StatVerif: Verification of Stateful Processes

Myrto Arapinis, Eike Ritter and Mark D. Ryan
School of Computer Science, University of Birmingham, UK

***Abstract*—We present StatVerif, which is an extension the ProVerif process calculus with constructs for explicit state, in order to be able to reason about protocols that manipulate global state. Global state is required by protocols used in hardware devices (such as smart cards and the TPM), as well as by protocols involving databases that store persistent information. We provide the operational semantics of StatVerif. We extend the ProVerif compiler to a compiler for StatVerif: it takes processes written in the extended process language, and produces Horn clauses. Our compilation is carefully engineered to avoid many false attacks. We prove the correctness of the StatVerif compiler. We illustrate our method on two examples: a small hardware security device, and a contract signing protocol. We are able to prove their desired properties automatically.**

I. Introduction

Motivation: Agents that engage in security protocols necessarily involve a notion of state. For example, a protocol may require an agent A to send a certain message, to receive a response, and then to send another message that depends on the response. In this case, A needs to maintain some state information so that it knows which step of the protocol is the next one. This notion of state is local within a session.

Sometimes, there is also a requirement to maintain longer-term global state. This state is not local to a session: if one session updates the state, then it is updated for other sessions too. There seems to be two broad classes of protocols where global state is relevant:

- Protocols involving security device interfaces. This includes smartcards, stateful RFID chips, the *trusted platform module* (TPM), and secure co-processors such as the IBM 4758. Such devices maintain data including keys, and also metadata about keys, including whether a key is loaded, valid, or revoked. They may also have special registers for recording state information, such as monotonic counters, or the platform configuration registers of the TPM. They may allow other data to be saved on the device, such as the identity of a stateful RFID tag, that affects its future behaviour.
- Protocols involving databases, such as protocols for RFID tags (where a database holds information about the status of tags), protocols for websites (e.g., where a database holds status about transactions, and browsers hold cookies), and key servers (where a database records the status of keys). It also includes specialised protocols such as fair exchange protocols and contract signing protocols, where a trusted party maintains the status of the exchange in a database.

This notion of global mutable state poses a problem for existing protocol verification techniques, because those techniques often make abstractions that will introduce false attacks when state is considered. We show this in the running example, below. This has been noted before, e.g., by Herzog [1], Mödersheim [2], Guttman [3] and Delaune *et al.* [4]. For example, the ProVerif protocol analyser [5] models an ever-increasing set of derivable facts representing attacker knowledge, and is not able to associate those facts with the states in which they arose. For this reason, the tool typically reports many false attacks. The AVISPA tool [6] aims to handle mutable state via its OFMC, CL-AtSe and SATMC backends. However, the first two of these require concrete bounds to be given on the number of sessions and fresh nonces. SATMC can avoid this restriction in principle [7], but as we mentioned in [4], SATMC performed poorly in our experiments due to the relatively large message length, a known weakness of SATMC.

To address this problem, Mödersheim [2] has developed a framework which takes global state into account. He introduces a low-level language called AIF, which extends the IF language of AVISPA by adding sets. The framework is based on a strong abstraction that identifies all objects that are in exactly the same sets. This method appears to work well, although he does not provide all the details. Since the method is tightly coupled with a particular abstraction, the scope of its applicability is not very clear. The author mentions the restrictions of the low-level and implementation-focused language, and points out the desirability of a high-level language for protocol designers. Guttman has also addressed the problem, by extending the strand space model with a notion of state [3]. However, this extended model does not currently have tool support. In a similar direction to ours, Delaune/Kremer/Ryan/Steel have coded stateful aspects of the TPM directly in Horn clauses [8].

Our approach and contributions: We present StatVerif, which is an extension the ProVerif process language with constructs that allow one to directly model global mutable state. This approach allows us to build on ProVerif's existing successes. More precisely,

- We extend the ProVerif process calculus with explicit state, including assignments, and provide its operational semantics.

- We extend the ProVerif compiler, which takes processes written in the process language, and produces Horn clauses. Our translation is carefully engineered to avoid the false attacks mentioned above.
- We prove the correctness of the extended compiler; that is, we show that attacks are not lost. Therefore, a security proof at the clause level implies a proof at the process calculus level.
- We illustrate our method on two examples: a small hardware security device, and a contract signing protocol. We are able to prove their desired properties automatically.

Full details of our code for the examples are available on the web[1].

Running example: The following example allows us to explain our results more fully. Consider a hardware device whose behaviour can be configured by the user. The device generates a public key k. A user Alice may use software to encrypt pairs (x, y) of secrets with k, resulting in $\{(x, y)\}_k$. Later, she can give the device and a set $\{\{(x_1, y_1)\}_k, \ldots, \{(x_n, y_n)\}_k\}$ of such ciphertexts to another user Bob. The device allows Bob to configure it as "left" or "right". If Bob chooses to configure it as "left", then after doing so he can use the device to obtain any of the first components x_i of the pairs. If he configures it with "right", then he gets to have the second components y_i. Such a device might, for example, be used to allow a customer to choose between vouchers for a music website, or vouchers for a social networking site, but not both.

We model such a device in our stateful language as the following process:

```
1   new s; new k;
2     out(c, pk(k)) | [s ↦ init] |
3     ( ! lock; in(c, x); read s as y;
4         if y = init then s := x; unlock ) |
5     ( ! in(c, x); read s as y; let z = adec(k, x) in
6         let xl = projl(z) in
7         let xr = projr(z) in
8         (if y = left then out(c, xl) |
9          if y = right then out(c, xr)) )
```

In line 2, we declare a cell s with initial value init. In lines 3-4, we allow the user to provide any value to configure the device (the useful values are left and right); this sets the cell s. In lines 5-9, we allow the user to provide a ciphertext; in return, the user will receive the left or right component, according to the configuration. Notice that the device, once configured left or right, cannot be configured again.

Details of the constructs including lock will be explained later. We assume the usual equational theory for public key encryption. The desired property is that, given a ciphertext $\{(x, y)\}_k$, the attacker cannot obtain the pair (x, y). This property is easily automatically proved using our techniques.

It is interesting to note that it is possible to convert such a process into a semantically-equivalent pure ProVerif process. The cell s could be represented by a private channel, that stores the configuration value. The subprocesses that read the value s would instead input it from this private channel. However, although the private channel process is semantically equivalent to our process, ProVerif is not able to prove that it satisfies the desired property because, as mentioned, ProVerif's abstractions would introduce false attacks. In particular, once init is placed on the private channel, it remains forever available. Therefore, in the private channel model, the device allows itself to be configured and reconfigured at will. The user can obtain (x, y) by configuring it first as left and then as right. Our technique does not introduce for states the abstractions that ProVerif uses for private channels.

Outline: We give some necessary background about ProVerif and Horn clauses in section II. In section III, we detail the syntax and semantics of our stateful language, and show how it is translated into clauses in section IV. We also prove the correctness of the translation. In section V, we treat the case studies.

II. Background

A. ProVerif process language

We start from the ProVerif process language introduced in [9], which we recall in the first half of Figure 1 (up to and including the conditional process). This language is similar to the applied pi calculus [10], and is designed to model security protocols. It allows processes to send terms built over a signature including names and variables. These terms model the messages that are exchanged during a protocol. Cryptographic operations are modelled by reductions such as

$$\begin{aligned} \mathsf{sdec}(x, \mathsf{senc}(x, y)) &\to y \\ \mathsf{adec}(x, \mathsf{aenc}(\mathsf{pk}(x), y)) &\to y \\ \mathsf{check\_getmsg}(\mathsf{pk}(x), \mathsf{sign}(x, y)) &\to y \\ \mathsf{checkpcs}(\mathsf{pk}(x), \mathsf{pcs}(x, \mathsf{pk}(y), z)) &\to \mathsf{ok} \\ \mathsf{convert}(y, \mathsf{pcs}(x, \mathsf{pk}(y), z)) &\to \mathsf{sign}(x, z) \end{aligned}$$

In this example, we consider a signature that has the constructors senc, aenc, pk, pcs, sign and ok. The functions sdec, checkpcs, check_getmsg and convert are destructors. The symbols x, y, z are variables. The first three reductions model symmetric and asymmetric encryption and digital signing of messages in the usual way. The last two model *private contract signatures* that are used in our example in section V.

Processes $P, Q, R, \ldots$ are constructed as follows. The process 0 is the empty process which does nothing. In new $a; P$, we restrict the name a in P; this can be used to

[1] markryan.eu/research/projects/StatVerif/

model that a is a fresh random number or key. The process $\mathsf{in}(M, x); P$ models the input of a term on a channel M; the term is then substituted for x in process P. The process $\mathsf{out}(M, N)$ outputs a term N on a channel M. The parallel composition $P \mid Q$ models processes P and Q running concurrently. The conditional if $M = N$ then P else Q behaves as P when M and N are equal modulo the reductions, and behaves as Q otherwise. $!P$ is the replication of P, modelling an unbounded number of copies of the process P. ProVerif can automatically check security properties, while assuming that an arbitrary adversary process is run in parallel.

Example 1: The following process P models a simple mutual authentication protocol in which a party A engages with another party, say B, by sending to B a signed and encrypted session key k:

$$\begin{aligned} P &= \mathsf{new}\ sk_A; \mathsf{new}\ sk_B; \mathsf{new}\ s; \\ &\quad (\mathsf{out}(c, \mathsf{pk}(sk_A)) \mid \mathsf{out}(c, \mathsf{pk}(sk_B)) \mid !P_A \mid !P_B) \\ P_A &= \mathsf{in}(c, x_{pk}); \mathsf{new}\ k; \\ &\quad \mathsf{out}(c, \mathsf{aenc}(x_{pk}, \mathsf{sign}(sk_A, k))); \\ &\quad \mathsf{in}(c, z); Q \\ P_B &= \mathsf{in}(c, y); \mathsf{let}\ y' = \mathsf{adec}(sk_B, y)\ \mathsf{in} \\ &\quad \mathsf{let}\ y_k = \mathsf{check\_getmsg}(\mathsf{pk}(sk_A), y')\ \mathsf{in} \\ &\quad \mathsf{out}(c, \mathsf{senc}(y_k, s)) \end{aligned}$$

B responds by sending a secret s encrypted with k. Of course, this protocol is known not to be secure; an attacker can send its own public key to A, and use the session key it receives to start a parallel session with B. Then then the attacker will be able to decrypt B's secret.

B. Horn clauses

The ProVerif tool works by translating processes written in the process language into clauses of a particular form. Such a clause

$$H_1, H_2, \ldots, H_n \rightarrow C$$

is a conjunction of hypotheses and a conclusion. The hypotheses H_i and the conclusion C are called facts, and are built by applying predicate symbols to terms. ProVerif uses the two predicates attacker and message. The fact $\mathsf{attacker}(N)$ means that the attacker can learn the value N. The fact $\mathsf{message}(M, N)$ means that the message N has appeared on the channel M.

