This ensures that protocols can only create terms of the form $\mathsf{succ}^n(\mathsf{zero})$. On the other hand, terms forged by the attacker are also of the expected form since we consider a typed attacker (only w.r.t. the type nat). In our examples, the message format enforces this condition anyway. Note however that a function may take a nat as input and therefore natural numbers may be included in messages.

The equality between natural numbers is the equality between terms of type nat. To define inequalities, we introduce predicates:

$$\begin{array}{lll} F ::= & & \text{fact} \\ & \dots & \\ & p(M_1, \dots, M_n) & \text{predicate } p \end{array}$$

We define two predicates for strict inequality and inequality respectively:

$$\mathsf{pred\ less}(\mathsf{nat}, \mathsf{nat}) \qquad \mathsf{pred\ lesseq}(\mathsf{nat}, \mathsf{nat})$$

Their semantics is defined on closed terms of type nat as expected: $\mathsf{less}(\mathsf{succ}^n(\mathsf{zero}), \mathsf{succ}^m(\mathsf{zero})) = \mathsf{true}$ iff $n < m$, and $\mathsf{lesseq}(\mathsf{succ}^n(\mathsf{zero}), \mathsf{succ}^m(\mathsf{zero})) = \mathsf{true}$ iff $n \le m$. We may write $N < M$ (resp $N \le M$) instead of $\mathsf{less}(N, M)$ (resp $\mathsf{lesseq}(N, M)$).

5.2. Discussion

Instead of defining our own (interpreted) predicates, we could have relied on an advanced modeling feature implemented in ProVerif: predicates defined by Horn clauses. Since ProVerif's internal algorithm already translates a protocol into Horn clauses, these predicates are a natural extension to ProVerif calculus. For example, the predicate lesseq could be modeled by the following Horn clauses:

$$\begin{array}{l} \to \mathsf{lesseq}(\mathsf{zero}, x) \\ \to \mathsf{lesseq}(x, \mathsf{succ}(x)) \\ \mathsf{lesseq}(x, y)\ \&\&\ \mathsf{lesseq}(y, z) \to \mathsf{lesseq}(x, y) \end{array}$$

However, we cannot express the fact $\mathsf{lesseq}(x, y)$ and $\mathsf{lesseq}(\mathsf{succ}(y), x)$ cannot hold at the same time. Moreover, we can neither declare that if $\mathsf{lesseq}(x, y)$ and $\mathsf{lesseq}(y, x)$ both hold then $x = y$.

Therefore, it is much more powerful to consider interpreted predicates for less and lesseq.

5.3. Extending ProVerif to natural numbers

To soundly extend ProVerif to natural numbers, we proceed in two steps:

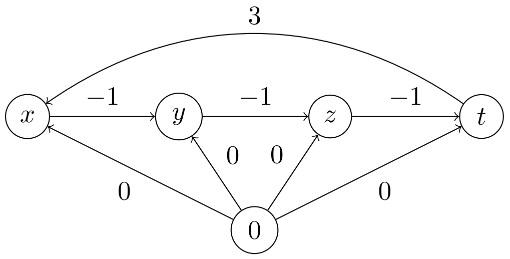
1) First, we implement the algorithm of Pratt [31] for checking satisfiability of inequalities. Not only we can decide (in polynomial time) whether a set of inequalities can be satisfied, but we also detect *forced equalities*. For example, the set of inequalities $\{x \leq y{+}1, y < x, z \leq 3\}$ has solutions and any solution is such that $x = y + 1$.
2) Second, we refine ProVerif's procedure in order to better detect when the queried properties are satisfied.

Other algorithms (*e.g.* UTVPI [34]) exist in the literature for solving more complex inequalities like $x \leq y + z + k$. However, such inequalities would require to introduce $+$ as a true operator (instead of succ) and would require much more work to obtain a sound and reasonably terminating saturation procedure.

5.3.1. Solving inequalities.

Recall that terms of type nat are necessarily either n or $x{+}n$ with $n \in \mathbb{N}$ and x a variable of type nat. Following Pratt's algorithm [31], we associate to each conjunction ϕ of inequalities between terms of type nat a weighted directed graph where an arrow of weight k between two nodes x and y represents that $x \leq y + k$.

Example 9. Consider the conjunction $\phi = x < y \wedge y < z \wedge z < t \wedge t \leq x + 3$. We obtain the following graph.



▶

There is a solution if and only if there is no cycle of negative weight. Moreover, any cycle of weight 0 indicates forced equalities. This yields a simple (polynomial time) procedure for solving inequalities.

Proposition 1. *There is a polynomial time algorithm* checkeq *that given a conjunction ϕ of inequalities between terms of type* nat *returns:*

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

This proposition follows the intuition of [31] and is formally proven in our technical report [14].

5.3.2. Refined ProVerif procedure.

We need to extend ProVerif procedure in order to cope with natural numbers. ProVerif proceeds in two steps. First, given an initial configuration (E, P), and a fact F, it internally translates P into a sound set S of Horn clauses: if F can be executed by P then there a derivation of F from S. ProVerif then saturates S until it reaches a fixed point, the set $\mathsf{solve}_{E,P}(F)$. We only need to know that if F can be executed, then one instance of a clause of $\mathsf{solve}_{E,P}(F)$ occured in the derivation of F.

Proposition 2 ([10]). *Let (E, P) be a valid initial configuration. Let F be an attacker or an event fact. For any trace $\mathsf{tr} = E, P \rightarrow^* E', \mathcal{S}', \mathcal{P}', \Phi'$, for any substitution σ, if* tr *executes $F\sigma$ then there exist a clause $H \Rightarrow C \in \mathsf{solve}_{E,P}(F)$ and a substitution σ' such that $F\sigma = C\sigma'$ and for any fact F' in $H\sigma$,* tr *executes F'.*

Consider now a query $F \rightsquigarrow \phi$. Given a clause $H \Rightarrow C \in \mathsf{solve}_{E,P}(F)$, ProVerif tries to guarantee that, in case C may produce $F\sigma$ for some σ then $\phi\sigma$ is entailed by H.

We further refine this procedure in Algorithm 1. Correspondence queries are now of the form $F \rightsquigarrow \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} F_{i,j}$ where the $F_{i,j}$ are either equality facts (EF), disequality facts (DF), inequality facts (IF), or other facts (OF), that is, events or predicates that are not less nor lesseq. A subquery $\bigwedge_{j=1}^{m_i} F_{i,j}$ is *specialized* if the variables in IF and DF facts occur in F or in OF facts.

For the sake of the presentation, we assume here that ϕ is a specialized sub-query of the form $\phi_{\mathsf{OF}} \wedge \phi_{\mathsf{EF}} \wedge \phi_{\mathsf{DF}} \wedge \phi_{\mathsf{IF}}$ where ϕ_{OF} (resp. ϕ_{EF}, ϕ_{DF}, ϕ_{IF}) is a conjunction of OF facts (resp. EF, DF, IF facts). Our procedure `verif(`$H \Rightarrow C$`,`$F \rightsquigarrow \phi$`)` works intuitively as follows.

1) First, we simplify clauses $H \Rightarrow C \in \mathsf{solve}_{E,P}(F)$ by examining the set R of IF facts of H and applying checkeq.
   a) If R has no solution, then H is always false and the clause $H \Rightarrow C$ can be removed.
   b) Else our algorithm checkeq(R) returns a set of equalities E that must be satisfied by any solution of R. Thus we consider the simplified clause $H\theta \Rightarrow C\theta$ where the equalities of E are satisfied: $\theta = mgu(E)$.
2) Then, for any simplified clause $H \Rightarrow C$, we match F with C, that is, we try to write $F\sigma = C$ for some σ. If $\phi\sigma$ is not already false, that is $\sigma \models \phi_{\mathsf{EF}} \wedge \phi_{\mathsf{DF}}$, $\phi_{\mathsf{OF}}\sigma \subseteq H$, and $\phi_{\mathsf{IF}}\sigma$ has a solution, then we try to show that $\phi\sigma$ is implied by H. In case $\phi\sigma$ is not immediately implied by H as for the ProVerif procedure, we further refine the procedure by characterizing what can happen if a DF or IF fact is not satisfied.
   a) For any DF fact $M \neq N \in \phi_{\mathsf{DF}}$, we check that in case the disequality is not satisfied, that is, for any σ' such that $M\sigma\sigma' = N\sigma\sigma'$, then the clause $H\sigma' \Rightarrow C\sigma'$ already entails the query, by calling `verif(`$H\sigma' \Rightarrow C\sigma'$`,`$F \rightsquigarrow \phi$`)`.
   b) For any IF fact $M < N$, we similarly check that in case the inequality is not satisfied, that is, $N\sigma \leq M\sigma$, then the clause $H \Rightarrow C$ already entails the query, by calling `verif(`$H \wedge N\sigma \leq M\sigma \Rightarrow C$`,`$F \rightsquigarrow \phi$`)`; and similarly for an IF fact $M \leq N$.

Note that Step 2.a is actually independent from our in-

Algorithm 1: Extended verification algorithm.

```
Function verif($H \Rightarrow C, F \rightsquigarrow \phi$)
    Data: A Horn clause $H \Rightarrow C$ and a query $F \rightsquigarrow \phi$
    Result: A boolean

    try
        $H' \Rightarrow C'$ := simplify($H \Rightarrow C$);
        if $\exists$ $\sigma$ and $\phi_{\mathsf{OF}} \wedge \phi_{\mathsf{EF}} \wedge \phi_{\mathsf{DF}} \wedge \phi_{\mathsf{IF}}$ in $F \rightsquigarrow \phi$ such that $F\sigma = C'$, $\sigma \models \phi_{\mathsf{EF}} \wedge \phi_{\mathsf{DF}}$, $\phi_{\mathsf{OF}}\sigma \subseteq H'$ and $\phi_{\mathsf{IF}}\sigma$ has a solution.
         then

            foreach $M \neq N \in \phi_{\mathsf{DF}}$ do
                if $\sigma' = mgu(M\sigma, N\sigma)$ and verif($H'\sigma' \Rightarrow C'\sigma'$,$F \rightsquigarrow \phi$) = false then return false
            foreach $M < N \in \phi_{\mathsf{IF}}$ do
                if verif($H' \wedge N\sigma \leq M\sigma \Rightarrow C'$, $F \rightsquigarrow \phi$) = false then return false
            foreach $M \leq N \in \phi_{\mathsf{IF}}$ do
                if verif($H' \wedge N\sigma < M\sigma \Rightarrow C'$,$F \rightsquigarrow \phi$) = false then return false
            return true
        else
            return verifPV($H' \Rightarrow C'$,$F \rightsquigarrow \phi$)      /* We use the original verification procedure of
             ProVerif when no specialized query satisfies the conditions. */
    with  exception False_hypothesis
        return true
Function simplify($H \Rightarrow C$)
    Data: A Horn clause $H \Rightarrow C$
    Result: Two set of disequality or natural number inequalty facts respectively

    $R := \{M \; op \; N \in H \mid op \in \{<; \leq\}\}$              /* select the inequality facts in the clause */
    if checkeq($R$) $\neq \bot$ then
        $\sigma$ := checkeq($R$);                                        /* $\sigma$ can be the identity */
        if for all $M \neq N \in H$, $\sigma \models M \neq N$ then  return $H\sigma \Rightarrow C\sigma$  else  raise False_hypothesis
        /* The condition ensures that the disequality in the hypotheses of the clause are
           not trivially false                                                                  */
    else
        raise False_hypothesis
```

troduction of natural numbers and is of independent interest. With its current procedure, ProVerif typically fails to prove a query of the form $E(x) \rightsquigarrow x = a \vee x \neq a$ (where E is some event). This is due to the fact that ProVerif actually tries to prove $E(x) \rightsquigarrow x = a$ or $E(x) \rightsquigarrow x \neq a$. This is no longer the case with our refined procedure.

Of course, we can show that our new algorithm preserves the soundness of ProVerif.

Theorem 3. *Let (E, P) be a valid initial configuration. Let $F \rightsquigarrow \phi$ be a correspondence query. If for all $H \Rightarrow C \in \mathsf{solve}_{E,P}(F)$,* verif($H \Rightarrow C, F \rightsquigarrow \phi$) = true *then $E, P \models F \rightsquigarrow \phi$.*

Thanks to Theorem 3, ProVerif can now soundly analyse protocols with natural numbers and comparisons.

6. Transformations with natural numbers

Natural numbers are particularly useful when modeling counters. Counters are used e.g. in security devices (Yubikey [35], CANauth [25]) or payment protocols [15]. There is no explicit way for expressing counters in ProVerif. The most natural encoding is to use a private channel that sends and receives the value of the counter,

Similarly to Section 4, we detect when a channel is used as a counter and we show how to annotate a process with events in order to help ProVerif when protocols use counters.

