Proving More Observational Equivalences with ProVerif

Vincent Cheval[1] and Bruno Blanchet[2]

[1] LSV, ENS Cachan & CNRS & INRIA Saclay Île-de-France, France
[2] INRIA Paris-Rocquencourt, France

Abstract. This paper presents an extension of the automatic protocol verifier ProVerif in order to prove more observational equivalences. ProVerif can prove observational equivalence between processes that have the same structure but differ by the messages they contain. In order to extend the class of equivalences that ProVerif handles, we extend the language of terms by defining more functions (destructors) by rewrite rules. In particular, we allow rewrite rules with inequalities as side-conditions, so that we can express tests "if then else" inside terms. Finally, we provide an automatic procedure that translates a process into an equivalent process that performs as many actions as possible inside terms, to allow ProVerif to prove the desired equivalence. These extensions have been implemented in ProVerif and allow us to automatically prove anonymity in the private authentication protocol by Abadi and Fournet.

1 Introduction

Today, many applications that manipulate private data incorporate a cryptographic protocol, in order to ensure that such private information is never disclosed to anyone but the entitled entities. However, it has been shown that some currently used cryptographic protocols are flawed, e.g., the e-passport protocols [5]. It is therefore essential to obtain as much confidence as possible in the correctness of security protocols. To this effect, several tools have been developed to automatically verify security protocols. Until recently, most tools focused on reachability properties (or trace properties), such as authentication and secrecy, which specify that the protocols cannot reach a bad state. However, privacy-type properties cannot be naturally formalised as reachability properties and require the notion of behavioural equivalence, in order to specify the indistinguishability between several instances of the protocols. In the literature, the notion of *may-testing equivalence* was first introduced in [16] and has been studied for several calculi, e.g., the spi-calculus [3,13]. Typically, two processes P and Q are may-testing equivalent if for any process O, the processes $P \mid O$ and $Q \mid O$ can both emit on the same channels. However, the high difficulty of deciding this equivalence led to the introduction of stronger equivalences such as *observational equivalence* that additionally checks the bisimilarity of the process P and Q. This notion was the focus of several works, e.g., [7,12]. In this paper, we focus on the automation of the proofs of observational equivalence.

Related Work. The first algorithms to verify equivalence properties for security protocols dealt with a bounded number of sessions, a fixed set of basic primitives, and no else branches [14,13], but their complexity was too large for practical implementations. [6] showed that diff-equivalence, a strong equivalence between processes that have the same structure but differ by the terms they contain, is also decidable for bounded processes without else-branches; this result applies in particular to the detection of off-line guessing attacks against password-based protocols and to the proof of strong secrecy. However, the procedure does not seem to be well-suited for an implementation. Recently, a more practical algorithm was designed for bounded processes with else branches, non-determinism, and a fixed set of primitives [9] but there is no available implementation. These techniques rely on a symbolic semantics [8,12,15]: in such a semantics, the messages that come from the adversary are represented by variables, to avoid an unbounded case distinction on these messages.

To our knowledge, only three works resulted in automatic tools that verify equivalence properties for security protocols: ProVerif [7], SPEC [17], and AKiSs [10]. The tool SPEC provides a decision procedure for observational equivalence for processes in the spi-calculus. The tool AKiSs decides a weaker equivalence close to the may-testing equivalence for a wide variety of primitives. The scope of these two tools is limited to bounded determinate processes without non-trivial else branches, that is, processes whose executions are entirely determined by the adversary inputs. At last, the tool ProVerif was first a protocol analyser for trace properties but, since [7], it can also check the diff-equivalence between processes written in the applied pi calculus [1]. Although the diff-equivalence is stronger than observational equivalence, it still allows one to express many interesting properties such as anonymity and unlinkability, and it is much easier to prove than observational equivalence. Furthermore, ProVerif is the only tool that accepts unbounded processes with else branches and any cryptographic primitives that can be represented by an equational theory and/or rewrite rules. Even if it does not always terminate, it was shown very efficient for many case studies (e.g., proving the absence of guessing attacks in EKE, proving the core security of JFK [7] or proving anonymity and unlinkability of the Active Authentication protocol of the electronic passport [4]). Hence the present paper focuses on the tool ProVerif.