In the clause language of ProVerif, terms are formed from variables and names, and by application of function symbols. Names are distinguished syntactically by the fact that they are followed by square brackets $[\ldots]$; function symbols are followed by round brackets $(\ldots)$; and variables are not followed by brackets. To handle generation of new names by a process, such names in the clause representation are parametrised by the inputs that have occurred before the new name is generated. The new name k in the running example above is generated after the input of xpk; therefore, since there may be different k's for different xpk's, the k becomes parametrised by xpk, and is written $k[xpk]$. The running example process P above is translated into the following clauses:

Clauses corresponding to the process:

$$\begin{aligned} &\mathsf{message}(c[], \mathsf{pk}(skA[])) \\ &\mathsf{message}(c[], \mathsf{pk}(skB[])) \\ &\mathsf{message}(c[], xpk) \rightarrow \\ &\qquad \mathsf{message}(c[], \mathsf{aenc}(xpk, \mathsf{sign}(skA[], k[xpk]))) \\ &\mathsf{message}(c[], \mathsf{aenc}(\mathsf{pk}(skB[]), \mathsf{sign}(skA[], y))) \rightarrow \\ &\qquad\qquad \mathsf{message}(c[], \mathsf{senc}(y, s[])) \end{aligned}$$

The first two clauses correspond to the output of the public keys in the main process P. The third one corresponds to the attacker's ability to supply any data xpk to P_A, and in return obtain $\mathsf{aenc}(xpk, \mathsf{sign}(skA[], k[xpk]))$. The last one corresponds to a similar service offered by P_B.

Clauses corresponding to the attacker's ability to apply function symbols: These clauses depend only on the equational theory and not on the specific process.

$$\begin{aligned} &\mathsf{attacker}(ok()) \\ &\mathsf{attacker}(v) \rightarrow \mathsf{attacker}(pk(v)) \\ &\mathsf{attacker}(v_1) \wedge \mathsf{attacker}(v_2) \rightarrow \mathsf{attacker}(\mathsf{sign}(v_1, v_2)) \\ &\mathsf{attacker}(v_1) \wedge \mathsf{attacker}(v_2) \rightarrow \mathsf{attacker}(\mathsf{senc}(v_1, v_2)) \\ &\mathsf{attacker}(v_1) \wedge \mathsf{attacker}(v_2) \rightarrow \mathsf{attacker}(\mathsf{aenc}(v_1, v_2)) \\ &\mathsf{attacker}(v_1) \wedge \mathsf{attacker}(v_2) \wedge \mathsf{attacker}(v_3) \rightarrow \\ &\qquad\qquad \mathsf{attacker}(\mathsf{pcs}(v_1, v_2, v_3)) \end{aligned}$$

Clauses corresponding to the term reductions:

$$\begin{aligned} &\mathsf{attacker}(\mathsf{pk}(x)) \wedge \mathsf{attacker}(\mathsf{sign}(x, y)) \rightarrow \mathsf{attacker}(y) \\ &\mathsf{attacker}(x) \wedge \mathsf{attacker}(\mathsf{aenc}(\mathsf{pk}(x), y)) \rightarrow \mathsf{attacker}(y) \\ &\mathsf{attacker}(x) \wedge \mathsf{attacker}(\mathsf{senc}(x, y)) \rightarrow \mathsf{attacker}(y) \end{aligned}$$

Clauses corresponding to the attacker's ability to send and receive messages: These clauses are the same for all protocols and all equational theories.

$$\begin{aligned} &\mathsf{message}(v_1, v_2) \wedge \mathsf{attacker}(v_1) \rightarrow \mathsf{attacker}(v_2) \\ &\mathsf{attacker}(v_1) \wedge \mathsf{attacker}(v_2) \rightarrow \mathsf{message}(v_1, v_2) \\ &\mathsf{attacker}(c[]) \end{aligned}$$

The first of these three clause says that if the attacker has a channel name v_1 then he may read messages sent on it. The second one is a dual; he may also send messages on v_1. Lastly, we stipulate that the channel c is public.

Returning to the authentication protocol example, one can check that the fact $\mathsf{attacker}(s[])$ can be derived from the set of clauses. Indeed, this derivation corresponds to a real attack, and the protocol is not secure.

C. Translation and correctness

Details of the translation from the process language to clauses may be found in [9]. We do not detail it here, although we extend it to handle states in section IV. The translation has an important correctness property: it does not omit attacks. More precisely, if the process allows the attacker to obtain a secret value, say s, then $\mathsf{attacker}(s)$ can be derived from the clauses that correspond to the process. ProVerif uses a clause resolution strategy that is complete. Therefore, if ProVerif declares that $\mathsf{attacker}(s)$ is not derivable, it is indeed not derivable. In that case, thanks to the correctness property of the translation, we can conclude that the attacker is indeed not capable of obtaining the secret s from the process.

III. The StatVerif language

We extend the process language of [9] recalled in section II with some constructs to handle global state.

A. Syntax and informal semantics

To model global state, StatVerif adds the following new processes:

- $[s \mapsto M]$, which represents a cell s that has the initial value M;
- $\mathsf{read}\ s\ \mathsf{as}\ x;\ P$, which reads the value of the cell s (calling it x), and then continues as P;
- $s := M;\ P$ which assigns M to s and then continues as P;
- $\mathsf{lock};\ P$. This process begins a *locked section*; that means that the process takes exclusive access to the state cells, and continues as P.
- $\mathsf{unlock};\ P$, which releases the lock on the state cells, continuing as P.

The full syntax of StatVerif is given in Figure 1, subject to the following additional restrictions:

- $[s \mapsto M]$ may occur only once for a given cell name s, and may occur only within the scope of new, a parallel and a replication. It may not be in the scope of an input, output, conditional, let, assignment, lock, or unlock.
- In every branch of the syntax tree, every lock must be followed by precisely one corresponding unlock.
- In $\mathsf{lock};\ P$, the part of the process P that occurs before the next unlock, if any, may not include parallel, replication, or lock.

In particular, these conditions imply that lock . . . unlock may not be nested. The purpose of lock . . . unlock is to allow manipulations of global state to proceed without interference from other concurrent processes. Obviously, such interactions would lead to unwanted results. For example, in our security device, the lock and unlock in lines 3 and 4 ensure that the device cannot move from the "left" configuration to the "right" configuration. If we didn't have the lock and unlock, it would be possible to have the following execution.

$M, N ::=$	terms
x, y, z	variables
a, b, c, k, s	names
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
0	nil
$\mathsf{out}(M, N);\ P$	output
$\mathsf{in}(M, x);\ P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a;\ P$	restriction
$\mathsf{let}\ x = g(M_1, \ldots, M_n)$ in P else Q	destructor application
$\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
$[s \mapsto M]$	state
$\mathsf{read}\ s\ \mathsf{as}\ x;\ P$	read
$s := M;\ P$	assign
$\mathsf{lock};\ P$	begin locked section
$\mathsf{unlock};\ P$	end locked section

Figure 1. The StatVerif calculus. The terms and the processes up to and including the conditional are from [9]. The remaining processes are our additions. Some syntax restrictions are mentioned in the text.

Consider two parallel sessions of the device. The first inputs left on channel c and reads the state s. Then the second session inputs right on channel c and reads the state s. At this moment both sessions consider the device to be in state init. It would thus be possible for the first session to update s to left and then for the second one to update s to right, *i.e.* the state s goes from init to left and then to right. In other words, without the locked section it is possible to reconfigure the device at will.

We sometimes omit the else branch of if-statements. If the subprocess $\mathsf{if}\ M = N\ \mathsf{then}\ P$ occurs in the scope of a lock, then it is an abbreviation for $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ \mathsf{unlock}; 0$. Otherwise, it is an abbreviation for $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ 0$.

The binders of our language are $\mathsf{new}\ a$, $\mathsf{in}(c, x)$, $\mathsf{let}\ x = g(M_1, \ldots, M_n)$, and $\mathsf{read}\ s\ \mathsf{as}\ x$. As usual, we denote $\mathrm{bn}(P)$ and $\mathrm{bv}(P)$ the set of bounded names and variables of P respectively, and $\mathrm{fn}(P)$ and $\mathrm{fv}(P)$ the set of free names and variables of P respectively.

B. Operational semantics

A semantic configuration for StatVerif is a tuple $(\mathcal{E}, \mathcal{S}, \mathcal{P})$, where the environment $\mathcal{E}$ is a finite set of names, $\mathcal{S}$ is a function mapping state cells to their values, $\mathcal{P}$ is a finite multiset of pairs of the form (P, μ) where P is a process and μ is a boolean indicating whether P has locked the state cells. In a configuration $(\mathcal{E}, \mathcal{S}, \mathcal{P})$, at most one of the μ is true. The environment $\mathcal{E}$ must

contain at least the free names of $\mathcal{S}$ and $\mathcal{P}$. The configuration $(\{a_1, \ldots, a_n\}, \mathcal{S}, \{(P_1, \mathsf{false}), \ldots, (P_m, \mathsf{false})\})$ intuitively corresponds to the process $\mathsf{new}\ a_1, \ldots, a_n; ([s \mapsto \mathcal{S}(s) \mid s \in \mathrm{dom}(\mathcal{S})] \mid P_1 \mid \ldots, \mid P_m)$.

The semantics of StatVerif is defined by a reduction relation $\rightarrow$ on semantic configurations, shown in Figure 2. It is an extension of the semantics of [9, Fig. 3]. Notice that it preserves the invariant that at most one of the processes in $\mathcal{P}$ can have locked the memory. The mode is set to true by lock, and only one process can be in mode true. So if a process has set its mode to true the other running processes cannot access the memory until the corresponding unlock. Assignment and read s as x update and read the store $\mathcal{S}$ in the expected way.

C. Definition of secrecy

An adversary A is represented as a process of our calculus. He has some initial knowledge of a finite set of names $Init$ with at least one channel name $attch \in Init$. A is said to be an $Init$-adversary if A is a closed process and $\mathrm{fn}(A) \subseteq Init$.

Informally, a protocol preserves the secrecy of a message M from $Init$ if when run in parallel with any $Init$-adversary A, M cannot be output on a public channel.

Definition 1: Let P be a closed process, $Init$ a finite set of names such that $attch \in Init$, M a message. P preserves the secrecy of M against $Init$ if for any $Init$-adversary A, there exists no trace of the form:

$$((Init \cup \mathrm{fn}(P) \cup \mathrm{fn}(M)), \emptyset, \{(P \mid A, \mathsf{false})\}) \rightarrow^* (\mathcal{E}, \mathcal{S}, \mathcal{Q} \cup \{(\mathsf{out}(attch, M); Q, \mu)\}).$$

Here we consider that M is secret if it is secret in all reachable states. We could have extended this definition to express secrecy relative to a particular state, or to states of a certain form, but for simplicity, and since we don't need to in what follows, we didn't include it here.