Formally, we say that d *is a counter* w.r.t. a valid configuration (E, P) if d is a cell w.r.t. (E, P) and

- any subprocess $\mathsf{out}(d, M); Q$ of P is such that M is a term of type nat.
- for any subprocess $\mathsf{in}(d, x : T); C[\mathsf{out}(d, M_1); Q_1, \ldots, \mathsf{out}(d, M_n); Q_n]$ of P (with no input nor output on d in C), then $T = \mathsf{nat}$ and $M_j = x + n_j$ for some $n_j \in \mathbb{N}$.

Note that our definition enforces that a counter may only increase. Our definition excludes for example updates of counters using incoming messages. Actually, the key property needed for the soundess of our transformation is the monocity of each counter. We could therefore relax our syntactic condition in order to cover more cases.

Example 10. Consider a simple protocol where two agents A and B may access the same counter d. When A retrieves the value of the counter, she outputs a hash of it with a secret s and increments the counter. When B receives a message, he checks whether it corresponds to the hash of the current value of the counter and the secret s. If yes, B leaks the secret s. In both cases, B increments the counter. The secret is never leaked since B may only receive the secret hashed with old values of the counter.

A and B can be modeled by the following processes.

$$\begin{aligned}
A &= \mathsf{in}(d, i : \mathsf{nat}); \mathsf{out}(c, \mathsf{h}(i, s)); \mathsf{out}(d, i + 1)\\
B &= \mathsf{in}(d, i : \mathsf{nat}); \mathsf{in}(c, y);\\
&\quad\ \mathsf{if}\ y = \mathsf{h}(i, s)\ \mathsf{then}\\
&\quad\quad \mathsf{out}(c, s); \mathsf{out}(d, i + 1)\\
&\quad\ \mathsf{else}\ \mathsf{out}(d, i + 1)
\end{aligned}$$

The complete protocol is simply:

$$P = !\ A \mid !\ B \mid \mathsf{out}(d, 0) \mid !\ \mathsf{in}(d, i : \mathsf{nat}); \mathsf{out}(d, i)$$

The last part models that the counter is always available to both agents. Unsurprisingly, ProVerif cannot prove secrecy. It also fails even after applying our transformation on cells because ϕ_{cell} does not convey the information that a counter may never take twice the same value (this is false in general for a cell). ▶

We define a new event Counter(stamp, nat), used to record each time a counter is updated.

Definition 5. *Let (E, P) be a valid initial configuration. Let d be a counter in (E, P). We denote by $[P]^d_{count}$ the process P in which we replace any subprocess $\mathsf{in}(d, x : \mathsf{nat}); C[\mathsf{out}(d, x+i_1); Q_1, \dots, \mathsf{out}(d, x+i_n); Q_n]$ of P (with no input nor output on d in C), with $i_j \neq 0$, by the process*

$$\begin{aligned}
P_i = &\ \mathsf{new}\ \mathsf{st} : \mathsf{stamp};\ \mathsf{in}(d, x : \mathsf{nat});\\
&\ \mathsf{event}\ \mathsf{Counter}(\mathsf{st}, x);\\
&\ C[\mathsf{out}(d, x + i_1); Q_1, \dots, \mathsf{out}(d, x + i_n); Q_n]
\end{aligned}$$

Example 11. Continuing Example 10, we have $[P]^d_{count} = !\ [A]^d_{count} \mid !\ [B]^d_{count} \mid \mathsf{out}(d, 0) \mid !\ \mathsf{in}(d, i : \mathsf{nat}); \mathsf{out}(d, i)$ where:

$$\begin{aligned}
[A]^d_{count} = &\ \mathsf{new}\ st : \mathsf{stamp}; \mathsf{in}(d, i : \mathsf{nat});\\
&\ \mathsf{event}\ \mathsf{Counter}(st, i);\\
&\ \mathsf{out}(c, \mathsf{h}(i, s)); \mathsf{out}(d, i + 1)\\
\\
[B]^d_{count} = &\ \mathsf{new}\ st' : \mathsf{stamp}; \mathsf{in}(d, i : \mathsf{nat});\\
&\ \mathsf{event}\ \mathsf{Counter}(st, i);\\
&\ \mathsf{in}(c, y);\\
&\ \mathsf{if}\ y = \mathsf{h}(i, s)\ \mathsf{then}\\
&\quad \mathsf{out}(c, s); \mathsf{out}(d, i + 1)\\
&\ \mathsf{else}\ \mathsf{out}(d, i + 1)\quad ▶
\end{aligned}$$

Since a counter may never take twice the same value, we introduce the formula ϕ_{count} defined as follows:

$$\begin{aligned}
&(\mathsf{event}(\mathsf{Counter}(x, i)) \wedge \mathsf{event}(\mathsf{Counter}(x, j)) \wedge i \neq j)\\
\vee\ &(\mathsf{event}(\mathsf{Counter}(x, i)) \wedge \mathsf{event}(\mathsf{Counter}(y, i)) \wedge x \neq y)
\end{aligned}$$

Similarly to Section 4, it is safe to query $\phi \vee \phi_{\mathsf{count}}$ instead of ϕ.

Theorem 4. *Let (E, P) be a valid configuration, d a counter in (E, P), and $F \rightsquigarrow \phi$ a correspondence query. We have:*

$$E, P \models F \rightsquigarrow \phi \textit{ iff } E, [P]^d_{count} \models F \rightsquigarrow (\phi \vee \phi_{\mathsf{count}})$$

Natural numbers used as counters are also useful to express finer properties. For example, we may express that once an element has been added to a table at step i, it cannot be removed at later steps $j > i$. We illustrate the issue and our transformation on an example.

Example 12. Consider a very simple voting protocol which only aim is to ensure that a server S never registers two votes for the same voter. The server S stores the names of the voters who voted already in a table $VoterTbl$. Moreover, it locks the table to avoid concurrent accesses. Such a protocol can be modeled as follows.

$$\begin{aligned}
&\mathsf{in}(c, (x_a, x_v)); \quad \text{// } S \text{ receives agent's id and vote.}\\
&\mathsf{in}(d, x); \quad\quad\quad\ \ \text{// } S \text{ locks the table.}\\
&\mathsf{get}\ VoterTbl(= x_a)\ \mathsf{in}\\
&\quad \mathsf{out}(d, x)\\
&\mathsf{else}\\
&\quad \mathsf{insert}\ VoterTbl(x_a);\\
&\quad \mathsf{event}\ HasVoted(x_a, x_v); \quad \text{// The vote is counted.}\\
&\quad \mathsf{out}(d, x)
\end{aligned}$$

The main process is $P = !\ S \mid \mathsf{new}\ a; out(d, a) \mid !\ in(d, x); out(d, x)$ to initiate the lock mechanism.

We wish to prove that the server cannot record two votes from the same voter, which corresponds to the query $\phi = HasVoted(x, y) \wedge HasVoted(x, z) \rightsquigarrow y = z$.

Note that if we remove the lock mechanism (that is, removing all input/output on channel d), the protocol becomes insecure since the server may emit two events $HasVoted(a, v_1)$, $HasVoted(a, v_2)$ for the same voter a before effectively recording in the table that a has voted.

The security of the protocol relies on the fact that once an element is stored in a table, it cannot be removed. We annotate the protocol with events $\mathsf{InTbl}(i, t)$ and $\mathsf{NotInTbl}(i, t)$ which indicate that an element t is (resp. is not) in the table at step i.

$$\begin{aligned}
&\mathsf{in}(c, (x_a, x_v));\\
&\mathsf{in}(d, (i : \mathsf{nat}, x));\\
&\mathsf{get}\ VoterTbl(= x_a)\ \mathsf{in}\\
&\quad \mathsf{event}\ \mathsf{InTbl}(i, x_a); \mathsf{out}(d, (i + 1, x))\\
&\mathsf{else}\\
&\quad \mathsf{event}\ \mathsf{NotInTbl}(i, x_a);\\
&\quad \mathsf{event}\ \mathsf{InTbl}(i + 1, x_a);\\
&\quad \mathsf{insert}\ VoterTbl(x_a);\\
&\quad \mathsf{event}\ HasVoted(x_a, x_v);\\
&\quad \mathsf{out}(d, (i + 1, x))
\end{aligned}$$

Once an element is in the table, it cannot disappear. That is, the following formula is always false:

$$\phi_{\mathsf{table}} = \mathsf{InTbl}(i, t) \wedge \mathsf{NotInTbl}(j, t) \wedge i \leq j$$

Therefore it is safe to query $\phi \vee \phi_{\mathsf{table}}$ instead of ϕ. ▶

The full definition of our transformation, together with a more general property ϕ_{table} can be found in our technical report [14].

7. Implementation

We implemented our tranformations in a frontend GSVerif that takes as input a standard ProVerif file. The user should simply specify which channels should be considered for our transformation, by indicating the keyword `precise`. Note that this does not change the semantics of the corresponding channels. Then GSVerif tries to automatically detect whether precise channels can be seen as a cell, a counter, or a lock for a table, and applies the most precise available transformation. By default, it uses ϕ_{act} for a public channel and ϕ_{com} for a (strongly) private one. Then it remains to run ProVerif on the resulting file. Note that this transformation is always immediate (less than 1 ms). As described in Section 5, we consider the enhanced version of ProVerif for natural numbers and disjunctions.

7.1. Experiments

We conducted an extensive study of existing tools, based on process algebra, for security protocols with global states (SAPIC [27], StatVerif [5], SetPi [13]), on the available protocols of the literature, as well as our own illustrative examples. Given that SetPi fails on many simple examples, we did not model our two large, time-consuming, examples: mobile EMV [15] and Scytl's voting protocol [16]. We also tested ProVerif itself since it does prove some properties when they do not rely strongly on global states. We failed to use the recent tool AIF-ω [30]. Direct encoding of our protocols in AIF-ω unfortunately trigger false attacks. Each protocol requires manual adaptations to guide the tool. For example, for the one-dec protocol (Example 2), the authors of [30] kindly provided us with a file where the recommended adaptation for the general decryption oracle $\mathsf{enc}(x, k) \to x$ is to identify when the key k is used for encryption and manually derive a set of simpler rules. The resulting protocol can then be proved secure. We leave (manual) adaptations of the other protocols for future work.

All our experiments are reported in Figure 3. We ran the different tools on a 10-core Intel 3.1 GHz Xeon with 50Gb of RAM. We stopped each experiment after 24h.

Methodology. For all the examples, we started from the proposed model(s) in the literature, that we adapted to the other tools. For GSVerif, a direct translation into ProVerif syntax was sufficient, showing that our tool can accomodate various styles of modeling. The only exceptions are the illustrative (and invented) examples presented throughout the paper, as well as the TPM protocol. The original model of the TPM protocol [20] was written directly in Horn clauses, too far from the process algebra dialects of the considered tools. Thus we chose to re-write a fresh model for the TPM.

GSVerif combined with ProVerif can prove all the protocols in our benchmarks in a very efficient way. The two exceptions are Yubikey and CANauth for which we had to add intermediate properties to help ProVerif, as discussed in the next section. Similarly, when SAPIC or Tamarin fail to automatically prove security, it is possible to enter manually some lemmas or even enter the interactive mode. The corresponding proofs are indicated with the sign ■ in Figure 3 with a reference to the corresponding paper.

StatVerif can only handle a finite number of states, typically 1 or 2. Therefore, for most of our examples, we also consider a version of the protocol with only one or two cells. For example, two cells for the security device (Example 6) means exactly two devices. In some cases (e.g. mobile EMV [15]), even a single regular session of the protocol involves 4 different states. In that case, we consider the minimal possible number of states. Of course, we also run our experiments without limiting the number of agents (and therefore of states), in order to explore the other tools.

Given that Tamarin (with or without Sapic) is not always automatic, we first looked for existing security proofs in the literature. For protocols that already have a security proof (possibly with lemmas), we did not try to do better than the original authors of the proof. For protocols without existing security proofs, we ran Sapic only, as it is based on process algebra, which eases the translation w.r.t. the other considered tools. However, this does not mean that an automatic proof in Tamarin is not possible.

Since not all tools support natural numbers, we sometimes had to replace the comparison $j < i$ (the counter is "fresh") by $j = i + 1$ (the server only accepts if the counter has been incremented exactly by one). This corresponds to our simplified versions of Yubikey, EMV, and CANauth. In some cases, we even had to replace counters by nonces (indicated by ⚠✓).

Our experiments yield the first fully automatic proof of a security API [27] and a payment protocol [15]. Not only an automatic analysis discharge the user from any interaction with the tool but the modeling task, at least on our examples, was simple. For example, the original model and proof in Tamarin of mobile EMV [15] required about a couple of months of work while its translation in GSVerif lasted a couple of days.