Motivation. Since the notion of equivalence proved by ProVerif is stronger than observational equivalence, it may yield false attacks. Indeed, ProVerif proves equivalences $P \approx Q$ in which P and Q are two variants of the same process obtained by selecting different terms for P and Q. Moreover, ProVerif requires that all tests yield the same result in both processes, in particular the tests of conditional branchings. Thus, for a protocol that does not satisfy this condition, ProVerif will fail to prove equivalence. Unfortunately, many indistinguishable processes do not satisfy this condition. Consider for example the processes:

$$P \stackrel{def}{=} c(x).\mathsf{if}\ x = \mathsf{pk}(sk_A)\ \mathsf{then}\ \overline{c}\langle\{s\}_{\mathsf{pk}(sk_A)}\rangle\ \mathsf{else}\ \overline{c}\langle\{N_p\}_{\mathsf{pk}(sk_A)}\rangle$$
$$Q \stackrel{def}{=} c(x).\mathsf{if}\ x = \mathsf{pk}(sk_B)\ \mathsf{then}\ \overline{c}\langle\{s\}_{\mathsf{pk}(sk_B)}\rangle\ \mathsf{else}\ \overline{c}\langle\{N_q\}_{\mathsf{pk}(sk_B)}\rangle$$

where all names but c are private and the public keys $\mathsf{pk}(sk_A)$ and $\mathsf{pk}(sk_B)$ are public. The protocol P is simply waiting for the public key of the agent A ($\mathsf{pk}(sk_A)$) on a channel c. If P receives it, then he sends some secret s encrypted with A's public key; otherwise, he sends a fresh nonce N_p encrypted with A's public key on channel c. On the other hand, the protocol Q does similar actions but is waiting for the public key of the agent B ($\mathsf{pk}(sk_B)$) instead of A. Assuming that the attacker does not have access to the private keys of A and B, then the two protocols are equivalent since the attacker cannot differentiate $\{s\}_{\mathsf{pk}(sk_A)}$, $\{N_p\}_{\mathsf{pk}(sk_A)}$, $\{s\}_{\mathsf{pk}(sk_B)}$, and $\{N_q\}_{\mathsf{pk}(sk_B)}$.

However, if the intruder sends the public key of the agent A ($\mathsf{pk}(sk_A)$), then the test of the conditional branching in P will succeed ($\mathsf{pk}(sk_A) = \mathsf{pk}(sk_A)$) whereas the test of the same conditional branching in Q will fail ($\mathsf{pk}(sk_A) \neq \mathsf{pk}(sk_B)$). Since this test does not yield the same result in both processes, ProVerif will fail to prove the equivalence between P and Q. This false attack also occurs in more realistic case studies, e.g., the private authentication protocol [2] and the Basic Access Control protocol of the e-passport [5].

Our Contribution. Our main contribution consists in addressing the issue of false attacks due to conditional branchings. In particular, we allow function symbols defined by rewrite rules with inequalities as side-conditions, so that we can express tests of conditional branchings directly inside terms (Section 2). Therefore, we still consider equivalences between processes that differ by the terms they contain, but our term algebra is now richer as it can express tests. Hence, we can now prove equivalences between processes that take different branches in internal tests, provided that what they do after these tests can be merged into a single process. We show how the original Horn clauses based algorithm of ProVerif can be adapted to our new calculus (Sections 3 and 4). Moreover, we provide an automatic procedure that transforms a process into an equivalent process that contains as few conditional branchings as possible, which allows ProVerif to prove equivalence on a larger class of processes. In particular, the implementation of our extension in ProVerif allowed us to automatically prove anonymity of the private authentication protocol for an unbounded number of sessions (Section 5). Anonymity was already proven by hand in [2] for the private authentication protocol; we automate this proof for a slightly simplified model. We eliminated some false attacks for the Basic Access Control protocol of the e-passport; however, other false attacks remain so we are still unable to conclude for this protocol. Our implementation is available as ProVerif 1.87beta, at http://proverif.inria.fr. A long version with additional details and proofs is available at http://www.lsv.ens-cachan.fr/∼cheval/(BC)POST13.pdf.