IV. Translation to Clauses

A. The translation

The translation generates clauses from a StatVerif process. The clauses are built around the predicates attacker and message with the following meanings:

- $\mathsf{attacker}(\tilde{M}, N)$ means that there is a reachable state of the process in which the state variables $\tilde{s}$ have the values $\tilde{M}$, and in that state the attacker knows the value N; this binary predicate is also used in [8].
- $\mathsf{message}(\tilde{M}, N, K)$ means that there is a reachable state of the process in which the state variables $\tilde{s}$ have the values $\tilde{M}$, and in that state the value K is available on channel N.

1) Clauses corresponding to the process: Our translation only applies to StatVerif processes of the form:

$$\mathsf{new}\ \tilde{m}; ([s_1 \mapsto M_1] \mid \cdots \mid [s_n \mapsto M_n] \mid P)$$

such that

- P has no $[s \mapsto M]$ in it. (Of course, P may have reads and assignments.)
- each name and variable is bound at most once in P; and each name and variable in P is either bound or free but not both.

The tuple $\tilde{m}$ contains cell names and ordinary names. Some of the $s_1, \ldots, s_n$ may be in $\tilde{m}$, and others not.

Note that any process with a finite number of cell names can be converted into one of the prescribed form. While the restriction of finite number of cells may appear to be severe, we will see in section V that it is still possible to correctly abstract processes with an unbounded number of memory cells by processes with a finite number of memory cells, and thus use our technique and then ProVerif to verify them.

Let ρ_0 be the function $\{a \mapsto a[],\ s_i \mapsto s_i[] \mid a \in \mathrm{fn}(P), 1 \leq i \leq n\}$ and let $\phi_0 = (\rho_0(M_1), \ldots, \rho_0(M_n))$. The process above is translated into the union of the following sets of clauses:

- $[\![P]\!]\ \rho_0\ \mathsf{true}\ [\,]\ \phi_0\ \mathsf{false}$ where the function $[\![\cdot]\!]\rho H \ell \phi \mu$ is given in Figure 3;
- Some other clauses given in the next two subsections.

The rules of Figure 3 generalise the ones given in [9, §5.2.2].

The StatVerif compiler that performs the translation maintains the variables ρ, H, ℓ, ϕ and T, which have the following purposes:

- ρ is a function mapping names and variables of the process language to names and variables of the clause language.
- H is a conjunction of formulas used to accumulate the hypotheses of clauses as they are constructed.
- ℓ accumulates the set of variables that have been input so far by the thread being processed. This set is used to parameterise the Skolem names that represent values created by "new".
- ϕ is a tuple of terms $(M_1, \ldots, M_n)$ representing the last known values of the state variables in the thread under consideration. This information is used to generate clauses when a locked section is being processed.
- μ is a boolean indicating whether the currently processed thread is in a locked section.

We explain the rules for the translation given in Figure 3.

- The rules for processing 0, parallel, "new", "let" and "if" are similar to those of [9], with obvious changes for our more general setting.
- The rule for processing $!P$ is simpler than [9], since we don't treat correspondence properties for now.
- For an *input*, we record in ρ and ℓ the variable that is input, and add the appropriate hypothesis to H. If the lock is true, then we can be sure that the state used in the hypothesis is the current state of the thread. If the lock is false, then another subprocess could have

$$
\begin{array}{rcl}
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P\mid 0,\mathsf{false})\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P,\mathsf{false})\}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(!P,\mathsf{false})\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(!P\mid P,\mathsf{false})\}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P\mid Q,\mathsf{false})\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P,\mathsf{false}),(Q,\mathsf{false})\}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\mathsf{new}\ a;P,\mu)\}) & \to & (\mathcal{E}\cup\{a'\},\mathcal{S},\mathcal{P}\cup\{(P\{a'/a\},\mu)\})\ \textbf{if}\ a'\ \text{fresh} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\text{let } x=g(M_1,\ldots,M_n)\text{ in } P\text{ else } Q,\mu)\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P\{M'/x\},\mu)\}) \\
 & & \textbf{if}\ g(M_1,\ldots,M_n)\to M' \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\text{let } x=g(M_1,\ldots,M_n)\text{ in } P\text{ else } Q,\mu)\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(Q,\mu)\}) \\
 & & \textbf{if}\ \nexists M',\ g(M_1,\ldots,M_n)\to M' \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\text{if } M=M\text{ then } P\text{ else } Q,\mu)\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P,\mu)\}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\text{if } M=N\text{ then } P\text{ else } Q,\mu)\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(Q,\mu)\}) \qquad \textbf{if}\ M\neq N \\
 & & \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\mathsf{out}(M,N);P,\mu_1),(\mathsf{in}(M,x);Q,\mu_2)\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P,\mu_1),(Q\{N/x\},\mu_2))
\end{array}
$$

$$
\begin{array}{rcl}
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{([s\mapsto M],\mathsf{false})\}) & \to & (\mathcal{E},\mathcal{S}\cup\{s\mapsto M\},\mathcal{P}) \\
 & & \textbf{if}\ s\in\mathcal{E}\ \textbf{and}\ s\notin\mathrm{dom}(\mathcal{S}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\mathsf{lock};P,\mathsf{false})\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P,\mathsf{true})\}) \\
 & & \textbf{if}\ \forall(Q,\mu)\in\mathcal{P},\mu=\mathsf{false} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\mathsf{unlock};P,\mathsf{true})\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P,\mathsf{false})\}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\mathsf{read}\ s\ \mathsf{as}\ x;P,\mu)\}) & \to & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P\{\mathcal{S}(s)/x\},\mu)\})\ \textbf{if}\ s\in\mathrm{dom}(\mathcal{S}) \\
 & & \textbf{and}\ \forall(Q,\mu')\in\mathcal{P},\mu'=\mathsf{false} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(s:=M;P,\mu)\}) & \to & (\mathcal{E},\mathcal{S}[s\mapsto M],\mathcal{P}\cup\{(P,\mu)\})\ \ \textbf{if}\ s\in\mathrm{dom}(\mathcal{S}) \\
 & & \textbf{and}\ \forall(Q,\mu')\in\mathcal{P},\mu'=\mathsf{false}
\end{array}
$$

Figure 2. The semantics of StatVerif. $\mathcal{E}$ is a set of names. $\mathcal{S}$ is a function from state cells to their current values. $\mathcal{P}$ is the set of processes to be run, and P is the process currently running. μ is a boolean indicating whether the current process is in a locked section.

changed the state, so we have to allow an arbitrary state.

- An *output* generates a clause that reveals the output on the channel, using the hypotheses accumulated so far.
- For a lock, we initialise the assumed state with arbitrary information (that may be the result of a parallel subprocess), and set the lock μ to true.
- An unlock unsets the lock.
- The assignment $s := M$ updates the current state ϕ. As for an input, if the lock is true then other values in ϕ that were not assigned to are preserved. If the lock is false, they are not preserved, because another parallel process could have overwritten them. Additionally, we generate the "inheritance" clauses that transport attacker knowledge and message availability on channels from the state before the assignment to the one after it.
- The read process assigns the cell that is read to the stipulated variable. As in the input and assignment cases, the state is propagated when the lock is true, whereas an arbitrary state is assumed when the lock is false. The hypothesis added to H ensures that the s_i that was read is from a reachable state. Inside a locked section, read updates the record ρ of variable definitions.

2) Clauses corresponding to mutability of public state: If a state cell name s is known to the attacker, then the attacker is able to read and write values from and to the cell. For each $i \in \{1, 2, \ldots, n\}$, we have the following clauses for reading:

$$\mathsf{attacker}((x_1,\ldots,x_n),s_i[]) \to \mathsf{attacker}((x_1,\ldots,x_n),x_i)$$

and the following ones for writing:

$$\begin{array}{l}
\mathsf{attacker}((x_1,\ldots,x_n),s_i[])\ \wedge\ \mathsf{attacker}((x_1,\ldots,x_n),y)\ \wedge \\
\quad \mathsf{message}((x_1,\ldots,x_n),xc,xm)\to \\
\qquad \mathsf{message}((x_1,\ldots,x_{i-1},y,x_{i+1},\ldots,x_n),xc,xm) \\
\mathsf{attacker}((x_1,\ldots,x_n),s_i[])\ \wedge\ \mathsf{attacker}((x_1,\ldots,x_n),y)\ \wedge \\
\quad \mathsf{attacker}((x_1,\ldots,x_n),xm)\to \\
\qquad \mathsf{attacker}((x_1,\ldots,x_{i-1},y,x_{i+1},\ldots,x_n),xm)
\end{array}$$

3) Other clauses: Additionally, we have clauses corresponding to the function symbols and the term reductions for the signature at hand. These are the stateful counterparts of the clauses used by ProVerif:
For each constructor f of arity n,

$$\mathsf{attacker}(xs,x_1)\wedge\cdots\wedge\mathsf{attacker}(xs,x_n)\to \mathsf{attacker}(xs,f(x_1,\ldots,x_n)).$$

For each constructor g, for each rewrite rule $g(M_1,\ldots,M_n)\to M$, let xs be a fresh variable,

$$\mathsf{attacker}(xs,M_1)\wedge\cdots\wedge\mathsf{attacker}(xs,M_n)\to\mathsf{attacker}(xs,M)$$