7.2. Attacks

We discovered two attacks. First the Key Registration protocol, as described in [13], is actually flawed. This simple protocol aims at revoking public keys: the attacker should not learn a private key unless the corresponding public key has been revoked by the server. However, an attacker may fake the acknowledgment of the server and therefore trigger an agent to reveal her key while the server has not registered the revocation. This attack was not detected by the authors because SetPi actually assesses that the protocol is secure while it is not. The attack has been reported and acknowledged by the authors of [13].

Second, we found a flaw in the mobile payment protocol proposed in [15]. The attack relies on the fact that the protocol uses two hmac $\mathsf{hmac}(K_{\mathsf{pay}}, s)$ and $\mathsf{hmac}(K_{\mathsf{pay}}, (merchand, price, s))$ that can be confused. This attack was not detected by the authors because the message format in their model prevents the attack but a different format (pairs done left first instead of right first)

Protocol	Model's origin	# cells	ProVerif	StatVerif	SAPIC/Tamarin		Set-Pi	GSVerif
one-dec (Ex. 2 ‡)	invented	0	✗FA	✗FA	■ 2s		✗FA	✓< 1s
one-dec, table variant		0	✗FA	✗FA	✓7s		✗FA	✓< 1s
private-channel (Sec. 2.2)		0	✗FA	✗FA	✓2s		✗FA	✓< 1s
counter (Example 10)		∞	✗FA	—	⚠✓10s		—	✓< 1s
		2	✗FA	✗time	⚠✓24s		✗time	✓< 1s
voting (Example 12)		∞	✗FA	✗FA	✓3s		—	✓< 1s
		2	✗FA	✗FA	✓7s		✓< 1s	✓< 1s
TPM-enveloppe [20]	Horn clause [20] Tamarin [28]	∞	✗FA	—	✗memory	■ 1m50s [28]	—	✓< 1s
		2	✗FA	✗time			—	✓< 1s
TPM-bitlocker [20]		1	✓< 1s	✓< 1s	✗memory		—	✓< 1s
TPM-toy [20]		∞	✗FA	—	✗memory	■ 3s [28]	—	✓< 1s
		2	✗FA	✗time			—	✓< 1s
Key registration [13]	Set-Pi [13]	∞	✓< 1s	✓< 1s	✓11s	✓< 1s [28]	—	✓< 1s
		2	✓< 1s	✓< 1s	✓1m2s		✗bug	✓< 1s
Yubikey [35]	SAPIC [27] Set-Pi [13]	∞	✗FA	—	■ interactive [27], [28]		—	■< 1s
Yubikey simplified		∞	✗FA	—			—	✓< 1s
		2	✗FA	✗time			⚠✓< 1s	✓< 1s
Secure device [5]	StatVerif [5] SAPIC [27]	∞	✗FA	—	■ 24s [27]	■ < 1s [28]	—	✓< 1s
		2	✗FA	✓< 1s	■ 2m4s [27]		⚠✓< 1s	✓< 1s
PKCS#11 [27]	SAPIC [27]	∞	✓< 1s	✓< 1s	■ 23m13s [27]		—	✓< 1s
Security-API [27]	SAPIC [27]	∞	✗FA	—	■ 2m42s [27]		✗FA	✓< 1s
CANauth [25], [13]	Set-Pi [13]	∞	✗FA	—	✗memory		—	■< 1s
CANauth simplified		2	✗FA	✗time	✗memory		⚠✓< 1s	✓< 1s
Garay-Mackenzie [24]	StatVerif [5]	∞	✗FA	—	✓25s	✓< 1s [28]	—	✓< 1s
		1	✗FA	✓3s	✓51s		✗FA	✓< 1s
Mobile EMV [15]	Tamarin [15]	∞	✗FA	—				✓< 1s
		4	✗FA	✗time		■ 1m26s [15]		✓< 1s
Scytl Voting System [16]	ProVerif [9]	1	✗FA	✗FA	✓9s			✓10s

✓ Automatic proof ⚠✓Counters abstracted by nonces ■ manual proofs with lemmas or interactive mode

✗False attacks (FA), computation time >24h (time), memory used >50Gb (memory) — protocol out of scope

‡ with a fresh nonce to avoid trivial attacks when replicated

Figure 3. Experiments: protocols with global states

would enable the attack. Again, the attack has been reported and acknowledged by the authors of [15].

8. Beyond GSVerif

Our experiments (Figure 3) show that, in some cases, our frontend GSVerif fails to automatically prove security. Interestingly, we can still prove security by querying additional properties. This somehow adds some flavour of interactivity in ProVerif. The idea is very simple: instead of querying ϕ, it is always safe to query ψ and $\phi \vee \neg\psi$. This may be sufficient for ProVerif to conclude. Sometimes, we may need an induction. So we may simply query $\psi(0)$, as well as $\psi(n) \Rightarrow \psi(n+1)$. If both properties hold, it is safe to query $\phi \vee \exists n. \neg\psi(n)$ instead of ϕ.

We illustrate this approach with the Yubikey protocol [35]. Yubikey is a small simple device, designed to authenticate users with some web services. A user shall simply press a button to be authenticated. More specifically, a Yubikey device owns some public identifier pid, a secret id s_{id}, and a secret AES key k, shared with the server. Moreover, the Yubikey device also uses a counter tc. Every time the button is pressed, the device generates a one-time password based on k, s_{id}, the current value of the counter tc, as well as some random values $nonce$ and npr. The Yubikey authentication server checks whether the one-time password $\mathsf{enc}((s_{id}, tc, npr), k)$ contains a counter value tc that is strictly bigger than the previously received value, stored in a counter otc. If the checks succeed, the server grants access to the user.

This protocol can be modeled by the process P_{Yubi}:

$$\begin{aligned} P_{Yubi} = \; ! \; & \mathsf{new}\ k; \mathsf{new}\ pid; \mathsf{new}\ s_{id}; \mathsf{new}\ d_{usr}; \mathsf{new}\ d_{srv}; \\ & (\ \mathsf{out}(d_{srv}, (0, (s_{id}, k, 0))) \mid \mathsf{out}(d_{usr}, 1) \mid \\ & \quad \mathsf{out}(c, pid) \mid !P_{srv} \mid !P_{usr}\ ) \end{aligned}$$

where P_{serv} and P_{usr} are the processes of the server and user respectively. The process $\mathsf{out}(d_{srv}, (0, (s_{id}, k, 0)))$ models the initialization of the cell (or internal memory) of the server and the process $\mathsf{out}(d_{usr}, 1)$ represents the initialization of the counter of the user. Note that the content of the server's cell is a pair of a natural number and some tuple also containing a natural number. The latter represents the latest counter value seen by the server. The former is used to record each time that the server grants authentication.

The processes P_{usr} and P_{usr} are defined as follows.

$$\begin{aligned} P_{usr} = \; & \mathsf{in}(d_{usr}, tc : \mathsf{nat}); \\ & \mathsf{new}\ nonce; \mathsf{new}\ npr; \\ & \mathsf{event}\ \mathsf{YubiPress}(pid, s_{id}, k, tc); \\ & \mathsf{out}(c, (pid, nonce, \mathsf{enc}((s_{id}, tc, npr), k)); \\ & \mathsf{out}(d_{usr}, tc + 1) \end{aligned}$$

```
$P_{srv} = \mathsf{in}(c, (= pid, x_{nc}, y))$
          $\mathsf{in}(d_{srv}, (i : \mathsf{nat}, (= s_{id}, = k, otc : \mathsf{nat})));$
          $\mathsf{let}\ (= s_{id}, tc : \mathsf{nat}, npr) = \mathsf{dec}(y, k)\ \mathsf{in}$
            $\mathsf{if}\ otc < tc\ \mathsf{then}$
              $\mathsf{event\ Login}(pid, k, i + 1, tc);$
              $\mathsf{out}(d_{srv}, (i + 1, (s_{id}, k, tc)))$
            $\mathsf{else\ out}(d_{srv}, (i, (s_{id}, k, otc)))$
          $\mathsf{else\ out}(d_{srv}, (i, (s_{id}, k, otc)))$
```

The protocol should ensure that each successful login was triggered by a user (pressing the Yubikey button) and that no replay attacks are possible. These two properties are respectively expressed as follows.

$$\mathsf{Login}(pid, k, i, tc)) \rightsquigarrow \mathsf{YubiPress}(pid, s_{id}, k, tc)$$

$$\phi_{\mathsf{noreplay}} = \mathsf{Login}(pid, k, i, tc)) \wedge \mathsf{Login}(pid, k, j, tc)) \rightsquigarrow i = j$$

Whereas ProVerif can prove the first property without any help, it fails to prove the second property ϕ_{noreplay}.

So we introduce a stronger security property $\psi(i)$:

$$\begin{aligned}\mathsf{Login}(pid, k, i, tc)) \wedge \mathsf{Login}(pid, k, i', tc') \wedge i' \leq i\\ \rightsquigarrow (i = i' \wedge tc = tc') \vee tc' < tc\end{aligned}$$

This property can be proved by induction on i as follows.

$$\begin{aligned}&\mathsf{Login}(pid, k, i + 1, tc) \wedge \mathsf{Login}(pid, k, i', tc') \rightsquigarrow\\ &\quad i' > i + 1 \ \vee \ (i' = i + 1 \wedge tc = tc') \vee tc' < tc\\ &\quad \vee [j \leq i \wedge \mathsf{Login}(pid, k, j, y) \wedge \mathsf{Login}(pid, k, j', y')\\ &\qquad \wedge j' \leq j \wedge (j \neq j' \vee y \neq y') \wedge y \leq y']\end{aligned}$$

The two first lines of this property correspond to $\psi(i + 1)$ while the two last lines correspond to $\neg\psi(i)$.

Combined with our frontent GSVerif, ProVerif can automatically prove $\psi(0)$, $\psi(i) \Rightarrow \psi(i+1)$, and $\phi_{\mathsf{noreplay}} \vee \neg\psi(i)$. We easily conclude that ϕ_{noreplay} is guaranteed.
The CANauth protocol [25] is proved by manually adapting the transformations corresponding to ϕ_{cell} and ϕ_{com}.

9. Conclusion and discussion

We devise a simple, generic, and rather powerful approach that extends the popular tool ProVerif to global states. Maybe surprisingly, writing heavier queries actually helps ProVerif to conclude, thanks to its internal algorithm. We provide several sound transformations that cover private channels, cells, counters, and tables. Some of our transformations are quite specific (e.g. on cells). They will work only on protocols where the values of the cells increase. For examples where cells could decrease as well, we believe that it would be necessary to design new invariants with corresponding transformations. One interest of our approach is its flexibility, as exemplified in Section 8 on the Yubikey and CANauth protocols. One can easily adapt the approach to add a flavour of interactivity in ProVerif. Moreover, our transformations themselves are modular: each proof is independent from the other ones and quite simple. It is easy to add a new transformation and prove its soundness. However, the resulting, more complex, model may yield termination issue. For an integration in ProVerif, we envision a first pass with the original algorithm and, in a second step, only when the original algorithm could not prove the protocol, an automatic detection of precise channels and the application of our extension.

Adding natural numbers required to improve how ProVerif decides whether a query is satisfied. We believe that we could use similar ideas to revisit ProVerif's saturation algorithm itself by detecting earlier when a clause $H \Rightarrow C$ can be removed (e.g. when H does not satisfy inequalities between naturals or our properties). We expect that this should improve ProVerif in terms of efficiency. In this paper, we assume a typed attacker only w.r.t. the type nat. We plan to relax this condition and adapt the procedure to retrieve soundness even if the adversary may send a term that is not of the form $\mathsf{succ}^n(\mathsf{zero})$ where a nat is expected. We also plan to detect when the attacker is forced to comply with the type, in which case we could use finer properties (as it is done here).

Our introduction of natural numbers is sufficient for protocols with counters and tables. However, addition remains limited since two variables may not be added. As a future work, we plan to explore how to integrate a more general theory of addition into ProVerif, relying on more sophisticated algorithms on constraints on natural numbers.

The only protocol that we fail to address is the avionic protocol [11] as it requires to prove an injective property. We plan to explore how to (soundly) improve the treatment of disequalities for injective queries in ProVerif, as we did for non injective queries. We also plan to study whether GSVerif can scale up to protocol suites such as TLS1.3.

We considered correspondence and secrecy properties. Extending our approach to equivalence is not straightforward since, in ProVerif, (diff-)equivalence is directly encoded into processes. As future work, we plan to explore how to convey formula such as ϕ_{act}, ϕ_{cell}, …to the saturation procedure of ProVerif.

Acknowledgements. We would like to thank Jannik Dreier, Steve Kremer, Eike Ritter, Sebastian Mödersheim, and Alessandro Bruni for their interactions and helpful guidance for using their tools. We are also grateful to Bruno Blanchet for his advices and comments when modifying ProVerif.

This work has been partially supported by the European Research Council (ERC) under the European Unions Horizon 2020 research (grant agreement No 645865-SPOOC) and by the ANR project TECAP (ANR-17-CE39-0004-01).