2 Model

This section introduces our process calculus, by giving its syntax and semantics. As mentioned above, our work extends the behaviour of destructor symbols, so our syntax and semantics of terms change in comparison to the original calculus

$M ::=$	message
x, y, z	variables
a, b, c	names
$\mathsf{f}(M_1, \ldots, M_n)$	constructor application
$U ::=$	may-fail message
M	message
fail	failure
u	may-fail variable
$D ::=$	term evaluation
U	may-fail message
$\text{eval } \mathsf{h}(D_1, \ldots, D_n)$	function evaluation
$P, Q, R ::=$	processes
0	nil
$M(x).P$	input
$\overline{M}\langle N\rangle.P$	output
$P \mid Q$	parallel composition
$!\,P$	replication
$(\nu a)P$	restriction
$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	term evaluation

Fig. 1. Syntax of terms and processes

of ProVerif [7]. However, we did not modify the syntax of processes thus the semantics of processes differs only due to changes coming from the modifications in the semantics of terms.

2.1 Syntax

The syntax of our calculus is summarised in Fig. 1. The messages sent on the network are modelled using an abstract term algebra. We assume an infinite set of names $\mathcal{N}$ and an infinite set of variables $\mathcal{X}$. We also consider a signature Σ consisting of a finite set of function symbols with their arity. We distinguish two categories of function symbols: constructors f and destructors g. Constructors build terms; destructors, defined by rewrite rules, manipulate terms, as detailed below. We denote by h a constructor or a destructor. *Messages* M are terms built from variables, names, and constructors applied to terms.

We define an *equational theory* by a finite set of equations $M = N$, where M, N are terms without names. The equational theory is then obtained from these equations by reflexive, symmetric, and transitive closure, closure under application of function symbols, and closure under substitution of terms for variables. By identifying an equational theory with its signature Σ, we denote $M =_\Sigma N$ an equality modulo the equational theory, and $M \neq_\Sigma N$ an inequality modulo the equational theory. We write $M = N$ and $M \neq N$ for syntactic equality and inequality, respectively. In this paper, we only consider consistent equational theories, i.e., there exist terms M and N such that $M \neq_\Sigma N$.

Destructors. In [7], the rewrite rules describing the behaviour of destructors follow the usual definition of a rewrite rule. However, as previously mentioned, we want to introduce tests directly into terms and more specifically into the definition of destructors. Hence, we introduce *formulas* on messages in order to express these tests. We consider formulas ϕ of the form $\bigwedge_{i=1}^{n} \forall \tilde{x}_i.M_i \neq_\Sigma N_i$, where $\tilde{x}$ stands for a sequence of variables $x_1, \ldots, x_k$. We denote by $\top$ and $\bot$ the *true* and *false* formulas, respectively corresponding to an empty conjunction ($n = 0$) and to $x \neq_\Sigma x$, for instance. Formulas will be used as side conditions for destructors. We denote by $fv(\phi)$ the free variables of ϕ, i.e., the variables that are not universally quantified. Let σ be a substitution mapping variables to ground terms. We define $\sigma \vDash \phi$ as follows: $\sigma \vDash \bigwedge_{i=1}^{n} \forall \tilde{x}_i.M_i \neq_\Sigma N_i$ if and only if for $i = 1, \ldots n$, for all σ_i of domain $\tilde{x}_i$, $\sigma\sigma_i M_i \neq_\Sigma \sigma\sigma_i N_i$.

In [7], destructors are partial functions defined by rewrite rules; when no rewrite rule can be applied, we say that the destructor fails. However, this formalism does not allow destructors to succeed when one of their arguments fails. We shall need this feature in order to include as many tests as possible in terms. Therefore, we extend the definition of destructors by defining *may-fail messages*, denoted by U, which can be messages M, the special value fail, or a variable u. We separate fail from ordinary messages M so that the equational theory does not apply to fail. May-fail messages represent the possible arguments and result of a destructor. We differentiate variables for may-fail messages, denoted u, v, w from variables for messages, denoted x, y, z. A may-fail variable u can be instantiated by a may-fail term while a message variable x can be instantiated only by a message, and so cannot be instantiated by fail.