$$
\begin{array}{lcl}
[\![0]\!]\rho H\ell\phi\mu & = & \emptyset \\
[\![Q_1 \mid Q_2]\!]\rho H\ell\phi\mathsf{false} & = & [\![Q_1]\!]\rho H\ell\phi\mathsf{false} \cup [\![Q_2]\!]\rho H\ell\phi\mathsf{false} \\
[\![!Q]\!]\rho H\ell\phi\mathsf{false} & = & [\![Q]\!]\rho H\ell\phi\mathsf{false} \\
[\![\mathsf{new}\ a; Q]\!]\rho H\ell\phi\mu & = & \begin{cases} [\![Q]\!](\rho \cup \{a \mapsto a[\ell]\})H\ell\phi\mu & \text{if } a \in \mathsf{bn}(P) \\ [\![Q]\!](\rho \cup \{a \mapsto attn[]\})H\ell\phi\mu & \text{otherwise} \end{cases} \\
[\![\mathsf{in}(M,x); Q]\!]\rho H\ell\phi\mathsf{false} & = & [\![Q]\!](\rho \cup \{x \mapsto x, vs_1 \mapsto vs_1, \ldots, vs_n \mapsto vs_n\})\ H'\ (x :: \ell)\ \phi\ \mathsf{false} \\
 & & \quad \textbf{where}\ \phi_0 = (vs_1, \ldots, vs_n), \textbf{with}\ vs_1, \ldots, vs_n\ \text{fresh} \\
 & & \quad \textbf{and}\ H' = H \wedge \mathsf{message}(\phi_0, \rho(M), x) \\
[\![\mathsf{in}(M,x); Q]\!]\rho H\ell\phi\mathsf{true} & = & [\![Q]\!](\rho \cup \{x \mapsto x\})(H \wedge \mathsf{message}(\phi, \rho(M), x))(x :: \ell)\phi\mathsf{true} \\
[\![\mathsf{out}(M,N); Q]\!]\rho H\ell\phi\mu & = & \{H \Rightarrow \mathsf{message}(\phi, \rho(M), \rho(N))\} \cup [\![Q]\!]\rho H\ell\phi\mu \\
[\![\mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in} & & \\
\quad Q_1\ \mathsf{else}\ Q_2]\!]\rho H\ell\phi\mu & = & \bigcup \Big\{ [\![Q_1]\!]((\rho\sigma) \cup \{x \mapsto p'\sigma'\})(H\sigma)(\ell\sigma)(\phi\sigma)\mu \mid \\
 & & \qquad g(p'_1, \ldots, p'_n) \to p' \in def(g)\ \text{and}\ (\sigma, \sigma')\ \text{mgus and} \\
 & & \qquad M_1\rho\sigma = p'_1\sigma', \ldots, M_n\rho\sigma = p'_n\sigma' \Big\} \cup [\![Q_2]\!]\rho H\ell\phi\mu \\
[\![\mathsf{if}\ M = N\ \mathsf{then}\ Q_1 & & \\
\quad \mathsf{else}\ Q_2]\!]\rho H\ell\phi\mu & = & [\![Q_1]\!](\rho\sigma)(H\sigma)(\ell\sigma)(\phi\sigma)\mu \cup [\![Q_2]\!]\rho H\ell\phi\mu \quad \textbf{where}\ \sigma = mgu(\rho(M), \rho(N)) \\
[\![\mathsf{lock}; Q]\!]\rho H\ell\phi\mathsf{false} & = & [\![Q]\!](\rho \cup \{vs_1 \mapsto vs_1, \ldots, vs_n \mapsto vs_n\})H\ell\phi_0\mathsf{true} \\
 & & \quad \textbf{where}\ \phi_0 = (vs_1, \ldots, vs_n),\ \textbf{with}\ vs_1, \ldots, vs_n\ \text{fresh} \\
[\![\mathsf{unlock}; Q]\!]\rho H\ell\phi\mathsf{true} & = & [\![Q]\!]\rho H\ell\phi\mathsf{false} \\
[\![s_i := M; Q]\!]\rho H\ell\phi\mathsf{false} & = & [\![Q]\!](\rho \cup \{vs_1 \mapsto vs_1, \ldots, vs_n \mapsto vs_n, vc \mapsto vc, vm \mapsto vm\})H\ell\phi\mathsf{false} \\
 & & \cup\{H \wedge \mathsf{message}(\phi_0, vc, vm) \Rightarrow \mathsf{message}(\phi_1, vc, vm)\} \\
 & & \cup\{H \wedge \mathsf{attacker}(\phi_0, vm) \Rightarrow \mathsf{attacker}(\phi_1, vm)\} \\
 & & \quad \textbf{where}\ \phi_0 = (vs_1, \ldots, vs_{i-1}, vs_i, vs_{i+1}, \ldots, vs_n), \\
 & & \quad \textbf{and}\ \phi_1 = (vs_1, \ldots, vs_{i-1}, \rho(M), vs_{i+1}, \ldots, vs_n) \\
 & & \quad \textbf{with}\ vs_1, \ldots, vs_n, vc, vm\ \text{fresh} \\
[\![s_i := M; Q]\!]\rho H\ell\phi\mathsf{true} & = & [\![Q]\!](\rho \cup \{vc \mapsto vc, vm \mapsto vm\})H\ell\phi'\mathsf{true} \\
 & & \cup\{H \wedge \mathsf{message}(\phi, vc, vm) \Rightarrow \mathsf{message}(\phi', vc, vm)\} \\
 & & \cup\{H \wedge \mathsf{attacker}(\phi, vm) \Rightarrow \mathsf{attacker}(\phi', vm)\} \\
 & & \quad \textbf{where}\ \phi = (M_1, \ldots, M_{i-1}, M_i, M_{i+1}, \ldots, M_n), \\
 & & \quad \textbf{and}\ \phi' = (M_1, \ldots, M_{i-1}, \rho(M), M_{i+1}, \ldots, M_n), \\
 & & \quad \textbf{and}\ vc, vm\ \text{fresh} \\
[\![\mathsf{read}\ s_i\ \mathsf{as}\ x; Q]\!]\rho H\ell\phi\mathsf{false} & = & [\![Q]\!](\rho \cup \{x \mapsto vs_i, vs_1 \mapsto vs_1, \ldots, vs_i \mapsto vs_i, \ldots, vs_n \mapsto vs_n, \\
 & & \quad vc \mapsto vc, vm \mapsto vm\})(H \wedge \mathsf{message}(\phi_0, vc, vm))\ell\phi\mathsf{false} \\
 & & \quad \textbf{where}\ \phi_0 = (vs_1, \ldots, vs_i, \ldots, vs_n), \\
 & & \quad \textbf{with}\ vs_1, \ldots, vs_i, \ldots, vs_n, vc, vm\ \text{fresh} \\
[\![\mathsf{read}\ s_i\ \mathsf{as}\ x; Q]\!]\rho H\ell\phi\mathsf{true} & = & [\![Q]\!](\rho \cup \{x \mapsto M_i, vc \mapsto vc, vm \mapsto vm\})(H \wedge \mathsf{message}(\phi, vc, vm))\ell\phi\mathsf{true} \\
 & & \quad \textbf{where}\ \phi = (M_1, \ldots, M_i, \ldots, M_n)\ \textbf{and}\ vc, vm\ \text{fresh}
\end{array}
$$

Figure 3. The rules for translating a stateful process into clauses.

The attacker is also able to read and write on channels that it knows, and the stateful analogues of those clauses are:

$$\mathsf{message}(xs, v_1, v_2) \wedge \mathsf{attacker}(xs, v_1) \rightarrow \mathsf{attacker}(xs, v_2)$$
$$\mathsf{attacker}(xs, v_1) \wedge \mathsf{attacker}(xs, v_2) \rightarrow \mathsf{message}(xs, v_1, v_2)$$

Finally, the attacker knows

- all the free names of P, *i.e.* we have the clause $\mathsf{attacker}(\rho_0(\phi_0), n[])$ for every $n \in \mathrm{fn}(P)$,
- as well as the channel $attch$ and a name $attn$ he has generated on his own, *i.e.* we have the clauses $\mathsf{attacker}(\rho_0(\phi_0), attch[])$ and $\mathsf{attacker}(\rho_0(\phi_0), attn[])$,

where ρ_0 and ϕ_0 are as defined in section IV-A1.

B. Correctness

Let $P' = \mathsf{new}\ \tilde{m}([s_1 \mapsto M_1] \mid \cdots \mid [s_n \mapsto M_n] \mid P)$ be a closed process and A an $Init$-adversary *s.t.* $attch \in Init$. Without loss of generality, we can assume that the free cell names in A are included in the free cell names of P, and that the set of bounded cell names of A is empty. The reason is that any other cell name of the intruder can be equivalently encoded using channel names as described by Milner.

Instrumented operational semantics

In what follows, we will need to identify different instances of a $\mathsf{new}\ a$ arising when $\mathsf{new}\ a$ is in the scope of a !. We will consider instrumented semantic configurations $(\mathtt{E}, \mathcal{S}, \mathcal{P})$ where $\mathtt{E}$ will now be a mapping from names to StatVerif terms ($\mathtt{E}$ records for each name a' the $\mathsf{new}\ a$ in P it is an instance of), $\mathcal{S}$ is as before a function from cell names to terms, and $\mathcal{P}$ is a set of tuples (Q, ℓ, μ) where we will record in ℓ the list of $M_1, \ldots, M_n$ that where previously input to reach this configuration.

We adapt the semantics to an *instrumented operational semantics* which is defined by a reduction relation on instrumented configurations. Except for the reduction rules for new and comm all the other rules of Figure 2 give rise to a corresponding instrumented rule where $\mathtt{E}$ and the ℓ's are unchanged. And

- The reduction rule for communication becomes the following

$$(\mathtt{E}, \mathcal{S}, \mathcal{P} \cup \{(\mathsf{out}(M, N); Q_1, \ell_1, \mu_1), (\mathsf{in}(M, x); Q_2, \ell_2, \mu_2)\}) \rightarrow (\mathtt{E}, \mathcal{S}, \mathcal{P} \cup \{(Q_1, \ell_1, \mu_1), (Q_2\{N/x\}, (N :: \ell_2), \mu_2)\})$$

  which records N in ℓ_2.
- The reduction rule for name generation is replaced by the two following ones

$$(\mathtt{E}, \mathcal{S}, \mathcal{P} \cup \{(\mathsf{new}\ a; Q, \ell, \mu)\}) \rightarrow (\mathtt{E} \cup \{a' \mapsto a[\mathtt{E}(\ell)]\}, \mathcal{S}, \mathcal{P} \cup \{(Q\{a'/a\}, \ell, \mu)\}) \quad \text{if } a \in \mathrm{bn}(P) \text{ and } a' \text{ fresh}$$

  which records that a' is an instance of $\mathsf{new}\ a$. ℓ is used to distinguish two instances of $\mathsf{new}\ a$ on the basis of the previous inputs.

$$(\mathtt{E}, \mathcal{S}, \mathcal{P} \cup \{(\mathsf{new}\ a; Q, \ell, \mu)\}) \rightarrow (\mathtt{E} \cup \{a' \mapsto attn[]\}, \mathcal{S}, \mathcal{P} \cup \{(Q\{a'/a\}, \ell, \mu)\}) \quad \text{if } a \notin \mathrm{bn}(P) \text{ and } a' \text{ fresh}$$

  which records that a' is an name of the attacker A.

It is easy to see that the instrumented semantics allows exactly the same traces as the original semantics, only adding annotations on the origin of each name.

Proof of correctness

Let $\mathcal{C}_P$ be the set of clauses generated by StatVerif when applied to P, and $\mathcal{F}_P$ the set of closed facts derivable from $\mathcal{C}_P$. Let $\mathcal{S}_0 = \{s_1 \mapsto M_1, \ldots, s_n \mapsto M_n\}$. Let $\mathtt{E}_0$ be the environment such that

- $\mathrm{fn}(P) \cup \mathrm{cells}(P) \cup \mathrm{fn}(A) = \mathrm{dom}(\mathtt{E}_0)$,
- $\mathtt{E}_0(a) = a[]$ for all $a \in \mathrm{fn}(P) \cup \mathrm{cells}(P) \cup \{attch\}$,
- $\mathtt{E}_0(a) = attn[]$ for all $a \in \mathrm{fn}(A) \setminus \{attch\}$.

Let $S = \{s_1 \mapsto K_1, \ldots, s_n \mapsto K_n\}$ be a state. $\overline{S}$ denotes the ordered representation of S, defined as $\overline{S} = (K_1, \ldots, K_n)$.