References

[1] *Tamarin Manual.* https://tamarin-prover.github.io/manual/.

[2] *Trusted Computing Group. TPM Specification version 1.2. Parts 13, revision 103*, 2007.

[3] GSVerif. https://sites.google.com/site/globalstatesverif/, Jan. 2018.

[4] 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, 2001.

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

[6] 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 K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification, CAV'2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285, Edinburgh, Scotland, 2005. Springer.

[7] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *25th IEEE Computer Security Foundations Symposium (CSF'12)*, 2012.

[8] K. Bhargavan, R. Corin, C. Fournet, and E. Zalinescu. Cryptographically verified implementations for tls. In *15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 459–468, 2008.

[9] B. Blanchet. An automatic security protocol verifier based on resolution theorem proving (invited tutorial). In *20th International Conference on Automated Deduction (CADE-20)*, Tallinn, Estonia, July 2005.

[10] 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.

[11] 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, 2017.

[12] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre. Proverif 1.97: Automatic cryptographic protocol verifier, user manual and tutorial, 2017.

[13] A. Bruni, S. Moedersheim, F. Nielson, and H. R. Nielson. Set-pi: Set membership p-calculus. In *28th Computer Security Foundations Symposium (CSF 2015)*. IEEE, 2015.

[14] V. Cheval, V. Cortier, and M. Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in Proverif. Research report, Inria Nancy - Grand Est, Apr. 2018.

[15] V. Cortier, A. Filipiak, J. Florent, S. Gharout, and J. Traoré. Designing and proving an emv-compliant payment protocol for mobile devices. In *2nd IEEE European Symposium on Security and Privacy (EuroSP'17)*, pages 467–480, 2017.

[16] 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)*, London, UK, April 2018.

[17] V. Cortier and C. Wiedling. A formal analysis of the norwegian e-voting protocol. *Journal of Computer Security*, 25(15777):21–57, 2017.

[18] C. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc.*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.

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

[20] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel. Formal analysis of protocols based on TPM state registers. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*, pages 66–82. IEEE Computer Society Press, June 2011.

[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):162–202, 2006.

[22] D. Galindo, S. Guasch, and J. Puiggali. 2015 Neuchâtel's Cast-as-Intended Verification Mechanism. In *5th International Conference, (VoteID 2015)*, pages 3–18, 2015.

[23] J. A. Garay, M. Jakobsson, and P. MacKenzie. Abuse-free optimistic contract signing. In *19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'99)*, pages 449–466. Springer-Verlag, 1999.

[24] J. A. Garay, M. Jakobsson, and P. MacKenzie. *Abuse-Free Optimistic Contract Signing*, pages 449–466. Springer, 1999.

[25] A. V. Herrewege, D. Singelee, and I. Verbauwhede. CANAuth-A simple, backward compatible broadcast authentication protocol for CAN bus. In *Proceedings of ECRYPT*, 2011.

[26] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*, pages 435–450, 2017.

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

[28] S. Meier. *Advancing automated security protocol verification*. PhD thesis, ETH Zurich, 2013.

[29] S. Meier, B. Schmidt, C. Cremers, and D. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In Springer, editor, *International Conference on Computer Aided Verification (CAV'13)*, pages 696–701, 2013.

[30] S. Moedersheim and A. Bruni. Aif-omega: Set-based protocol abstraction with countable families. In *5th Conference on Principles of Security and Trust (POST'16)*, 2016.

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

[32] RSA Security Inc. *PKCS #11: Cryptographic Token Interface Standard*, v2.20 edition, 2004.

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

[34] A. Schutt and P. J. Stuckey. Incremental satisfiability and implication for UTVPI constraints. *INFORMS Journal on Computing*, 22(4):514–527, 2010.

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

Appendix A. More formal definitions

The semantics of processes is defined through a reduction relation $\rightarrow$ between configuration, defined in Figure 4.

As expected, the reduction rule NIL removes the process 0 from the multiset since it does nothing, the rule PAR apply the parallel composition and the rule REPL duplicates the process P hence modeling replication. The rules RES and PUB manage names. In particular, the rule PUB allows the attacker to generate new fresh public names whereas the rule RES renames private names. Note that the condition $a' \not\in \mathcal{N}_{\text{pub}} \cup \mathcal{N}_{\text{priv}} \cup names(P)$ is necessary to avoid confusion between a' and the names already declared, i.e. in $\mathcal{N}_{\text{pub}}$ or $\mathcal{N}_{\text{priv}}$, or not yet declared but present in the process, i.e. $names(P)$. The rule COMM allows internal communication between an output and input. The OUT allows the attacker to retrieve a message M sent on a channel N provided that he is able to deduce the channel. The deduction of N is modeled by the existence of a public expression D that can be evaluated to N containing only variables from the domain of the frame and declared public names. The rule IN is the direct counter part of the rule OUT as it allows the attacker to inject a message on a deducible channel. The rules LETIN and LETELSE define the semantics of the evaluation of an expression D. Note that the condition $D \Downarrow M$ expresses the fact that the evaluation of D succeeded since M is a term hence not the expression constant fail. The rules IFTHEN and IFELSE define the semantics of conditional branching. The rules TBLIN, TBLELSE and INSERT handle the management of tables. Note that in rule TBLIN, several entries in $\mathcal{S}$ may evaluate the expression D to true. Hence we may have different transitions, one for each entry that evaluates D to true.

Appendix B. First transformations

B.1. Unique Action

We provide proof sketches of the results stated in Section 4.1.

Lemma 1. *Let (E, P) be a valid configuration. For all correspondence queries $F \rightsquigarrow \phi$,*

$$E, P \models F \rightsquigarrow \phi \quad \textit{iff} \quad E, [P]_{act} \models F \rightsquigarrow \phi$$

Proof sketch. Since we assume that the added events do not already appear in the input file, they do not appear in ϕ nor P. Moreover, notice from the rule EVENT and RES that the event $\mathsf{UAction}(\mathsf{st}, x)$ and the name generation $\mathsf{new\ st : stamp}$ do not affect the behavior of the protocol. Specifically, we can easily show by induction on the length of the reduction that $\mathsf{tr} = (E, P) \rightarrow^* E', \mathcal{S}', \mathcal{P}', \Phi'$ if and only if $\mathsf{tr}' = E, [P]_{act} \rightarrow^* E' \cup N, \mathcal{S}', \mathcal{P}'', \Phi'$ for some $\mathcal{P}''$ and N containing the renamed instances of st. At the same time, we show that for all events $ev(M_1, \dots, M_n)$ with ev different from UAction, tr executes $ev(M_1, \dots, M_n)$ if and only if tr' executes $ev(M_1, \dots, M_n)$, which allows us to conclude. □

Lemma 2. *Let (E, P) be a valid initial configuration. For all names* st*, for all ground terms M_1, M_2, for all traces* $\mathsf{tr} = E, [P]_{act} \rightarrow^* E', \mathcal{S}', \mathcal{P}', \Phi'$*, if* tr *executes* $\mathsf{event}(\mathsf{UAction}(\mathsf{st}, M_1))$ *and* $\mathsf{event}(\mathsf{UAction}(\mathsf{st}, M_2))$ *then* $M_1 = M_2$.

Proof sketch. The proof of this lemma is straight forward since a fresh stamp is created before each event UAction. Hence, according to the rule RES of the semantics, for all traces $\mathsf{tr} = E, [P]_{act} \rightarrow^* E', \mathcal{S}', \mathcal{P}', \Phi'$, two distinct events UAction executed by tr necessarily have two distinct stamps. In other words, if tr executes $\mathsf{event}(\mathsf{UAction}(\mathsf{st}, M_1))$ and $\mathsf{event}(\mathsf{UAction}(\mathsf{st}, M_2)$ then $M_1 = M_2$. □

Theorem 1. *Let (E, P) be a valid initial configuration. Let $F \rightsquigarrow \phi$ be a correspondance query. We have*

$$E, P \models F \rightsquigarrow \phi \textit{ if and only if } E, [P]_{act} \models F \rightsquigarrow (\phi \vee \phi_{\mathsf{act}})$$

Proof. Direct from lemmas 1 and 2. □

B.2. Private channels

We now explain how to soundly deal with private channels, avoiding the common over-approximation of ProVerif, sketched in Section 2.2.

Our transformation only applies to strongly private channels. A name d is a *strongly private channel* w.r.t. a valid initial configuration $((\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), P)$ if $d \in \mathcal{N}_{\text{priv}}$ or d is bound (once) in P and if d only occurs as first argument of input or output in P. Not only this guarantees that d remains secret but also that we can syntactically detect when something is sent on d.

As explained in Section 2.2, we add a stamp (an identifier) to each sent message and we record any input. Formally, we consider a function symbol $\mathsf{UComm}(\mathsf{stamp}, \mathsf{stamp}) : \mathsf{bitstring}$ and we define our transformation as follows.

Definition 6. *Let (E, P) be a valid initial configuration. Let d be a strongly private channel in (E, P). We denote by $[P]^d_{com}$ the process P in which we replace*

- *every instance of* $\mathsf{out}(d, M); Q$ *by*

  $$\mathsf{new\ st : stamp; out}(d, (\mathsf{st}, M)); Q$$

  where st *is fresh;*
- *every instance of* $\mathsf{in}(d, x : T); Q$ *by*

  $$\begin{array}{l} \mathsf{new\ st : stamp;} \\ \mathsf{in}(d, (x_{st} : \mathsf{stamp}, x : T)); \\ \mathsf{event\ UComm}(x_{st}, \mathsf{st}); Q \end{array}$$

  where st *and* x_{st} *are fresh.*

Example 13. Consider the processes A' and A'_{com} as defined in Section 2.2, we have d is a strongly private channel of A' and $[A']^d_{com} = A'_{\mathsf{com}}$. ▶

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{0\}\!\}, \Phi \to E, \mathcal{S}, \mathcal{P}, \Phi \qquad (\textsc{Nil})$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{P \mid Q\}\!\} \to E, \mathcal{S}, \mathcal{P} \cup \{\!\{P, Q\}\!\}, \Phi \qquad (\textsc{Par})$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{!P\}\!\} \to E, \mathcal{S}, \mathcal{P} \cup \{\!\{P, !P\}\!\}, \Phi \qquad (\textsc{Repl})$$

$$(\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}), \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{new}\ a : T; P\}\!\}, \Phi \to (\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}} \cup \{a' : T\}), \mathcal{S}, \mathcal{P} \cup \{\!\{P\}\!\}, \Phi \qquad (\textsc{Res})$$
$$\text{if } a' \notin \mathcal{N}_{\mathrm{pub}} \cup \mathcal{N}_{\mathrm{priv}} \cup \mathit{names}(P)$$

$$(\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}), \mathcal{S}, \mathcal{P}, \Phi \to (\mathcal{N}_{\mathrm{pub}} \cup \{a : T\}, \mathcal{N}_{\mathrm{priv}}), \mathcal{S}, \mathcal{P}, \Phi \qquad (\textsc{Pub})$$
$$\text{if } a \notin \mathcal{N}_{\mathrm{pub}} \cup \mathcal{N}_{\mathrm{priv}} \cup \mathit{names}(\mathcal{P})$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{out}(N, M); P, \mathsf{in}(N, x : T); Q\}\!\}, \Phi \to E, \mathcal{S}, \mathcal{P} \cup \{\!\{P, Q\{M/x\}\}\!\}, \Phi \qquad (\textsc{Comm})$$
$$\text{if } M \text{ is of type } T$$

$$(\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}), \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{out}(N, M); P\}\!\}, \Phi \to (\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}), \mathcal{S}, \mathcal{P} \cup \{\!\{P\}\!\}, \Phi \cup \{M/z\} \qquad (\textsc{Out})$$
$$\text{if there exists } D \text{ such that } z \notin \mathit{dom}(\Phi),\ \mathit{fv}(D) \subseteq \Phi,\ \mathit{fn}(D) \subseteq \mathcal{N}_{\mathrm{pub}} \text{ and } D\Phi \Downarrow N$$