For two ground may-fail messages U_1 and U_2, we say that $U_1 =_\Sigma U_2$ if and only if $U_1 = U_2 = \mathsf{fail}$ or U_1, U_2 are both messages, denoted M_1, M_2, and $M_1 =_\Sigma M_2$. Given a signature Σ, a destructor g of arity n is defined by a finite set of rewrite rules $\mathsf{g}(U_1, \ldots, U_n) \to U \mid\mid \phi$ where $U_1, \ldots, U_n, U$ are may-fail messages that do not contain any name, ϕ is a formula as defined above that does not contain any name, and the variables of U and $fv(\phi)$ are bound in $U_1, \ldots, U_n$. Note that all variables in $fv(\phi)$ are necessarily message variables. Variables are subject to renaming. We omit the formula ϕ when it is $\top$. We denote by $\mathrm{def}_\Sigma(\mathsf{g})$ the set of rewrite rules describing g in the signature Σ.

Example 1. Consider a symmetric encryption scheme where the decryption function either properly decrypts a ciphertext using the correct private key, or fails. To model this encryption scheme, we consider, in a signature Σ, the constructor senc for encryption and the destructor sdec for decryption, with the following rewrite rules:

- $\mathsf{sdec}(\mathsf{senc}(x, y), y) \to x$ (decryption succeeds)
- $\mathsf{sdec}(x, y) \to \mathsf{fail} \mid\mid \forall z.x \neq_\Sigma \mathsf{senc}(z, y)$ (decryption fails, because x is not a ciphertext under the correct key)
- $\mathsf{sdec}(\mathsf{fail}, u) \to \mathsf{fail}$, $\mathsf{sdec}(u, \mathsf{fail}) \to \mathsf{fail}$ (the arguments failed, the decryption also fails)

Let $U_1, \ldots, U_n$ be may-fail messages and g be a destructor of arity n. We say that g rewrites $U_1, \ldots, U_n$ into U, denoted $\mathsf{g}(U_1, \ldots, U_n) \to U$, if there exist

$\mathsf{g}(U'_1, \ldots, U'_n) \to U' \mid\mid \phi$ in $\mathrm{def}_\Sigma(\mathsf{g})$, and a substitution σ such that $\sigma U'_i =_\Sigma U_i$ for all $i = 1 \ldots n$, $\sigma U' = U$ and $\sigma \vDash \phi$. At last, we ask that, given a signature Σ, for all destructors g of arity n, $\mathrm{def}_\Sigma(\mathsf{g})$ satisfies the following properties:

P1. For all ground may-fail messages $U_1, \ldots, U_n$, there exists a may-fail message U such that $\mathsf{g}(U_1, \ldots, U_n) \to U$.
P2. For all ground may-fail messages $U_1, \ldots, U_n, V_1, V_2$, if $\mathsf{g}(U_1, \ldots, U_n) \to V_1$ and $\mathsf{g}(U_1, \ldots, U_n) \to V_2$ then $V_1 =_\Sigma V_2$.

Property P1 expresses that all destructors are total while Property P2 expresses that they are deterministic (modulo the equational theory). By Property P2, a destructor cannot reduce to fail and to a message at the same time.

In Example 1, the destructor sdec follows the classical definition of the symmetric decryption. However, thanks to the formulas and the fact that the arguments of a destructor can fail, we can describe the behaviour of new primitives.

Example 2. We define a destructor that tests equality and returns a boolean by:

$$\mathsf{eq}(x, x) \to \mathsf{true} \qquad \mathsf{eq}(x, y) \to \mathsf{false} \mid\mid x \neq_\Sigma y$$
$$\mathsf{eq}(\mathsf{fail}, u) \to \mathsf{fail} \qquad \mathsf{eq}(u, \mathsf{fail}) \to \mathsf{fail}$$

This destructor fails when one of its arguments fails. This destructor could not be defined in ProVerif without our extension, because one could not test $x \neq_\Sigma y$.