We will say that a state R is a predecessor of the state S, denoted $R \leq S$ if:

$$\begin{array}{lll} & \mathsf{attacker}(\overline{R}, attch[]) \in \mathcal{F}_P & \\ \wedge & \forall M, N \quad \mathsf{message}(\overline{R}, M, N) \in \mathcal{F}_P & \Rightarrow \\ & & \mathsf{message}(\overline{S}, M, N) \in \mathcal{F}_P \\ \wedge & \forall M \quad \mathsf{attacker}(\overline{R}, M) \in \mathcal{F}_P & \Rightarrow \\ & & \mathsf{attacker}(\overline{S}, M) \in \mathcal{F}_P \end{array}$$

The proof uses the type system to capture invariants of processes. This type system is defined by the rules of Figure 4 (an extended version of the type system of [9], [11]).

A process Q is well typed *w.r.t.* the environment $\mathtt{E}$, the state S, the list of StatVerif terms ℓ, and the mode μ, if $(\mathtt{E}, S, \ell, \mu) \vdash P$ can be derived from the rules and axiom of Figure 4.

Before proceeding with the proof of our main theorem, we need to establish some properties of our typing system.

Lemma 1 (Typability of A): $(\mathtt{E}_0, \mathtt{E}_0(\mathcal{S}_0), [], \mathsf{false}) \vdash A$

Proof sketch: Let B be a subprocess of A. Let $\mathtt{E}$ be an environment, S a state, ℓ a list of terms. We first prove that if *(i)* $\mathtt{E}_0 \subseteq \mathtt{E}$; and *(ii)* $\mathtt{E}_0(\mathcal{S}_0) \leq S$; and *(iii)* for all $a \in \mathrm{fn}(B)$, $\mathsf{attacker}(\overline{S}, \mathtt{E}(a)) \in \mathcal{F}_P$; and *(iv)* for all $x \in \mathrm{fv}(B)$, $\mathsf{attacker}(\overline{S}, \mathtt{E}(x)) \in \mathcal{F}_P$, then

$$(\mathtt{E}, S, \ell, \mu) \vdash B$$

where $\mu = \mathsf{true}$ if B is under a lock in A and $\mu = \mathsf{false}$ otherwise. The proof is an induction on the depth of B.

To conclude the proof of Lemma 1 we need to show that A, $\mathtt{E}_0$, $\mathtt{E}_0(\mathcal{S}_0)$, and $[]$ satisfy conditions (i)-(iv). *(i)* $\mathtt{E}_0 \subseteq \mathtt{E}_0$. *(ii)* $\mathtt{E}_0(\mathcal{S}_0) \leq \mathtt{E}_0(\mathcal{S}_0)$. *(iii)* By construction, $\forall a \in \mathrm{fn}(A)$

- If $a = attch$, then $\mathtt{E}_0(a) = attch[]$, and by construction $\mathsf{attacker}(\overline{\mathtt{E}_0(\mathcal{S}_0)}, attch[]) \in \mathcal{C}_P$.

$$\frac{\mathsf{message}(\overline{S}, \mathtt{E}(M), \mathtt{E}(N)) \in \mathcal{F}_P \qquad (\mathtt{E}, S, \ell, \mu) \vdash Q}{(\mathtt{E}, S, \ell, \mu) \vdash \mathsf{out}(M, N); Q}\ \tau_{out}$$

$$\frac{\forall N\ \mathsf{message}(\overline{S}, \mathtt{E}(M), N) \in \mathcal{F}_P\ \Rightarrow\ (\mathtt{E} \cup \{x \mapsto N\}, S, (N :: \ell), \mathsf{true}) \vdash Q}{(\mathtt{E}, S, \ell, \mathsf{true}) \vdash \mathsf{in}(M, x); Q}\ \tau_{inT}$$

$$\frac{\forall T\ \forall N\ (S \leq T \wedge \mathsf{message}(\overline{T}, \mathtt{E}(M), N) \in \mathcal{F}_P)\ \Rightarrow\ (\mathtt{E} \cup \{x \mapsto N\}, T, (N :: \ell), \mathsf{false}) \vdash Q}{(\mathtt{E}, S, \ell, \mathsf{false}) \vdash \mathsf{in}(M, x); Q}\ \tau_{inF}$$

$$\frac{}{(\mathtt{E}, S, \ell, \mu) \vdash 0}\ \tau_{nil} \qquad \frac{(\mathtt{E}, S, \ell, \mathsf{false}) \vdash Q \qquad (\mathtt{E}, S, \ell, \mathsf{false}) \vdash Q'}{(\mathtt{E}, S, \ell, \mathsf{false}) \vdash Q \mid Q'}\ \tau_{par} \qquad \frac{(\mathtt{E}, S, \ell, \mathsf{false}) \vdash Q}{(\mathtt{E}, S, \ell, \mathsf{false}) \vdash !Q}\ \tau_{repl}$$

$$\frac{(\mathtt{E}(M) = \mathtt{E}(N) \Rightarrow (\mathtt{E}, S, \ell, \mu) \vdash Q) \qquad (\mathtt{E}, S, \ell, \mu) \vdash Q'}{(\mathtt{E}, S, \ell, \mu) \vdash \text{if } M = N \text{ then } Q \text{ else } Q'}\ \tau_{if}$$

$$\frac{a \in \mathsf{bn}(P)\ \Rightarrow\ (\mathtt{E} \cup \{a \mapsto a[\ell]\}, S, \ell, \mu) \vdash Q}{(\mathtt{E}, S, \ell, \mu) \vdash \mathsf{new}\ a; Q}\ \tau_{newP} \qquad \frac{a \in \mathsf{bn}(A)\ \Rightarrow\ (\mathtt{E} \cup \{a' \mapsto attn[]\}, S, \ell, \mu) \vdash Q}{(\mathtt{E}, S, \ell, \mu) \vdash \mathsf{new}\ a; Q}\ \tau_{newA}$$

$$\frac{\forall M\ (g(\mathtt{E}(M_1), \ldots, \mathtt{E}(M_n)) \rightarrow M)\ \Rightarrow ((\mathtt{E} \cup \{x \mapsto M\}, S, \ell, \mu) \vdash Q\ \wedge\ (\mathtt{E}, S, \ell, \mu) \vdash Q')}{(\mathtt{E}, S, \ell, \mu) \vdash \text{let } x = g(M_1, \ldots, M_n) \text{ in } Q \text{ else } Q'}\ \tau_{let}$$

$$\frac{(\mathtt{E} \cup \{x \mapsto S(s_i)\}, S, \ell, \mathsf{true}) \vdash Q}{(\mathtt{E}, S, \ell, \mathsf{true}) \vdash \mathsf{read}\ s_i\ \mathsf{as}\ x; Q}\ \tau_{readT} \qquad \frac{\forall T\ S \leq T\ \Rightarrow (\mathtt{E} \cup \{x \mapsto T(s_i)\}, T, \ell, \mathsf{false}) \vdash Q}{(\mathtt{E}, S, \ell, \mathsf{false}) \vdash \mathsf{read}\ s_i\ \mathsf{as}\ x; Q}\ \tau_{readF}$$

$$\frac{S \leq S[s_i \mapsto \mathtt{E}(M)] \qquad (\mathtt{E}, S[s_i \mapsto \mathtt{E}(M)], \ell, \mathsf{true}) \vdash Q}{(\mathtt{E}, S, \ell, \mathsf{true}) \vdash s_i := M; Q}\ \tau_{writeT}$$

$$\frac{\forall T\ S \leq T\ \Rightarrow (T \leq T[s_i \mapsto \mathtt{E}(M)]\ \wedge\ (\mathtt{E}, T[s_i \mapsto \mathtt{E}(M)], \ell, \mathsf{false}) \vdash Q)}{(\mathtt{E}, S, \ell, \mathsf{false}) \vdash s_i := M; Q}\ \tau_{writeF}$$

$$\frac{\forall T\ S \leq T\ \Rightarrow\ (\mathtt{E}, T, \ell, \mathsf{true}) \vdash Q}{(\mathtt{E}, S, \ell, \mathsf{false}) \vdash \mathsf{lock}; Q}\ \tau_{b\_atom} \qquad \frac{(\mathtt{E}, S, \ell, \mathsf{false}) \vdash Q}{(\mathtt{E}, S, \ell, \mathsf{true}) \vdash \mathsf{unlock}; Q}\ \tau_{e\_atom}$$

Figure 4. Typing system for correctness proof

- If $a \neq attch$, then $\mathtt{E}_0(a) = attn[]$, and by construction $\mathsf{attacker}(\overline{\mathtt{E}_0(\mathcal{S}_0)}, attn[]) \in \mathcal{C}_P$.

Thus $\forall a \in \mathsf{fn}(A)\ \mathsf{attacker}(\overline{\mathtt{E}_0(\mathcal{S}_0)}, \mathtt{E}_0(a)) \in \mathcal{F}_P$. *(iv)* A is an $Init$-adversary, so it is a closed process. Thus $\mathrm{fv}(A) = \emptyset$.

We can thus apply the induction result we just established to conclude that $(\mathtt{E}_0, \mathtt{E}_0(\mathcal{S}_0), [], \mathsf{false}) \vdash A$. ■

Lemma 2 (Typability of P): $(\mathtt{E}_0, \mathtt{E}_0(\mathcal{S}_0), [], \mathsf{false}) \vdash P$.

Proof sketch: Let Q be a subprocess of P, and σ, ρ, H, ℓ, ϕ, and μ. We first prove that if

(i) ρ binds all the free names and variables of Q,
(ii) σ is a closed substitution,
(iii) $\mu = \mathsf{true}$ if Q is under a lock in P, and $\mu = \mathsf{false}$ otherwise,
(iv) $\mathcal{C}_P \supseteq [\![Q]\!]\rho H \ell \overline{\phi} \mu$,
(v) for all $\mathsf{message}(\xi, M, N) \in H$, $\mathsf{message}(\xi\sigma, M\sigma, N\sigma)$ can be derived from $\mathcal{C}_P$,
(vi) $\mathsf{attacker}(\overline{\phi\sigma}, attch[])$.

Then,

$$(\rho\sigma, \phi\sigma, \ell\sigma, \mu) \vdash Q$$

The proof is an induction over the depth of Q.