$$(\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}), \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{in}(N, x : T); P\}\!\}, \Phi \to (\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}), \mathcal{S}, \mathcal{P} \cup \{\!\{P\{M/x\}\}\!\}, \Phi \qquad (\textsc{In})$$
$$\text{if there exists } D_1, D_2 \text{ such that } \mathit{fv}(D_1, D_2) \subseteq \Phi,\ \mathit{fn}(D_1, D_2) \subseteq \mathcal{N}_{\mathrm{pub}},$$
$$D_1\Phi \Downarrow N,\ D_2\Phi \Downarrow M \text{ and } M \text{ is of type } T$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\}, \Phi \to E, \mathcal{S}, \mathcal{P} \cup \{\!\{P\{M/x\}\}\!\}, \Phi \qquad \text{if } D \Downarrow M \qquad (\textsc{LetIn})$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\}, \Phi \to E, \mathcal{S}, \mathcal{P} \cup \{\!\{Q\}\!\}, \Phi \qquad \text{if } D \Downarrow \mathsf{fail} \qquad (\textsc{LetElse})$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q\}\!\}, \Phi \to E, \mathcal{S}, \mathcal{P} \cup \{\!\{P\}\!\}, \Phi \qquad \text{if } M = \mathsf{true} \qquad (\textsc{IfThen})$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q\}\!\}, \Phi \to E, \mathcal{S}, \mathcal{P} \cup \{\!\{Q\}\!\}, \Phi \qquad \text{if } M \neq \mathsf{true} \qquad (\textsc{IfElse})$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{event}(M); P\}\!\}, \Phi \to E, \mathcal{S}, \mathcal{P} \cup \{\!\{P\}\!\}, \Phi \qquad (\textsc{Event})$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{get}\ \mathit{tbl}(x_1 : T_1, \dots, x_n : T_n)\ \mathsf{suchthat}\ D\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\}, \Phi \to E, \mathcal{S}, \mathcal{P} \cup \{\!\{P\{M_i/x_i\}_{i=1}^n\}\!\}, \Phi \qquad (\textsc{TblIn})$$
$$\text{if there exists } (\mathit{tbl}, M_1, \dots, M_n) \in \mathcal{S} \text{ such that } D\{M_i/x_i\}_{i=1}^n \Downarrow \mathsf{true}$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{get}\ \mathit{tbl}(x_1 : T_1, \dots, x_n : T_n)\ \mathsf{suchthat}\ D\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\}, \Phi \to E, \mathcal{S}, \mathcal{P} \cup \{\!\{Q\}\!\}, \Phi \qquad (\textsc{TblElse})$$
$$\text{if for all } (\mathit{tbl}, M_1, \dots, M_n) \in \mathcal{S},\ D\{M_i/x_i\}_{i=1}^n \not\Downarrow \mathsf{true}$$

$$E, \mathcal{S}, \mathcal{P} \cup \{\!\{\mathsf{insert}\ \mathit{tbl}(M_1, \dots, M_n); P\}\!\}, \Phi \to E, \mathcal{S} \cup \{(\mathit{tbl}, M_1, \dots, M_n)\}, \mathcal{P} \cup \{\!\{P\}\!\}, \Phi \qquad (\textsc{Insert})$$

Figure 4. Transitions between configurations.

Similarly to Lemma 1, this transformation does not modify the overall behaviour of the protocol.

Lemma 3. *Let (E, P) be a valid configuration. Let d be a strong private channel in (E, P). For all correspondence queries $F \rightsquigarrow \phi$,*

$$E, P \models F \rightsquigarrow \phi \quad \textit{iff} \quad E, [P]^d_{com} \models F \rightsquigarrow \phi$$

Proof sketch. We start by noticing that any strong private channels is not deducible from the attacker since strong private channels never appear inside an output message. Therefore, we deduce that the only possible semantics rule involving input or output on the strong private channel d is the rule Comm. Consider an instance of this rule, i.e. $\mathsf{tr} = E, P \to^* E', \mathcal{S}', \mathcal{P}' \cup \{\!\{\mathsf{out}(d, M); P, \mathsf{in}(d, x : T); Q\}\!\}, \Phi \to E', \mathcal{S}', \mathcal{P}' \cup \{\!\{P, Q\{M/x\}\}\!\} \to^* \dots$. Recall that in our transformation $[P]^d_{com}$, we replace every instance of $\mathsf{out}(d, M); P$ by $\mathsf{new\ st : stamp}; \mathsf{out}(d, (\mathsf{st}, M)); P$ and every instance $\mathsf{in}(d, x : T); Q$ by $\mathsf{new\ st : stamp}; \mathsf{in}(d, (x_{st} : \mathsf{stamp}, x : T)); \mathsf{event\ UComm}(x_{st}, \mathsf{st}); Q$. Thus in $E, [P]^d_{com}$, the corresponding application of the rule Comm will occur, that is between $\mathsf{out}(d, (\mathsf{st}, M))$ and $\mathsf{in}(d, (x_{st} : \mathsf{stamp}, x : T))$. Note that x will still be instanciated by M and x_{st} will be instantiated by st. □

Moreover, whenever $\mathsf{event}(\mathsf{UComm}(\mathsf{st}_1, \mathsf{st}_2))$ and $\mathsf{event}(\mathsf{UComm}(\mathsf{st}'_1, \mathsf{st}'_2)$ are executed in a trace, then $\mathsf{st}_1 = \mathsf{st}'_1$ iff $\mathsf{st}_2 = \mathsf{st}'_2$.

Lemma 4. *Let (E, P) be a valid configuration. Let d be a strong private channel in (E, P). For all names $\mathsf{st}_1, \mathsf{st}_2, \mathsf{st}'_1, \mathsf{st}'_2$, for all traces $\mathsf{tr} = E, [P]^d_{com} \to^* E', \mathcal{S}', \mathcal{P}', \Phi'$, if tr executes $\mathsf{event}(\mathsf{UComm}(\mathsf{st}_1, \mathsf{st}_2))$ and $\mathsf{event}(\mathsf{UComm}(\mathsf{st}'_1, \mathsf{st}'_2)$ then $\mathsf{st}_1 = \mathsf{st}'_1$ iff $\mathsf{st}_2 = \mathsf{st}'_2$.*

Proof sketch. Due to the freshness of the stamps added in $[P]^d_{com}$, the stamp st in a message (st, M) input on a process $\mathsf{in}(d, (x_{st}, x))$ indicates which output of the process $[P]^d_{com}$ was used in the transition rule Comm. Similarly, the fresh stamps added in front of each inputs also indicates which input of the process was used. □

These are the main ingredient to show that is sound to query $\phi \vee \phi_{\mathsf{com}}$ instead of ϕ, where ϕ_{com} has been introduced in Section 2.2.

Theorem 5. *Let (E, P) be a valid configuration. Let d be a strong private channel in (E, P). Let $F \rightsquigarrow \phi$ be a correspondance query. We have:*

$$E, P \models F \rightsquigarrow \phi \ \textit{iff}\ E, [P]^d_{com} \models F \rightsquigarrow \phi \vee \phi_{\mathsf{com}}$$

Proof. Direct from lemmas 3 and 4 □

Example 14. Continuing Example 13 and thanks to Theorem 5, to show that $((\emptyset, \{s\}), A')$ preserves the secrecy of s, it is sufficient to check whether $((\emptyset, \{s\}), [A']^d_{com}) \models \mathsf{attacker}(s) \rightsquigarrow \phi_{\mathsf{com}}$. Interestingly, ProVerif can now (automatically) prove the later property, hence proving that A' preserves the secrecy of s. ▶

B.3. Cells

We provide in this section a more formal definition of a cell.

In order to formally define what a cell is, we first consider the *read* or *write* operations.

Definition 7. *Let (E, P) be a valid configuration. Let d be a strong private channel in (E, P). Let P' a sub-process of P. We say that P' may* directly read *(resp.* write *on d) if $P' = C[P_1, \dots, P_n]$ for some process context C and some processes $P_1, \dots, P_n$ such that:*

- *C does not contain an instance of $\mathsf{in}(d, x : T)$ or $\mathsf{out}(d, M)$ for any $x : T, M$*
- *there exists $i \in \{1, \dots, n\}$ such that $P_i = \mathsf{in}(d, x : T); Q$ for some $x : T$ and Q (resp. $\mathsf{out}(d, M); Q$ for some M and Q)*

We sometimes say that P' may directly write on d by the subprocess $\mathsf{out}(d, M); Q$ when we need to specify where the output occurred.

Intuitively, a process may directly write on d if an output on d is syntactically available from P without having to previously input on d. Similarly for directly reading on d. We say syntactically because it is possible that no trace can reach such output semantically.

Example 15. Consider $P_1 = \mathsf{new}\ d; (\mathsf{out}(d, M) \mid \mathsf{in}(c, x); \mathsf{in}(d, y))$ and $P_2 = \mathsf{new}\ d; \mathsf{in}(c, x); \mathsf{in}(d, y); \mathsf{out}(d, M)$. The process P_1 may both read and write on d but P_2 may only directly read on d. ▶

A cell can now be formalized as follows.

Definition 8. *Let (E, P) be a valid configuration. Let d be a strong private channel in (E, P). We say that d is a cell in (E, P) if for all subprocesses P' of P,*

- *$P' = Q_1 \mid Q_2$ implies Q_1 and Q_2 may not both directly write on d; and*
- *$P' = \mathsf{in}(d, x : T); Q$ implies Q may not directly read on d; and*
- *$P' = \mathsf{out}(d, M); Q$ implies Q may not directly write on d; and*
- *$P' = !Q$ implies Q may not directly write on d.*

Let us now recall the definition of our transformation for cell.

Definition 4. *Let (E, P) be a valid configuration and d be a cell in (E, P). We denote by $[P]^d_{cell}$ the process obtained from P by replacing any subprocess $P' = \mathsf{in}(d, x : T); C[\mathsf{out}(d, M_1); Q_1, \dots, \mathsf{out}(d, M_n); Q_n]$ (where C does not contain inputs nor outputs on d) by*

$$\begin{array}{l}\mathsf{in}(d, (x_{st} : \mathsf{stamp}, x : T));\\ \mathsf{event}\ \mathsf{VCell}_T(x_{st}, x);\\ C[Q'_1, \dots, Q'_n]\end{array}$$

where Q'_i is defined as follows:

- *if $M_i = x$ then $Q'_i = \mathsf{out}(d, (x_{st}, x)); Q_i$, this corresponds to the case where the value of the cell does not change so we do not need to annotate this action;*
- *otherwise*

$$\begin{array}{rl}Q'_i = & \mathsf{new}\ st : \mathsf{stamp};\\ & \mathsf{event}\ \mathsf{VCell}_T(st, M_i);\\ & \mathsf{event}\ \mathsf{VLink}(x_{st}, st);\\ & \mathsf{out}(d, (\mathsf{st}, M_i)); Q_i\end{array}$$

Moreover, if P contains a subprocess $\mathsf{out}(d, M); Q$ that is not preceeded by an input on d (initialization case), then it is replaced by $\mathsf{new}\ st : \mathsf{stamp}; \mathsf{out}(d, (st, M)); Q$.

Following the techniques of the previous transformations, we can show that it is safe to query $\phi \vee \phi_{\mathsf{cell}}$ on $[P]^d_{cell}$ instead of ϕ on P, where ϕ_{cell} has been defined in Section 2.3.

Lemma 5. *Let (E, P) be a valid configuration. Let d be cell in (E, P). For all correspondence query $F \rightsquigarrow \phi$,*

$$E, P \models F \rightsquigarrow \phi \quad \textit{iff} \quad E, [P]^d_{cell} \models F \rightsquigarrow \phi$$

Proof. The transformation in $[P]^d_{cell}$ involves the same transformation as in $[P]^d_{com}$ and $[P]_{act}$, that is adding events inside the process and stamps in the input and output messages of a strong private channels. Therefore, we show this lemma by reusing the same reasonning as in lemmas 2 and 4. □

We can now express the fact that within a trace, the value of the a cell are modified sequentially.