Now, let $\rho = \mathtt{E}_0$, σ *s.t.* $\mathrm{dom}(\sigma) = \emptyset$, $H = \mathsf{true}$, $\ell = []$, $\phi = \mathtt{E}_0(\mathcal{S}_0)$ and $\mu = \mathsf{false}$. *(i)* Since by hypotheses $\mathrm{fv}(P) = \emptyset$ and $\mathrm{fn}(P) \subseteq \mathrm{dom}(\mathtt{E}_0)$ by construction, ρ binds the free names and variables of P. *(ii)* σ is a closed substitution. *(iii)* P is not under a lock in P, thus $\mu = \mathsf{false}$ satisfies condition (iii). *(iv)* By definition $\mathcal{C}_P = [\![P]\!]\rho H \ell \overline{\phi} \mu$. *(v)* H being empty, $H\sigma$ can be derived from $\mathcal{C}_P$. *(vi)* By construction, $\mathsf{attacker}(\overline{\mathtt{E}_0(\mathcal{S}_0)}, attch[]) \in \mathcal{C}_P$. So in particular $\mathsf{attacker}(\overline{\phi\sigma}, attch[]) \in \mathcal{F}_P$. Thus, P, ρ, σ, H, ℓ, ϕ and μ satisfy the conditions of our induction result according to which $(\mathtt{E}_0, \mathtt{E}_0(\mathcal{S}_0), [], \mathsf{false}) \vdash P$. ■

Lemma 3 (Subject reduction): Let $(\mathtt{E}, \mathcal{S}, \mathcal{Q}) \rightarrow (F, \mathcal{T}, \mathcal{R})$ be a valid instrumented transition, and no $[s \mapsto M]$ occurs in $\mathcal{Q}$. If $(\mathtt{E}, \mathtt{E}(\mathcal{S}), \mathtt{E}(\imath), \mu) \vdash Q$ for all $(Q, \imath, \mu) \in \mathcal{Q}$, then $(F, F(\mathcal{T}), F(\jmath), \nu) \vdash R$ for all $(R, \jmath, \nu) \in \mathcal{R}$.

Proof sketch: The proof is done by case analysis on the rule that fired the transition $(\mathtt{E}, \mathcal{S}, \mathcal{Q}) \rightarrow (F, \mathcal{T}, \mathcal{R})$. ■

Theorem 1: Consider the instrumented trace

$$tr = (\mathtt{E}_0, \mathcal{S}_0, \{(P \mid A, [], \mathsf{false})\}) \rightarrow^* (\mathtt{E}, \mathcal{S}, \mathcal{Q} \cup \{(Q, \ell, \mu)\})$$

If $Q = \mathsf{out}(M, N); Q'$ then $\mathsf{message}(\overline{\mathtt{E}(\mathcal{S})}, \mathtt{E}(M), \mathtt{E}(N)) \in \mathcal{F}_P$ and $\mathsf{attacker}(\overline{\mathtt{E}(\mathcal{S})}, attch[]) \in \mathcal{F}_P$.

Proof: Consider the instrumented trace

$$\begin{aligned} tr \;&=\; (\mathtt{E}_0, \mathcal{S}_0, \mathcal{Q}_0) \;=\; (\mathtt{E}_0, \mathcal{S}_0, \{(P \mid A, [], \mathsf{false})\}) \\ &\rightarrow\; (\mathtt{E}_1, \mathcal{S}_1, \mathcal{Q}_1) \\ &\rightarrow\; \ldots \\ &\rightarrow\; (\mathtt{E}_n, \mathcal{S}_n, \mathcal{Q}_n) \;=\; (\mathtt{E}, \mathcal{S}, \mathcal{Q} \cup \{(Q, \ell, \mu)\}) \end{aligned}$$

We prove by induction on i, that for all $i \in \{0, \ldots, n\}$

$$\mathsf{attacker}(\overline{\mathtt{E}_i(\mathcal{S}_i)}, attch[]) \in \mathcal{F}_P \quad \text{and} \quad \forall (R, \imath, \nu) \in \mathcal{Q}_i \;\; (\mathtt{E}_i, \mathtt{E}_i(\mathcal{S}_i), \mathtt{E}_i(\imath), \nu) \vdash R$$

Base case ($i = 0$)**.** By definition of the StatVerif compiler $\mathsf{attacker}(\overline{\mathtt{E}_0(\mathcal{S}_0)}, attch[]) \in \mathcal{C}_P$ and thus $\mathsf{attacker}(\overline{\mathtt{E}_0(\mathcal{S}_0)}, attch[]) \in \mathcal{F}_P$. Moreover, by Lemma 1 we have $(\mathtt{E}_0, \mathtt{E}_0(\mathcal{S}_0), [], \mathsf{false}) \vdash A$, and by Lemma 2 we have $(\mathtt{E}_0, \mathtt{E}_0(\mathcal{S}_0), [], \mathsf{false}) \vdash P$. Thus, according to the typing rule τ_{par}, $(\mathtt{E}_0, \mathtt{E}_0(\mathcal{S}_0), [], \mathsf{false}) \vdash A \mid P$.

Inductive case ($i = n$)**.** By inductive hypothesis we know that the last transition satisfies the hypotheses of Lemma 3 according to which $\mathsf{attacker}(\mathtt{E}_n(\mathcal{S}_n), \mathtt{E}_n(attch)) \in \mathcal{F}_P$, and $(\mathtt{E}_n, \mathtt{E}_n(\mathcal{S}_n), \mathtt{E}_n(\imath), \nu) \vdash R$ for all $(R, \imath, \nu) \in \mathcal{Q}$.

This concludes our induction and gives us $(\mathtt{E}, \mathtt{E}(\mathcal{S}), \mathtt{E}(\ell), \mu) \vdash Q$ and $\mathsf{attacker}(\overline{\mathtt{E}(\mathcal{S})}, attch[]) \in \mathcal{F}_P$. But then, by rule τ_{out} we know that $\mathsf{message}(\overline{\mathtt{E}(\mathcal{S})}, \mathtt{E}(M), \mathtt{E}(N)) \in \mathcal{F}_P$. ■

Corollary 1 (Correctness w.r.t. *secrecy):* Let M a message. If P doesn't preserve the secrecy of M against $Init$, *i.e.* there exist an instrumented trace $tr = (\mathtt{E}_0, \mathcal{S}_0, \{(P \mid A, [], \mathsf{false})\} \rightarrow^* \{(\mathtt{E}, \mathcal{S}, \mathcal{Q} \cup \{(Q, \ell, \mathsf{false})\}$, *s.t.* $Q = \mathsf{out}(attch, M); Q'$. Then the fact $\mathsf{attacker}(\overline{\mathtt{E}(\mathcal{S})}, \mathtt{E}(M)) \in \mathcal{F}_P$.

Proof: This follows immediately from Theorem 1. ■

V. Case studies

To illustrate our method, we describe two case studies in detail. We show the processes in the StatVerif language, and use our rules to translate them to clauses. We use ProVerif to reason with the clauses and to verify the security properties.

A. A simple security device

1) Description and process: Consider again the hardware device introduced in the introduction. We take the process representing the device, together with the process representing Alice who creates the ciphertexts:

```
1   let device =
2     new s; [s ↦ init] |
3       out(c, pk(k)) |
4       ( ! lock; in(c, x); read s as y;
5           if y = init then s := x; unlock ) |
6       ( ! lock; in(c, x); read s as y;
7           let z = adec(k, x) in
8           let zl = projl(z) in let zr = projr(z) in
9           if y = left then (out(c, zl); unlock)
10          else if y = right then out(c, zr); unlock)

11  let user =
12    new sl; new sr; new r;
13      out(c, aenc(pk(k), r, (sl, sr)))

14  let system = new k; device | ! user
```

Bob is the attacker. He receives the device and the ciphertexts, and chooses the messages to send to the device. We assume the term reductions:

$$\begin{aligned} \mathsf{adec}(u, \mathsf{aenc}(\mathsf{pk}(u), v, w)) &\rightarrow w \\ \mathsf{projl}((u, v)) &\rightarrow u \\ \mathsf{projr}((u, v)) &\rightarrow v \end{aligned}$$

The query is query $\mathsf{attacker}(vs, (sl[], sr[]))$

2) Clauses corresponding to the protocol: We apply the translation described in section IV. We will only show how to compute the clauses corresponding to the system process. In other words we will compute $[\![\text{system}]\!]\rho_0\,\mathsf{true}\,[\,]\phi_0\mathsf{false}$, where $\rho_0 = \{c \mapsto c[], \mathsf{left} \mapsto \mathit{left}[], \mathsf{right} \mapsto \mathit{right}[], \mathsf{init} \mapsto \mathit{init}[]\}$ and $\phi_0 = (\mathit{init}[])$.

The $\mathsf{out}(c, \mathsf{pk}(k))$ on line 3 is translated to:

$$\mathsf{message}(\mathit{init}[], c[], \mathsf{pk}(k[]))$$

The $s := x$ on line 5, with $\mathsf{in}(c, x)$ and $\mathsf{read}\ s\ \mathsf{as}\ y$ from line 4, generates:

$$\begin{aligned} &\mathsf{message}(\mathit{init}[], c[], x) \;\wedge\; \mathsf{message}(\mathit{init}[], yc, ym) \wedge \\ &\quad \mathsf{message}(\mathit{init}[], zc, zm) \rightarrow \mathsf{message}(x, zc, zm) \\ &\mathsf{message}(\mathit{init}[], c[], x) \;\wedge\; \mathsf{message}(\mathit{init}[], yc, ym) \wedge \\ &\quad \mathsf{attacker}(\mathit{init}[], zm) \rightarrow \mathsf{attacker}(x, zm) \end{aligned}$$

The $\mathsf{out}(c, zl)$ on line 9, with lines 6 and 8, is translated to:

$$\begin{aligned} &\mathsf{message}(\mathit{left}[], c[], \mathsf{aenc}(\mathsf{pk}(k[]), xr, (xsl, xsr))) \wedge \\ &\quad \mathsf{message}(\mathit{left}[], yc, ym) \rightarrow \mathsf{message}(\mathit{left}[], c[], xsl) \end{aligned}$$

The $\mathsf{out}(c, zr)$ on line 10, with lines 6 and 8, is translated to:

$$\begin{aligned} &\mathsf{message}(\mathit{right}[], c[], \mathsf{aenc}(\mathsf{pk}(k[]), xr, (xsl, xsr))) \wedge \\ &\quad \mathsf{message}(\mathit{right}[], yc, ym) \rightarrow \mathsf{message}(\mathit{right}[], c[], xsr) \end{aligned}$$

The output on line 13 is translated to:

$$\mathsf{message}(init[], c[], \mathsf{aenc}(\mathsf{pk}(k[]), r[], (sl[], sr[])))$$

3) Results of the analysis: We ran ProVerif on the clauses, together with the query, given above. ProVerif immediately concluded that the query is not satisfied (i.e., the protocol is secure). We made a few sanity checks, such as modifying the device to allow it to be configured again, and in that case ProVerif reported the valid attack as expected.

B. Contract signing protocol

A contract signing protocol allows a set of participants to exchange messages with each other in order to arrive at a state in which each of them has a pre-agreed contract signed by the others. An important property of contract signing protocols is fairness: no participant should be left in the position of having sent another participant his signature on the contract but not having received the others' signatures. To ensure fairness, a trusted party is necessary. Garay and Mackenzie [12] proposed such a protocol which, for efficiency, involves the trusted party only to resolve disputes. This protocol is based on private contract signatures. A private contract signature by A for B on m *w.r.t.* trusted party T acts as a promise by A to B to sign m.