Lemma 6. *Let (E, P) be a valid configuration. Let d be a cell in (E, P). For all names $\mathsf{st}_1, \mathsf{st}_2, \mathsf{st}'_1, \mathsf{st}'_2$, for all terms M, N for all traces $\mathsf{tr} = E, [P]^d_{cell} \rightarrow^* E', \mathcal{S}', \mathcal{P}', \Phi'$,*

- *if tr executes $\mathsf{event}(\mathsf{VLink}(\mathsf{st}_1, \mathsf{st}_2))$ and $\mathsf{event}(\mathsf{VLink}(\mathsf{st}'_1, \mathsf{st}'_2)$ then $\mathsf{st}_1 = \mathsf{st}'_1$ iff $\mathsf{st}_2 = \mathsf{st}'_2$.*
- *if tr executes $\mathsf{event}(\mathsf{VCell}(\mathsf{st}_1, M))$ and $\mathsf{event}(\mathsf{VCell}(\mathsf{st}_1, N))$ then $N = M$.*

Proof sketch. Following the definition 8 of cells and our transformation (definition 4), we prove by induction on the number of transition steps in the trace $\mathsf{tr} = E, P \rightarrow^* E', \mathcal{S}', \mathcal{P}', \Phi'$ that tr executed sequentially the events $\mathsf{VLink}(st_0, st_1), \mathsf{VLink}(st_1, st_2), \dots, \mathsf{VLink}(st_{n-2}, st_{n-1})$, $\mathsf{VLink}(st_{n-1}, st_n)$ for some $n \in \mathbb{N}$ and some pairwise distinct names $st_0, \dots, st_n$ of type stamp. Moreover, since we know that the stamp value in the cell either is replaced by a fresh one or left unchanged when we know syntactically that the content of the cell does not change, we directly obtain that if tr executes $\mathsf{event}(\mathsf{VCell}(\mathsf{st}_1, M))$ and $\mathsf{event}(\mathsf{VCell}(\mathsf{st}_1, N))$ then $N = M$. □

Theorem 2. *Let (E, P) be a valid configuration. Let d be a cell in (E, P). Let $F \rightsquigarrow \phi$ be a correspondence query. We have:*

$$E, P \models F \rightsquigarrow \phi \text{ iff } E, [P]^d_{cell} \models F \rightsquigarrow (\phi \vee \phi_{cell})$$

Proof. Direct from lemmas 5 and 6. □

Example 16. Continuing Example 7, ProVerif can prove that $(E, [P_{\mathsf{cell}}]^d_{cell}) \models \mathsf{attacker}((s_l, s_r)) \rightsquigarrow \phi_{com}$. Thanks to Theorem 2, we deduce $(E, P_{\mathsf{cell}}) \models \mathsf{attacker}((s_l, s_r))$. ▶

Appendix C. ProVerif procedure with natural numbers

To soundly extend ProVerif to natural numbers, we proceed in two steps:

1) First, we implement the algorithm of Pratt [31] for checking satisfiability of inequalities. Not only we can decide (in polynomial time) whether a set of inequalities can be satisfied, but we also detect *forced equalities*. For example, the set of inequalities $\{x \leq y+1, y < x, z \leq 3\}$ has solutions and any solution is such that $x = y + 1$.
2) Second, we refine ProVerif's procedure in order to better detect when the queried properties are satisfied.

C.1. Checking inequalities

In this section we show how we compute a unifier of a conjunction of inequalities. Our algorithm will also allow us to determine if a solution exists. Recall that terms of type nat are necessarily either n or $x+n$ with $n \in \mathbb{N}$ and x a variable of type nat. Moreover, any conjunction of inequalities $M < N \wedge \phi$ have the same solutions as $M + 1 \leq N \wedge \phi$. Finally, if a conjunction ϕ contains inequality $n < m$ or $n \leq m$ with $n, m \in \mathbb{N}$, it is trivial to deduce if these particular inequalities holds. If they hold then we can remove them from ϕ else ϕ does not have any solution.

Therefore, we can reduce our problem to a conjunction in the canonical form $\bigwedge_i x_i - y_i \leq k_i \wedge \bigwedge_j z_i \leq p_i \wedge \bigwedge_\ell -w_\ell \leq q_i$ where x_i, y_i, z_j, w_ℓ are variables, $k_i, p_i, q_i \in \mathbb{Z}$ and we want to determine its solutions in $\mathbb{N}$. To do so, we will transform this formula into a weighted directed graph

Definition 9. *A weighted directed graph is a tuple (V, E, ω) where V the set of vertices, $E \subseteq V \times V$ the set of edges and $\omega : E \to \mathbb{Z}$ a function that associates edges with their weight.*

Given two vertices $v, v' \in V$, we will denote $v \xrightarrow{k} v'$ when $(v, v') \in E$ and $k = \omega((v, v'))$. We will denote $v \overset{k}{\Rightarrow} v'$ when $v \xrightarrow{k_1} v_1 \xrightarrow{k_2} \dots \xrightarrow{k_n} V'$ with $k = k_1 + k_2 + \dots + k_n$.

Finally for any weighted directed graph (V, E, ω) without negative cycle (i.e. for all $v, v' \in V$, $v \overset{k}{\Rightarrow} v'$ implies $k \geq 0$), we consider the partial function $\omega_{short} : V \times V \to \mathbb{Z}$ defined on all pairs of vertices $v, v' \in V$ with $v \overset{k}{\Rightarrow} v'$ for some k and such that $\omega_{short}(v, v') = \min\{k \mid v \overset{k}{\Rightarrow} v'\}$.

The function $\omega_{short}(v, v')$ typically computes the weight of the shortest path between v and v' when a path exists, otherwise $\omega_{short}(v, v')$ is not defined.

We now can defined the weighted directed graph modeling a conjunction of inequality in canonical form.

Definition 10. *Let ϕ be a conjunction of inequality in canonical form. We denote by $G(\phi)$ the weighted directed graph (V, E, ω) such that $V = fv(\phi) \cup \{0\}$, E is a smallest set such that*

- *for all $x - y \leq k$ in ϕ, $(x, y) \in E$; and*
- *for all $x \leq k$ in ϕ, $(x, 0) \in E$; and*
- *for all $x \in fv(\phi)$, $(0, x) \in E$;*

and for all $(x, y) \in E$, $\omega(x, y) = \min\{k \mid x - y \leq k \in \phi\}$; for all $(x, 0) \in E$, $\omega(x, y) = \min\{k \mid x \leq k \in \phi\}$; and for all $(0, x) \in E$, $\omega(0, x) = \min(\{0\} \cup \{k \mid -x \leq k \in \phi\})$.

Such a construction is provided in Example 9.

By representing the conjunction of inequality into directed graph, we can easily express relations between variables. We will say that σ is solution of ϕ if σ is closing for ϕ and $\sigma \models \phi$.

Lemma 7. *Let ϕ be a conjunction of inequalities in canonical form and let $G(\phi) = (V, E, \omega)$ its associated graph. For all $v, v' \in V$, if $v \overset{k}{\Rightarrow} v'$ then for all σ solution of ϕ, $\sigma \models v - v' \leq k$.*

Proof. We know that $v \xrightarrow{k} v'$ implies that either $v - v' \leq k \in \phi$ or $v = 0$, $v' = x$ and $k = 0$. In both cases, we obtain that for all σ, $\sigma \models \phi$ implies that $\sigma \models v - v' \leq k$. Therefore, since $v \overset{k}{\Rightarrow} v'$ implies $v \xrightarrow{k_1} v_1 \xrightarrow{k_2} \dots \xrightarrow{k_{n-1}} v_{n-1} \xrightarrow{k_n} v'$ with $k = k_1 + \dots + k_n$, we deduce that $\sigma \models v - v_1 \leq k_1 \wedge v_1 - v_2 \leq k_2 \wedge \dots \wedge v_{n-1} - v' \leq k_n$. By summing all these inequalities, we obtain that $v - v' \leq k_1 + k_2 + \dots + k_n$. □

This property is all we need to determine if a conjunction of inequalities has a solution and if yes, to obtain a unifier.

Theorem 6. *Let ϕ be a conjunction of inequalities in canonical form and let $G(\phi) = (V, E, \omega)$ its associated graph. The following two properties hold:*

1) *ϕ has a solution if and only if $G(\phi)$ is without negative cycle*
2) *for all $v, v' \in V$, if $\omega_{short}(v, v') + \omega_{short}(v', v) = 0$ then for all σ solution of ϕ, $\sigma \models v = v' + \omega_{short}(v, v')$*

Proof. Let us focus on the property 1. Assume that ϕ has a solution σ and assume by contradiction that $G(\phi)$ contains a negative cycle. Hence there exists $v, v' \in V$ such that $v \overset{k}{\Rightarrow} v'$ and $v' \overset{k'}{\Rightarrow} v$ with $k + k' < 0$. By Lemma 7, we deduce that $\sigma \models v - v' \leq k$ and $\sigma \models v' - v \leq k'$. By summing the two, we obtain $0 \leq k + k'$ which is a contradiction with $k + k' < 0$.

Assume now that $G(\phi)$ is without negative cycle. Consider σ such that for all $x \in fv(\phi)$, $x\sigma = -\omega_{short}(0, x)$. We show that $\sigma \models \phi$. Let first need to prove that $x\sigma \in \mathbb{N}$. By definition of $G(\phi)$, we know that for all $x \in fv(v)$, $0 \to kx$ for some $k \leq 0$. Hence $0 \xRightarrow{\omega_{short}(0,x)} x$ with

$\omega_{short}(0,x) \le k \le 0$. Thus $-\omega_{short}(0,x) \ge 0$ and so $x\sigma \ge 0$.

Consider now $x - y \le k \in \phi$. Note that there exists $x - y \le k' \in \phi$ such that $k' < k$ then $\sigma \models x - y \le k'$ implies $\sigma \models x - y \le k$. Thus, we only need to focus on the inequalities $x - y \le k$ in ϕ with k minimal. Hence, by definition of $G(\phi)$, we know $x \xrightarrow{k} y$. Moreover, we also know that there is a path from 0 to x and a path from 0 to y. Hence, $0 \xRightarrow{\omega_{short}(0,x)} x \xrightarrow{k} y$ and $0 \xRightarrow{\omega_{short}(0,y)} y$. By definition of $\omega_{short}(0,y)$, we deduce that $\omega_{short}(0,y) \le \omega_{short}(0,x) + k$. Thus $(-\omega_{short}(0,x)) - (-\omega_{short}(0,y)) \le k$ and so $x\sigma - y\sigma \le k$.

Consider now $-x \le k \in \phi$. Once again, we take w.l.o.g. k minimal. By definition of $G(\phi)$, $0 \xrightarrow{k} x$. Thus, $0 \xRightarrow{\omega_{short}(0,x)} x$ with $\omega_{short}(0,x) \le k$. With $x\sigma = -\omega_{short}(0,x)$, we conclude that $-x\sigma \le k$.

Finally, consider $x \le k \in \phi$ with k minimal. By definition of $G(\phi)$, $x \xrightarrow{k} 0$. Moreover, we also know that there is a path from 0 to x. Hence $0 \xRightarrow{\omega_{short}(0,x)} x$. Since we assumed that there is negative cycle, we deduce that $\omega_{short}(0,x) + k \ge 0$ and so $-\omega_{short}(0,x) \le k$ meaning that $x\sigma \le k$.

For property 2, we only need to apply lemma 7. Indeed, if $\omega_{short}(v,v') + \omega_{short}(v',v) = 0$ then it implies that $v \xRightarrow{\omega_{short}(v,v')} v'$ and $v' \xRightarrow{\omega_{short}(v',v')} v$. Thus, by lemma 7, $v - v' \le \omega_{short}(v,v')$ and $v' - v \le \omega_{short}(v',v)$. With $\omega_{short}(v,v') + \omega_{short}(v',v) = 0$, we deduce that $v \le v' + \omega_{short}(v,v')$ and $v \ge v' + \omega_{short}(v,v')$ which allows us to conclude. □

The second property of Theorem 6 allows us to build a unifier. Note that $\omega_{short}(v,v')$ is not necessarily positive. However recall that syntactically speaking, a term is necessarily of the form $x+n$ with $n \in \mathbb{N}$. Thus, to build the term substitution σ unifier of ϕ, we consider all pair of vertices v, v', check if $\omega_{short}(v,v') + \omega_{short}(v',v) = 0$; if so we check if $\omega_{short}(v,v') \ge 0$. If so then we add the equality $v' + \omega_{short}(v,v') = v$ else we add $v + (-\omega_{short}(v,v')) = v'$. To finally obtain the unifier σ, we compute the most general unifier of all added equalities.

Example 17. Coming back to example 9, one can note that there is no negative cycle. Moreover, $\omega_{short}(x,y) = -1$ and $\omega_{short}(y,x) = 1$. Hence we have the equality $x + 1 = y$. Similarly, we have $\omega_{short}(y,z) = -1$, $\omega_{short}(z,y) = 1$ and $\omega_{short}(z,t) = -1$, $\omega_{short}(t,z) = 1$ and $\omega_{short}(t,x) = 3$, $\omega_{short}(x,t) = -3$ which gives us $y + 1 = z$, $z + 1 = t$ and $t = x + 3$. To obtain the unifier σ, we compute the most general unifier of $\{x+1 = y, y+1 = z, z+1 = t, t = x+3\}$ which gives us $\sigma = \{x+3/t; x+2/z; x+1/y\}$.

Note that $\omega_{short}(0,x) = 0$, $\omega_{short}(0,y) = -1$, $\omega_{short}(0,z) = -2$ and $\omega_{short}(0,t) = -3$. Indeed we have that the substitution $\{0/x; 1/y; 2/z; 3/z\}$ is a solution of ϕ.

Finally, if instead of $t \le x + 3$ we had $t \le x + 2$ in ϕ (leading to a formula without solution) then we would have obtained a negative cycle in the corresponding graph. ▶

C.2. Extending ProVerif algorithm

We now prove Theorem 3.

Theorem 3. *Let* (E,P) *be a valid initial configuration. Let* $F \rightsquigarrow \phi$ *be a correspondence query. If for all* $H \Rightarrow C \in \mathsf{solve}_{E,P}(F)$, `verif(`$H \Rightarrow C, F \rightsquigarrow \phi$`)` = true *then* $E, P \models F \rightsquigarrow \phi$.

As for the original verification procedure `verifPV`, our verification procedure will aim to prove that all instantiations of the Horn clauses in $\mathsf{solve}_{E,P}(F)$ satisfy the query $F \rightsquigarrow \phi$. This is formalized by the following three definitions.

Definition 11. *Let* $H \Rightarrow C$ *be a Horn clause. Let* σ *be a substitution closing for* $H \Rightarrow C$*. We say that* σ *is a* valid instantiation *of* $H \Rightarrow C$ *if for all* $M\ op\ N$ *in* H *with* $op \in \{=, \neq, <, \le\}$, $M\sigma\ op\ N\sigma$ *holds.*

By requiring all equality, disequality and natural number predicates to hold, we ensure that the instantiation σ does not produce a Horn clause with false hypotheses. Since the satisfaction of a fact of the form $M\ op\ N$ with $op \in \{=, \neq, <, \le\}$ only depends on the substitution instantiating its variables, we will denote from now on $\sigma \models M\ op\ N$ when $M\sigma\ op\ N\sigma$ holds with $M\sigma$ and $N\sigma$ not necessarily ground (i.e. $\sigma \models M \neq N$ does not necessarily imply that $\sigma\sigma' \models M \neq N$ for all σ').

Definition 12. *Let* $H \Rightarrow C$ *be a Horn clause. Let* $F \rightsquigarrow \phi$ *be a query with* $\phi = \bigvee_i^n \bigwedge_j^{m_i} F_{i,j}$*. Let* σ *be a valid instantiation of* $H \Rightarrow C$*. We say that* the query $F \rightsquigarrow \phi$ holds for $H \Rightarrow C$ with σ, *denoted* $(H \Rightarrow C), \sigma \models F \rightsquigarrow \phi$, *when there exist* $i \in \{1, \dots, n\}$ *and a substitution* σ' *such that* $F\sigma' = C\sigma$ *and for all* $j \in \{1, \dots, n\}$,

- *if* $F_{i,j} = M\ op\ N$ *with* $op \in \{=, \neq, <, \le\}$ *then* $\sigma' \models M\ op\ N$ *holds*
- *else* $F_{i,j}\sigma'$ *is in* $H\sigma$.

Recall from section 5 that correspondence queries are of the form $F \rightsquigarrow \bigvee_{i=1}^n \bigwedge_{j=1}^{m_i} F_{i,j}$ where the $F_{i,j}$ are either equality facts (EF), disequality facts (DF), inequality facts (IF), or other facts (OF), that is events or predicates that are not less nor lesseq. Moreover, we say that a subquery $\bigwedge_{j=1}^{m_i} F_{i,j}$ is *specialized* if $fv(\mathsf{IF}) \subseteq fv(\mathsf{OF}, F)$ and $fv(\mathsf{DF}) \subseteq fv(\mathsf{OF}, F)$.

Lemma 8. *Let* $H \Rightarrow C$ *be a Horn clause. Let* $F \rightsquigarrow \phi$ *be a query. For all specialized subquery* $\phi_{\mathsf{OF}} \wedge \phi_{\mathsf{EF}} \wedge \phi_{\mathsf{DF}} \wedge \phi \mathsf{IF}$ *in* ϕ*, for all substitution* σ'*, if* $\sigma' \models \phi_{\mathsf{EF}}$, $\phi_{\mathsf{OF}}\sigma' \subseteq H$ *and* $F\sigma' = C$ *then for all valid instantiations* σ *of* $H \Rightarrow C$, $(H \Rightarrow C), \sigma \models F \rightsquigarrow \phi$ *iff one of the following properties hold:*

- $\sigma \models \phi_{\mathsf{IF}}\sigma' \wedge \phi_{\mathsf{DF}}\sigma'$
- *there exist* $M \neq N \in \phi_{\mathsf{DF}}\sigma'$ *and two substitutions* δ, α *such that* $\delta = mgu(M,N)$ *and* $\sigma = \delta\alpha$
- *there exists* $M < N \in \phi_{\mathsf{IF}}\sigma'$ *such that* $(H \wedge N \le M \Rightarrow C), \sigma \models F \rightsquigarrow \phi$
- *there exists* $M \le N \in \phi_{\mathsf{IF}}\sigma'$ *such that* $(H \wedge N < M \Rightarrow C), \sigma \models F \rightsquigarrow \phi$

Proof. The lemma can in fact be seen as a distinction case on the substitution σ. Since we know that $\sigma' \models \phi_{\mathsf{EF}}$ and $\phi_{\mathsf{OF}}\sigma'$, we deduce that $\sigma'\sigma \models \phi_{\mathsf{EF}}$ and $\phi_{\mathsf{OF}}\sigma'\sigma \subseteq H\sigma$. By distinction case, we know that either $\sigma \models \phi_{\mathsf{IF}}\sigma' \wedge \phi_{\mathsf{DF}}\sigma'$ or $\sigma \not\models \phi_{\mathsf{IF}}\sigma' \wedge \phi_{\mathsf{DF}}\sigma'$. In the former case, we directly obtain that $(H \Rightarrow C), \sigma \models F \rightsquigarrow \phi$ thanks to the subquery $\phi_{\mathsf{OF}} \wedge \phi_{\mathsf{EF}} \wedge \phi_{\mathsf{DF}} \wedge \phi_{\mathsf{IF}}$. The latter case implies that $\sigma \not\models M\sigma'$ *op* $N\sigma'$ for some M *op* $N \in \phi_{\mathsf{IF}} \cup \phi_{\mathsf{DF}}$. If *op* $= \neq$ then it implies that $\sigma \models M\sigma' = N\sigma'$ and so σ is a unifier of $M\sigma'$ and $N\sigma'$. If *op* $= <$ then we deduce that $\sigma \models N\sigma' \leq M\sigma'$. Thus $(H \Rightarrow C), \sigma \models F \rightsquigarrow \phi$ implies $(H \wedge N\sigma' \leq M\sigma' \Rightarrow C), \sigma \models F \rightsquigarrow \phi$. Note that $(H \wedge N\sigma' \leq M\sigma' \Rightarrow C), \sigma \models F \rightsquigarrow \phi$ implies by definition that $(H \Rightarrow C), \sigma \models F \rightsquigarrow \phi$. Similar case if *op* $= \leq$. □

The proof of Theorem 3 is a direct application of lemmas 2 and 8.

Example 18. Consider the process $P' = \mathsf{in}(c, y)$; $\mathsf{event}\ e(y); \mathsf{out}(c, y)$ and the query $\mathsf{event}(e(x)) \rightsquigarrow 0 \leq x$, denoted $F \rightsquigarrow \phi$. By applying $\mathsf{solve}_{E,P'}(\mathsf{event}(e(x)))$, we obtain the single clause $\mathsf{attacker}(y) \Rightarrow \mathsf{event}(e(y))$, denoted $H \Rightarrow C$. The computation of `verif` on $H \Rightarrow C$ and the query goes as follows:

1) $H \Rightarrow C$ is already simplified, i.e. $H' \Rightarrow C' = H \Rightarrow C$.
2) $0 \leq x$ is specialized subquery with $\mathsf{OF} = \emptyset$, $\mathsf{EF} = \emptyset$, $\mathsf{DF} = \emptyset$ and $\mathsf{IF} = \{0 \leq x\}$.
3) Since with $\sigma = \{y/x\}$, we have that $0 \leq y$ has a solution and $F\sigma = C'$ then we have to compute `verif(`$H \wedge x < 0 \Rightarrow C,\ F \rightsquigarrow \phi$`)`.
4) However the inequality $x < 0$ has no solution, hence `simplify(`$H \wedge x < 0 \Rightarrow C$`)` raises the exception False_hypothesis which implies that `verif(`$H \wedge x < 0 \Rightarrow C,\ F \rightsquigarrow \phi$`)` = true.
5) Therefore `verif(`$H \Rightarrow C, F \rightsquigarrow \phi$`)` = true

Consider now the query $\mathsf{event}(e(x)) \rightsquigarrow x = a \vee x \neq a$, denoted $F' \rightsquigarrow \phi'$. The computation of `verif` on $H \Rightarrow C$ and $F' \rightsquigarrow \phi'$ goes as follows:

1) $H \Rightarrow C$ is already simplified, i.e. $H' \Rightarrow C' = H \Rightarrow C$.
2) $x = a$ and $x \neq a$ are two specialized subqueries but only $x \neq a$ allows us to obtain a substitution $\sigma = \{y/x\}$ satisfying the conditional. Indeed, with the subquery $x = a$, $\sigma \not\models x = a$.
3) Since $\{a/y\} = mgu(y, a)$, we have to compute `verif(`$H\{a/y\} \Rightarrow C\{a/y\},\ F' \rightsquigarrow \phi'$`)`.
4) However, $H\{a/y\} \Rightarrow C\{a/y\}$ is the clause $\mathsf{attacker}(a) \Rightarrow \mathsf{event}(e(a))$. Hence considering the specialized subquery $x = a$, we obtain that there exists $\sigma' = \{a/x\}$ such that $\mathsf{event}(e(x))\sigma' = \mathsf{event}(e(a))$ and $\sigma' \models x = a$.
5) Therefore `verif(`$H\{a/y\} \Rightarrow C\{a/y\},\ F' \rightsquigarrow \phi'$`)` = true and so `verif(`$H \Rightarrow C, F' \rightsquigarrow \phi'$`)` = true. ▶

C.3. Transformation for tables

Here, we explain for to prove properties on protocols with tables, as informally described in Example12. Intuitively, a *locked table* is a table protected by a cell.

Definition 13. *Let (E, P) be a valid configuration. Let d be a cell in (E, P). Let tbl a table. We say that tbl* is a locked table w.r.t. *d in (E, P) if the following two properties hold:*

- *any insert or lookup operation on tbl occurs within a round of d in P;*
- *in any round of d in P, an insert on tbl cannot occur before a lookup on tbl, i.e. for all subprocesses* $\mathsf{insert}\ \mathit{tbl}(M_1, \ldots, M_n); C[\mathsf{out}(d, N_1); Q_1, \ldots, \mathsf{out}(d, N_n); Q_n]$ *in P, C does not contain an instance of* $\mathsf{get}\ \mathit{tbl}(\mathit{pat}_1, \ldots, \mathit{pat}_n)$ in R_1 else R_2.

The second condition can be seen as a programming disciple. Indeed, the lookup operation tries to see if some specific element is in the table. Thus, if that specific element was already inserted within the same round, the lookup operation becomes unnecessary. We introduce events $\mathsf{InTbl}(i, t)$ and $\mathsf{NotInTbl}(i, t)$ which indicate that an element t is (resp. is not) in a table at step i.

Definition 14. *Let pat be a pattern. We say that $[M_1, \ldots, M_n]$ are the fixed terms of pat if there exists C such that pat $= C[= M_1, \ldots, = M_n]$ and C does not contain $= M$ for any M. Moreover, we say that $C[M_1, \ldots, M_n]$ is the term of pat, denoted $\overline{pat}$.*

Let (E, P) be a valid configuration. For each tbl locked table w.r.t. d in (E, P) defined as $\mathit{tbl}(T_1, \ldots, T_n)$, we define the following events:

- $\mathsf{InTbl}_{\mathit{tbl}}(\mathsf{nat}, T_1, \ldots, T_n)$.
- $\mathsf{NotInTbl}_{\mathit{tbl},\mathit{pat}}(\mathsf{nat}, T'_1, \ldots, T'_m)$ *for each* $\mathsf{get}\ \mathit{tbl}(\mathit{pat}_1, \ldots, \mathit{pat}_n)$ in Q_1 else Q_2 *in P where $[M_1, \ldots, M_m]$ are the fixed terms of the pattern $\mathit{pat} = (\mathit{pat}_1, \ldots, \mathit{pat}_n)$ and for all $j \in \{1, \ldots, m\}$, M_j is a term of type T'_j.*

Example 19. Coming back to example 12, we have only one table and one lookup operation with the pattern $= x_a$. Moreover, x_a is the only fixed term of $= x_a$. Thus, we will generate the event $\mathsf{InTbl}(\mathsf{nat}, \mathsf{bitstring})$ and the event $\mathsf{NotInTbl}(\mathsf{nat}, \mathsf{bitstring})$ (since we only one event of each, we don't indices for simplicity). ▶

We can now define our transformation on locked table as follows.