In this section we will show how by applying our techniques to the two-party instance of the Garay and Mackenzie (GM) protocol, we automatically prove that the two-party version of this protocol satisfies fairness. To achieve this result we need no bound on the number of sessions/contracts or agents considered. In comparison, if we model the protocol by a plain ProVerif process, using private channels to model the state of the trusted party, and run ProVerif on it, then the tool reports a false attack. It reports the same false attack even if only one contract is considered.

Currently, the only other automatic proof of security of a protocol of this kind *i.e.* with participants holding global mutable state, is the one achieved by Mödersheim in his framework [2]. Again, because his language is so low level, we couldn't understand the hypotheses of his proof just by inspecting his model of the protocol. It is not clear to us how many participants, or contracts are considered.

1) Description and process: The protocol is informally described in Figure 5 and consists of four subprotocols: Main, Abort1, Resolve2 and Resolve1. Usually, contract signers try to achieve the exchange without the help of the trusted party. They first exchange their promises to sign the contract (messages m_1 and m_2), and then exchange their actual signatures of the contract (messages m_3 and m_4). If for some reason they do not succeed in completing their exchange, the signers can ask the trusted party to arbitrate, by asking it either to abort or to resolve:

1) If P_2 doesn't receive P_1's promise, he just quits.

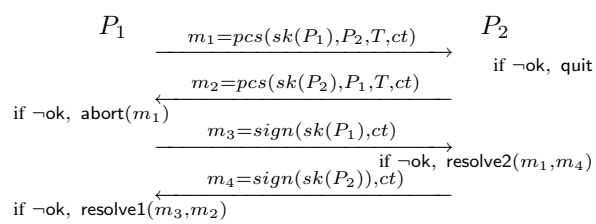

Figure 5. The GM Main protocol (see [12])

2) If P_1 doesn't receive P_2's promise, he asks the trusted party to abort. He includes his own promise in his request.
3) If P_2 (*resp.* P_1) doesn't receive P_1's (*resp.* P_2's) signature, he asks the trusted party to resolve. He includes his own signature, and P_1's (*resp.* P_2's) promise to sign the contract, in his request.

To deal with these requests, the trusted party records the following information for each contract ct:

- $status$ - indicating whether it has solved any dispute regarding ct in the past. The possible values are $init$, $aborted$, $resolved1$ and $resolved2$.
- $sigs$ - the acknowledgement of its decision, if it has made one. As we will now see, this is either its signature on the received abort request or its signature on the two contracts.

On receipt of a request, the trusted party checks whether it had to solve a dispute on the same contract in the past. If it did ($status \neq init$), it just sends the decision it had taken and stored at that time ($sigs$). If it is the first request it receives, then

- if it is an abort request including the promise $m = \mathsf{pcs}(x, y, T, ct)$, it acknowledges the request with the message $\mathsf{sign}(skT, m)$. It then updates the status of ct to $aborted$ and stores its decision $\mathsf{sign}(skT, m)$,
- if it is a resolve request including the promise $\mathsf{pcs}(x, y, T, ct)$ and the signature $\mathsf{sign}(sk(y), ct)$, it converts $x's$ promise into a valid signature $\mathsf{sign}(sk(x), ct)$ and replies with the message $\mathsf{sign}(skT, \mathsf{sign}(sk(x), ct), \mathsf{sign}(sk(y), ct))$. In other words, it sends to the plaintiff the signature corresponding to the promise. It also stores its reply in sig and updates the $status$ of ct to $resolved1$ or $resolved2$, according to which party sent the request.

The following process represents the trusted party:

```
1  let T = new skT; (out(c, pk(skT)) | !C)
2  let C = new s; new ct;
3          [s ↦ (init, init)] |
4              out(c, ct); in(c, xpk1); in(c, xpk2);
5              ( ! Abort1 | ! Resolve2 | ! Resolve1)
```

where Abort1, Resolve2 and Resolve1 are the subprocesses modelling the trusted party's behaviour upon an abort or

resolve request. After having published its public key (line 1), the trusted party can start handling contracts ($!C$). As we just discussed, for each contract it needs to create a new memory cell s, which it initialises with $(\mathsf{init}, \mathsf{init})$ (lines 2 and 3), to record information regarding the particular contract. It can then start replying to requests regarding this contract (line 5). The details of the subprocesses Abort1, Resolve2, and Resolve1, together with the details of the reduction rules modelling the cryptographic primitives, are given in appendix A.

As we explained in this section's introduction, it is important that the trusted party is fair to both parties. In other words, we want the following:

- if the participant P_1 has first contacted the trusted party and requested for contract ct an abort which was granted, then P_2 cannot obtain from the trusted party P_1's signature (*i.e.* he cannot receive the signature of P_1 on contract ct signed with the trusted party's secret key);
- if the participant P_1 (*resp.* P_2) has first contacted the trusted party and requested for contract ct a resolve which was granted, then P_2 (*resp.* P_1) cannot obtain from the trusted party an abort confirmation (*i.e.* the promise of P_1 (*resp.* P_2) on contract ct signed with the trusted party's secret key);

These three properties can be combined and stated as a secrecy property, and can be formalised as

$$\mathsf{query}\ \mathsf{attacker}(xs, (abortC, resolveC))$$

where $abortC = \mathsf{sign}(sk(T), ((ct, (pk(P_1), pk(P_2))), \mathsf{sign}(sk(P_2), (ct, (pk(P_1), pk(P_2))))))$ is the abort acknowledgement, and $resolveC = \mathsf{sign}(sk(T), (\mathsf{sign}(sk(P_1), ct), \mathsf{sign}(sk(P_2), ct)))$ is the resolve acknowledgement.

Of course, there are many more properties that one would want a contract signing protocol to satisfy, but we only considered this one for the purpose of illustrating our techniques and showing that they work in non-trivial situations.

2) From unbounded number of cell names to bounded: Our translation only applies to processes with a finite number of cell names, *i.e.* with no $[s \mapsto M]$ under a replication. However, in the GM protocol, the trusted party creates two cell names for each contract. So for an unbounded number of contracts it creates an unbounded number of cell names. To prove that the GM protocol satisfies fairness using our techniques we make the following correct abstraction: the trusted party behaves according to the protocol only for a single contract ct. For this witnessing contract it creates the two cells it needs, and to any request regarding ct it replies and updates its memory according to the protocol. Thus, fairness of the protocol is proved only for ct. To requests concerning any other contract ct' it replies as if it were the first time it received any request regarding ct'.

So the process for the trusted party that we actually verify is the following:

1 let T' = new skT; $(\mathsf{out}(c, \mathsf{pk}(skT)) \mid \text{C} \mid \text{!C'})$
2 let C' = new ct'; $\mathsf{out}(c, ct')$; $\mathsf{in}(c, xpk1)$; $\mathsf{in}(c, xpk2)$;
3 (! Abort1' | ! Resolve2' | ! Resolve1')

where C is as we defined it in section V-B1 and Abort1', Resolve2', Resolve1' are like Abort1, Resolve2, Resolve1 but with no checks on the status before replying. These subprocesses are given in details in appendix B.

Proposition 1: Let $Init$ be a finite set of names. If T' satisfies fairness against $Init$, then T does too.

Proof sketch: Let $attch \in Init$ and let A be an $Init$-attacker that breaks the fairness of T.

1) In any trace of T, A cannot read or write the trusted party's memory. Indeed, the cell names held by the trusted party are never sent on any channel and are under a restriction. So we can correctly consider A to be a plain process (no cell names occurring in it).

2) Because all the conditions before the trusted party's output are removed in Abort1', Resolve2', and Resolve1', the following holds: for any trace tr of T such that

$$(\text{fn}(T) \cup \text{fn}(A), \emptyset, \{(T \mid A, \mathsf{false})\}) \rightarrow^* (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \{(\mathsf{out}(attch, M); Q, \mathsf{false})\})$$

there exists a trace tr' of T'

$$(\text{fn}(T') \cup \text{fn}(A), \emptyset, \{(T' \mid A, \mathsf{false})\}) \rightarrow^* (\mathcal{E}', \mathcal{S}', \mathcal{P}' \cup \{(\mathsf{out}(attch, M); Q', \mathsf{false})\})$$

Now, since T doesn't preserve fairness against A, there exists a trace

$$(\text{fn}(T) \cup \text{fn}(A), \emptyset, \{(T \mid A, \mathsf{false})\}) \rightarrow^* (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \{(\mathsf{out}(attch, M); Q, \mathsf{false})\})$$

with $M = (abortC, resolveC)$. But then by 2) a trace of $T' \mid A$ breaking fairness also exists. ■

3) Results of the analysis: We applied our translation to T', and ran ProVerif on the set of clauses obtained. ProVerif concluded in less than 2 minutes that the query is not satisfied; in other words, that T' satisfies fairness. Thus, according to proposition 1, the two-party instance of the GM protocol satisfies fairness in the general case. The code for this example is available on the web[2].

VI. Conclusion

We presented StatVerif, an extension of the ProVerif process calculus with constructs for explicit global state, and detailed the StatVerif compiler that takes processes written in this language and returns a corresponding set of clauses. We proved that the compiler is correct with respect to the operational semantics.

This machinery allows us naturally to write protocols that manipulate state in an intuitive high-level language. The language includes locked sections to enable sequences of

[2] markryan.eu/research/projects/StatVerif/

state manipulations to be written conveniently and correctly. We demonstrated the language on a couple of case studies. The StatVerif compiler converts processes written in the language to clauses upon which ProVerif can be run. We have engineered the compiler carefully to result in clauses which do not introduce false attacks (as would be the case if one used the natural private-channel encoding of state). Moreover, ProVerif has a good chance to terminate on the translated clauses. Typically, it will do so easily if the state space is finite. For infinite state spaces, some further abstractions are likely to be necessary. We provided the clauses resulting from the translation of the case studies. ProVerif terminates easily on those examples, and we are able to prove their desired properties automatically.

As further work, we intend to implement the StatVerif compiler, and, if appropriate, to contribute it to the ProVerif codebase. We also want to develop some further abstractions that are likely to be necessary in common situations.

Acknowledgements. We gratefully acknowledge financial support from Microsoft Corporation, and from EPSRC via the projects *Verifying Interoperability Requirements in Pervasive Systems* (EP/F033540/1) and *Analysing Security and Privacy Properties* (EP/H005501/1).

References

[1] J. Herzog, "Applying protocol analysis to security device interfaces," *IEEE Security & Privacy Magazine*, vol. 4, no. 4, pp. 84–87, July-Aug 2006.

[2] S. Mödersheim, "Abstraction by set-membership: verifying security protocols and web services with databases," in *Proc. 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, 2010, pp. 351–360.

[3] J. D. Guttman, "Fair exchange in strand spaces," *Journal of Automated Reasoning*, 2011, to appear.

[4] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel, "A formal analysis of authentication in the TPM," in *Proc. 7th International Workshop on Formal Aspects in Security and Trust (FAST'10)*, Pisa, Italy, 2010.