Definition 15. *Let (E, P) be a valid initial configuration. Let tbl be a locked table w.r.t. d in (E, P). We denote by $[P]^{\mathit{tbl},d}_{\mathit{lock}}$ the process P such that we replace any round:*

- $\mathsf{in}(d, x : T); C[\mathsf{out}(d, M_j); Q_j]_j$ *of d in P that does not contain insert or lookup operation on tbl by*

$$\mathsf{in}(d, (i : \mathsf{nat}, x : T)); C[\mathsf{out}(d, (i, M_j)); Q_j]_j$$

- $\mathsf{in}(d, x : T); C[\mathsf{out}(d, M_j); Q_j]_j$ *of d in P containing an insert or lookup operation on tbl by*

$$\mathsf{in}(d, (i : \mathsf{nat}, x : T)); C'[\mathsf{out}(d, (i + 1, M_j)); Q_j]_j$$

where C' is the context C in which we replace any instance of $\mathsf{insert}\ \mathit{tbl}(N_1, \ldots, N_n); Q$ *by* $\mathsf{event}\ \mathsf{InTbl}(i + 1, N_1, \ldots, N_n); \mathsf{insert}\ \ \mathit{tbl}(N_1, \ldots, N_n); Q$*; and we* *replace any instance of* get $tbl(pat_1, \dots, pat_n)$ in Q_1 else Q_2 *by*

```
get $tbl(pat_1, \dots, pat_n)$ in
   event $\mathsf{InTbl}(i, \overline{pat_1}, \dots, \overline{pat_n}); Q_1$
else event $\mathsf{NotInTbl}_{tbl,pat}(i, L_1, \dots, L_k); Q_2$
```

where $[L_1, \dots, L_k]$ *are the fixed terms of* $pat = (pat_1, \dots, pat_n)$.

- *if* P *may directly write on* d *by the subprocess* $\mathsf{out}(d, M); Q$ *then we replace* $\mathsf{out}(d, M); Q$ *by* $\mathsf{out}(d, (0, M)); Q$.

Example 20. As described in example 12, the process $[P]^{VoterTbl,d}_{lock}$ is the process $!S' \mid \mathsf{new}\ a; \mathsf{out}(d, (0, a) \mid !\mathsf{in}(d, (i : \mathsf{nat}, x)); \mathsf{out}(d, (i, x))$ where S' is the following process.

```
in$(c, (x_a, x_v))$;
in$(d, (i : \mathsf{nat}, x))$;
get $VoterTbl(= x_a)$ in
   event $\mathsf{InTbl}(i, x_a)$; out$(d, (i + 1, x))$
else
   event $\mathsf{NotInTbl}(i, x_a)$;
   event $\mathsf{InTbl}(i + 1, x_a)$;
   insert $VoterTbl(x_a)$;
   out$(c, HasVoted(x_a, x_v))$;
   out$(d, (i + 1, x))$
```

▶

We now express the fact that our transformation does not interfere with the execution of the process.

Lemma 9. *Let* (E, P) *be a valid configuration. Let tbl be a locked table w.r.t. d in* (E, P)*. For all correspondence query* $F \rightsquigarrow \phi$,

$$E, P \models F \rightsquigarrow \phi \quad \textit{iff} \quad E, [P]^{tbl,d}_{lock} \models F \rightsquigarrow \phi$$

Thanks to our events we can now express the fact that a else branch of a lookup operation within a round can only be executed if the elements satisfying the expression D were not already inserted in the table.

Lemma 10. *Let* (E, P) *be a valid configuration. Let tbl be a locked table w.r.t. d in* (E, P)*. Let* $\mathsf{NotInTbl}_{tbl,pat}$ *be a defined event where* $[N_1, \dots, N_m]$ *are the fixed terms of pat with* $pat = C[N_1, \dots, N_m]$.

For all $i, j \in \mathbb{N}$*, for all terms* $M_1, \dots, M_n, L_1, \dots, L_m$*, for all traces* $\mathsf{tr} = E, [P]^d_{count} \rightarrow^* E', \mathcal{S}', \mathcal{P}', \Phi'$*, if* tr *executes* $\mathsf{event}(\mathsf{NotInTbl}_{tbl,pat}(j, L_1, \dots, L_m))$ *and* $\mathsf{event}(\mathsf{InTbl}_{tbl}(i, M_1, \dots, M_n))$ *and there exists* σ *such that:*

- σ *is closing for* $C[L_1, \dots, L_m]$
- $C[L_1, \dots, L_m]\sigma = (M_1, \dots, M_n)$

then $j < i$.

Proof sketch. Since $\mathsf{NotInTbl}_{tbl,pat}(j, L_1, \dots, L_m)$ is executed by a concrete trace, we know that $L_1, \dots, L_m$ are ground and corresponds to an instantiation of the fixed terms $N_1, \dots, N_m$ of the pattern pat. Moreover, by definition of our transformation and by definition of the semantics of the rule TBLELSE in Figure 4, we know that $\mathsf{NotInTbl}_{tbl,pat}(j, L_1, \dots, L_m)$ can only be executed if no instantiation of the pattern variables in $pat = (pat_1, \dots, pat_n)$ should yield a term that was already inserted in the table. Since we assume that $C[L_1, \dots, L_m]\sigma = (M_1, \dots, M_n)$ and $\mathsf{event}(\mathsf{InTbl}_{tbl}(i, M_1, \dots, M_n))$ as executed, we deduce that $(M_1, \dots, M_n)$ cannot have been inserted before the event $\mathsf{NotInTbl}_{tbl,pat}(j, L_1, \dots, L_m)$ is executed. Therefore $j < i$. □

As for previous transformation, we prevent ProVerif from yielding false attacks on locked table by amending the target query with the negation of the property stated in lemma 10.

Theorem 7. *Let* (E, P) *be a valid configuration. Let tbl be a locked table w.r.t. d in* (E, P)*. Let* $pat_k = C_k[N_{1,k}, \dots, N_{m_k,k}]$ *with* $N_{1,k}, \dots, N_{m_k,k}$ *the fixed terms of* pat_k *and* $k = 1 \dots \ell$ *be the patterns used in the defined events* $\mathsf{NotInTbl}_{tbl,pat_1}, \dots, \mathsf{NotInTbl}_{tbl,pat_n}$*. Let* $F \rightsquigarrow \phi$ *be a correspondence query. Let* $\phi_{tbl,k}$ *be the following formula*

$$\begin{aligned} &\mathsf{event}(\mathsf{InTbl}_{tbl}(i, x_1, \dots, x_n))\ \&\&\\ &\mathsf{event}(\mathsf{NotInTbl}_{tbl,pat_k}(j, y_{1,k}, \dots, y_{m_k,k}))\ \&\&\\ &C_k[y_1, \dots, y_{m_k,k}] = (x_1, \dots, x_n)\ \&\&\ i \leq j \end{aligned}$$

We have:

$$E, P \models F \rightsquigarrow \phi \textit{ iff } E, [P]^{tbl,d}_{lock} \models F \rightsquigarrow \phi \vee \textstyle\bigvee_{k=1}^{\ell} \phi_{tbl,k}$$

Example 21. Coming back to example 11, the new query ProVerif will try to prove the query $\mathsf{event}(HasVoted(x, y)) \wedge \mathsf{event}(HasVoted(x, z)) \rightsquigarrow y = z \vee \phi_{\mathsf{table}}$ with $\phi_{\mathsf{table}} = event(\mathsf{InTbl}(i, t))\ \&\&\ event(\mathsf{NotInTbl}(j, t))\ \&\&\ i \leq j$. Note that ProVerif is not able to directly prove this property. However, since we added a counter in the cell d, we can apply on top of $[P]^{tbl,d}_{lock}$ the transformation for counters obtaining a new process $P' = !\ S' \mid \mathsf{new}\ a; out(d, a) \mid !\ in(d, x); out(d, x)$ where S' is as follows:

```
in$(c, (x_a, x_v))$;
new $st$ : stamp;
in$(d, (i : \mathsf{nat}, x))$;
event $\mathsf{Counter}(st, i)$;
get $VoterTbl(= x_a)$ in
   event $\mathsf{InTbl}(i, x_a)$; out$(d, (i + 1, x))$
...
```

Since we can apply theorems 4 and 7, we know that proving that $\mathsf{event}(HasVoted(x, y)) \wedge \mathsf{event}(HasVoted(x, z)) \rightsquigarrow y = z$ holds for P is equivalent to proving that $\mathsf{event}(HasVoted(x, y)) \wedge \mathsf{event}(HasVoted(x, z)) \rightsquigarrow y = z \vee \phi_{\mathsf{table}} \vee \phi_{\mathsf{count}}$ holds for P'. ▶

This last example shows that individually, our transformation may not be enough to prevent ProVerif from yielding a false attack but a combination of transformation might.

Appendix D. Attacks

Attacks on Key Registration [13]. Key registration is a simple protocol where Alice may revoke her public key pk by sending a new public key pk' and signing her request with her "old" key. The server encrypts its confirmation message with Alice's new public key. Then Alice can safely lose her old key sk.

$$\begin{array}{lll} A & \to S: & \mathsf{sign}(sk,(new,A,pk')) \\ S & \to A: & \mathsf{aenc}(pk',confirm) \\ A & \to S: & sk \end{array}$$

"An attacker succeeds in breaking the protocol when she discovers a secret key that is still registered to the server" [13].

The attack is depicted below.

$$\begin{array}{lll} A & \to S: & \mathsf{sign}(sk,(new,A,pk')) \\ S & \to A: & \mathsf{aenc}(pk',confirm) \\ A & \to S: & sk \end{array}$$

The attacker learns pk and retrieves pk' from the first signature.

$$\begin{array}{lll} A & \to I(S): & \mathsf{sign}(sk',(new,A,pk'')) \\ I(S) & \to A: & \mathsf{aenc}(pk'',confirm) \\ A & \to I(S): & sk' \end{array}$$

The attacker learns sk' while pk' is still registered to the server. We need two sessions as, surprisingly, the model of [13] assumes the initial public key pk to be secret.

This attack was not detected by the authors because SetPi actually assesses that the protocol is secure while it is not. The attack (and the bug in the tool) has been reported and acknowledged by the authors of [13].

Attacks on mobile EMV [15]. Since the mobile EMV protocol is complex, we assume the reader to have its description, in particular Figure 3 of [15]. The attack relies on the fact that two MACs are used for two distinct purposes:

- A key $\mathsf{k} = \mathsf{MAC}(\mathsf{K}_{\mathsf{pay}}, \mathsf{s})$ is needed to open the EMV token (used to make a payment).
- A value $\mathsf{T}_{\mathsf{val}} = \mathsf{MAC}(\mathsf{K}_{\mathsf{pay}}, \mathsf{M}_{\mathsf{ID}}, \mathsf{price}, \mathsf{s})$ is built to make sure that the payment will be executed only for the price price and the merchant M_{ID}, validated by the user.

However, the value s is not stored by the secure device (trusted enclave) but recorded and passed by the mobile application. Therefore the attack works as follows. The (untrusted) mobile application first observes one normal session between the user and the (possibly trusted) merchant. The attacker retrieves $\mathsf{k} = \mathsf{MAC}(\mathsf{K}_{\mathsf{pay}}, \mathsf{s})$ and $\mathsf{T}_{\mathsf{val}} = \mathsf{MAC}(\mathsf{K}_{\mathsf{pay}}, \mathsf{M}_{\mathsf{ID}}, \mathsf{price}, \mathsf{s})$ and does not proceed with the payment. Instead, the mobile application triggers the user to validate a second payment (e.g. pretending that the first payment did not go through). The user validates again a transaction for some merchant M_{ID} and some price price (M_{ID} and price may be the same than the initial ones, or different). However, the mobile application provides $\mathsf{s}' = \mathsf{M}_{\mathsf{ID}}^{\mathsf{att}}, \mathsf{price}^{\mathsf{att}}, \mathsf{s}$ instead of a valid nonce. This way, it obtains $\mathsf{k}' = \mathsf{MAC}(\mathsf{K}_{\mathsf{pay}}, \mathsf{M}_{\mathsf{ID}}^{\mathsf{att}}, \mathsf{price}^{\mathsf{att}}, \mathsf{s})$. With the key $\mathsf{k} = \mathsf{MAC}(\mathsf{K}_{\mathsf{pay}}, \mathsf{s})$ and the value $\mathsf{MAC}(\mathsf{K}_{\mathsf{pay}}, \mathsf{M}_{\mathsf{ID}}^{\mathsf{att}}, \mathsf{price}^{\mathsf{att}}, \mathsf{s})$, it can proceed with a payment of price $\mathsf{price}^{\mathsf{att}}$ for merchant $\mathsf{M}_{\mathsf{ID}}^{\mathsf{att}}$, entirely controlled by the attacker.

This attack was not detected by the authors because the message format in their model prevents the attack. Indeed, MAC is unary function, and the key k is written $\mathsf{k} = \mathsf{MAC}((\mathsf{s}, \mathsf{K}_{\mathsf{pay}}))$ while the value $\mathsf{T}_{\mathsf{val}}$ is written $\mathsf{MAC}((\mathsf{M}_{\mathsf{ID}}, (\mathsf{price}, (\mathsf{s}, \mathsf{K}_{\mathsf{pay}}))))$. But a different format (pairs done left first instead of right first) would enable the attack. Again, the attack has been reported and acknowledged by the authors of [15].