[5] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proc. of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*. Cape Breton, Nova Scotia, Canada: IEEE Computer Society Press, Jun. 2001, pp. 82–96.

[6] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. 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 *Proc. 17th International Conference on Computer Aided Verification (CAV'05)*, 2005, pp. 281–285.

[7] S. Fröschle and G. Steel, "Analysing PKCS#11 key management APIs with unbounded fresh data," in *Proc. Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, ser. LNCS, P. Degano and L. Viganò, Eds., vol. 5511. York, UK: Springer, 2009, pp. 92–106, to appear.

[8] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel, "Formal analysis of protocols based on TPM state registers," in *Proc. of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*. IEEE Computer Society Press, 2011.

[9] B. Blanchet, "Automatic verification of correspondences for security protocols," *Journal of Computer Security*, vol. 17, no. 4, pp. 363–434, 2009.

[10] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Proc. 28th Symposium on Principles of Programming Languages (POPL'01)*, H. R. Nielson, Ed. London, UK: ACM Press, 2001, pp. 104–115.

[11] B. Blanchet, "Automatic verification of correspondences for security protocols," *CoRR*, vol. abs/0802.3444, 2008.

[12] J. A. Garay, M. Jakobsson, and P. D. MacKenzie, "Abuse-free optimistic contract signing," in *Proceedings of the 19th Annual Cryptology Conference on Advances in Crypto*, ser. CRYPTO '99, London, UK, 1999, pp. 449–466.

Appendix A.
Contract signing: Trusted party with unbounded memory

In this section we detail the process of our language modelling the GM protocol without any restriction on the number of cell names held by the trusted party T.

```
1  let T = new skT; (out(c, pk(skT)) | !C)
2  let C =  new s; new ct;
3           [s ↦ (init, init)] |
4                out(c, ct); in(c, xpk1); in(c, xpk2);
5                ( ! Abort1 | ! Resolve2 | ! Resolve1)
```

If P_1 doesn't receive P_2's promise, he requests from T an abort by sending him a message containing the information about the contract for which he requests the resolve, and of the form:

$$\underbrace{(\underbrace{\mathsf{abort}}_{xcmd}, \underbrace{((\overbrace{ct}^{ycontract}, \overbrace{(P_1, P_2)}^{yparties}), \overbrace{\mathsf{sign}(sk(P_1), (ct, (P_1, P_2)))}^{ysig})}_{y})}_{x}$$

Upon receipt of such a command (line 7), the trusted party executes the subprocess Abort1 which consists of

- Extracting from x the parts $xcmd$, $ycontract$, $yparties$, and $ysig$ (lines 8, 10-13, 16).
- Checking that it is an abort request (line 9).
- Checking that it has a record for this contract with these participants (lines 14-15).
- Checking that the third component of x is a signature of the second (lines 17-18).

Once all these checks on the received message are done and passed, it handles the request:

- If it has already handled an abort request regarding ct, (*i.e.* $ystatus$ = aborted at line 21) then it retrieves (line 22) and replies (line 23) with its previous decision regarding this contract.
- Otherwise, if this is the first request regarding ct, (*i.e.* $ystatus$ = init at line 24), it updates the status of ct to aborted (line 25), stores (line 25) and sends (line 26) the acknowledgement $\mathsf{sign}(skT, y)$.

```
6   let Abort1 =
7     lock; in(c, x);
8     let xcmd = projl(x) in
9     if xcmd = abort then
10      let y = projr(x) in
11      let yl = projl(y) in
12      let ycontract = projl(yl) in
13      let yparties = projr(yl) in
14      if yparties = (xpk1, xpk2) then
15        if ycontract = ct then
16          let ysig = projr(y) in
17          let ym = check_getmsg(xpk1, ysig) in
18          if ym = yl then
19            read s as ys;
20            let ystatus = projl(ys) in
21            if ystatus = aborted then (
22              let ysigs = projr(ys) in
23              out(c, ysigs); unlock
24            ) else if ystatus = init then
25              s := (aborted, sign(skT, y));
26              out(c, sign(skT, y)); unlock
```

If P_2 doesn't receive P_1's signature, he asks T to resolve by sending it a message containing the information about the contract for which he requests the resolve. This message is of the form:

$$\underbrace{(\underbrace{\mathsf{resolve2}}_{xcmd}, \underbrace{(\mathsf{pcs}(skP_1, P2, T, ct)}_{ypcs1}, \underbrace{\mathsf{sign}(skP_2, \overbrace{ct}^{ycontract})}_{ysig2}))}_{x}$$

Upon receipt of such a command (line 28), the trusted party executes the subprocess Resolve2 which consists of

- Extracting from x the parts $xcmd$, $ycontract$, $ypcs1$, and $ysig2$ (lines 29, 34, 32, and 33).
- Checking that it is a resolve request from a responder (line 30).
- Checking that it has a record for this contract (line 35).
- Checking that the received promise ($ypcs1$) and the received signature ($ysig2$) concern the same contract ($ycontract$) (lines 36-38).

Once all these checks on the received message are done and passed, it handles the request:

- If it has already handled a resolve2 request regarding ct, (*i.e.* $ystatus$ = resolved2 at line 41) then it retrieves (line 42) and replies (line 43) with its previous decision regarding this contract.
- Otherwise, if this is the first request regarding ct, (*i.e.* $ystatus$ = init at line 44), it updates the status of ct to resolved2 and converts the promise $ypcs1$ into a valid signature $ysig1$ (line 45) and stores (line 46) and sends (line 47) the acknowledgement $\mathsf{sign}(skT, (ysig1, ysig2))$.

```
27  let Resolve2 =
28    lock; in(c, x);
29    let xcmd = projl(x) in
30    if xcmd = resolve2 then
31      let y = projr(x) in
32      let ypcs1 = projl(y) in
33      let ysig2 = projr(y) in
34      let ycontract = check_getmsg(xpk2, ysig2) in
35      if ycontract = ct then
36        let ycheck = checkpcs(ct, xpk1, xpk2,
37                              pk(skT), ypcs1) in
38        if ycheck = ok then
39          read s as ys;
40          let ystatus = projl(ys) in
41          if ystatus = resolved2 then (
42            let ysigs = projr(ys) in
43            out(c, ysigs); unlock
44          ) else if ystatus = init then
45            let ysig1 = convertpcs(skT, ypcs1) in
46            s := (resolved2, sign(skT, (ysig1, ysig2)));
47            out(c, sign(skT, (ysig1, ysig2))); unlock
```

If P_1 doesn't receive P_2's signature, he requests from T to resolve. Upon receipt of such a command, the trusted party executes the subprocess Resolve1 which is analogous to Resolve2 that we just described.

```
48 let Resolve1 =
49   lock; in(c, x);
50   let xcmd = projl(x) in
51   if xcmd = resolve1 then
52     let y = projr(x) in
53     let ysig1 = projl(y) in
54     let ypcs2 = projr(y) in
55     let ycontract = check_getmsg(xpk1, ysig1) in
56     if ycontract = ct then
57       let ycheck = checkpcs(ct, xpk2, xpk1,
58                             pk(skT), ypcs2) in
59       if ycheck = ok then
60         read s as ys;
61         let ystatus = projl(ys) in
62         if ystatus = resolved1 then (
63           let ysigs = projr(ys) in
64           out(c, ysigs); unlock
65         ) else if ystatus = init then
66           let ysig2 = convertpcs(skT, ypcs2) in
67           s := (resolved1, sign(skT, (ysig1, ysig2)));
68           out(c, sign(skT, (ysig1, ysig2))); unlock
```

APPENDIX B.
CONTRACT SIGNING: TRUSTED PARTY WITH BOUNDED MEMORY

In this section we detail the process of our language model that we actually verified to prove that the 2-party GM protocol satisfies fairness. As we established in Section V-B2, T' is a correct abstraction of T *w.r.t.* fairness. Note that in what follows, we took care to ensure that the primed versions do not handle our witnessing contract ct which is handled by C.

```
1 let T' = new skT; (out(c, pk(skT)) | C' | !C'')
2 let C' =  new ct; [s ↦ (init, init)] | out(c, ct);
3           Abort1 [pk(skA)/xpk1, pk(skB)/xpk2]
4           | Resolve2 [pk(skA)/xpk1, pk(skB)/xpk2]
5           | Resolve1 [pk(skA)/xpk1, pk(skB)/xpk2]
6 let C'' =  new ct'; out(c, ct');
7           in(c, xpk1); in(c, xpk2);
8           ( ! Abort1' | ! Resolve2' | ! Resolve1')
```

Abort1′ is built from Abort1 just by removing from Abort1 lines 19-25. Because Abort1′ replies to a request without checking the status of the requested contract, it will always reply with the abort acknowledgement.

```
9  let Abort1' =
10   lock; in(c, x);
11   let xcmd = projl(x) in
12   if xcmd = abort then
13     let y = projr(x) in
14     let yl = projl(y) in
15     let ycontract = projl(yl) in
16     let yparties = projr(yl) in
17     if yparties = (xpk1, xpk2) then
18       if ycontract = ct' then
19         let ysig = projr(y) in
20         let ym = check_getmsg(xpk1, ysig) in
21         if ym = yl then
22           out(c, sign(skT, y)); unlock
```

Resolve2′ is built from Resolve2 just by removing from Resolve2 lines 39-44 and line 46. Because Resolve2′ replies to a request without checking the status of the requested contract, it will always reply with the resolve acknowledgement.

```
23 let Resolve2' =
24   lock; in(c, x);
25   let xcmd = projl(x) in
26   if xcmd = resolve2 then
27     let y = projr(x) in
28     let ypcs1 = projl(y) in
29     let ysig2 = projr(y) in
30     let ycontract = check_getmsg(xpk2, ysig2) in
31     if ycontract = ct' then
32       let ycheck = checkpcs(ct', xpk1, xpk2,
33                             pk(skT), ypcs1) in
34       if ycheck = ok then
35         let ysig1 = convertpcs(skT, ypcs1) in
36         out(c, sign(skT, (ysig1, ysig2))); unlock
```

Resolve1′ is built from Resolve1 just by removing from Resolve1 lines 60-65 and line 67. Because Resolve1′ replies to a request without checking the status of the requested contract, it will always reply with the resolve acknowledgement.

```
37 let Resolve1' =
38   lock; in(c, x);
39   let xcmd = projl(x) in
40   if xcmd = resolve1 then
41     let y = projr(x) in
42     let ysig1 = projl(y) in
43     let ypcs2 = projr(y) in
44     let ycontract = check_getmsg(xpk1, ysig1) in
45     if ycontract = ct' then
46       let ycheck = checkpcs(ct', xpk2, xpk1,
47                             pk(skT), ypcs2) in
48       if ycheck = ok then
49         let ysig2 = convertpcs(skT, ypcs2) in
50         out(c, sign(skT, (ysig1, ysig2))); unlock
```