From Usual Destructors to our Extension. From a destructor defined, as in [7], by rewrite rules $\mathsf{g}(M_1, \ldots, M_n) \to M$ without side conditions and such that the destructor is considered to fail when no rewrite rule applies, we can build a destructor in our formalism. The algorithm is given in Lemma 1 below.

Lemma 1. *Consider a signature Σ. Let g be a destructor of arity n described by the set of rewrite rules $\mathcal{S} = \{\mathsf{g}(M_1^i, \ldots, M_n^i) \to M^i \mid i = 1, \ldots, m\}$. Assume that g is deterministic, i.e., $\mathcal{S}$ satisfies Property P2. The following set $\mathrm{def}_\Sigma(\mathsf{g})$ satisfies Properties P1 and P2:*

$$\begin{aligned}\mathrm{def}_\Sigma(\mathsf{g}) = \mathcal{S} &\cup \{\mathsf{g}(x_1, \ldots, x_n) \to \mathsf{fail} \mid\mid \phi\} \\ &\cup \{\mathsf{g}(u_1, \ldots, u_{k-1}, \mathsf{fail}, u_{k+1}, \ldots, u_n) \to \mathsf{fail} \mid k = 1, \ldots, n\}\end{aligned}$$

where $\phi = \bigwedge_{i=1}^m \forall \tilde{y}_i.(x_1, \ldots, x_n) \neq_\Sigma (M_1^i, \ldots, M_n^i)$ and $\tilde{y}_i$ are the variables of $(M_1^i, \ldots, M_n^i)$, and $x_1, \ldots, x_n$ are message variables.

The users can therefore continue defining destructors as before in ProVerif; the tool checks that the destructors are deterministic and automatically completes the definition following Lemma 1.

Generation of Deterministic and Total Destructors. With our extension, we want the users to be able to define destructors with side conditions. However, these destructors must satisfy Properties P1 and P2. Instead of having to verify these properties a posteriori, we use a method that allows the user to provide

precisely the destructors that satisfy P1 and P2: the user inputs a sequence of rewrite rules $\mathsf{g}(U_1^1, \ldots, U_n^1) \to V^1$ otherwise $\ldots$ otherwise $\mathsf{g}(U_n^m, \ldots, U_n^m) \to V^m$ where U_k^i, V^i are may-fail messages, for all i, k. Intuitively, this sequence indicates that when reducing terms by the destructor g, we try to apply the rewrite rules in the order of the sequence, and if no rule is applicable then the destructor fails. To model the case where no rule is applicable, we add the rewrite rule $\mathsf{g}(u_1, \ldots, u_n) \to \mathsf{fail}$ where $u_1, \ldots, u_n$ are distinct may-fail variables, at the end of the previous sequence of rules. Then, the obtained sequence is translated into a set $\mathcal{S}$ of rewrite rules with side conditions as follows

$$\mathcal{S} \stackrel{def}{=} \left\{ \mathsf{g}(U_1^i, \ldots, U_n^i) \to V^i \;||\; \bigwedge_{j<i} \forall \tilde{u}^j.(U_1^i, \ldots, U_n^i) \neq_\Sigma (U_1^j, \ldots, U_n^j) \right\}_{i=1..m+1}$$

where $\tilde{u}^j$ are the variables of $U_1^j, \ldots, U_n^j$. We use side-conditions to make sure that rule i is not applied if rule j for $j < i$ can be applied. Notice that, in the set $\mathcal{S}$ defined above, the formulas may contain may-fail variables or the constant fail. In order to match our formalism, we instantiate these variables by either a message variable or fail, and then we simplify the formulas.

Term Evaluation. A *term evaluation* represents the evaluation of a series of constructors and destructors. The term evaluation eval $\mathsf{h}(D_1, \ldots, D_n)$ indicates that the function symbol h will be evaluated. While all destructors must be preceded by eval, some constructors might also be preceded by eval in a term evaluation. In fact, the reader may ignore the prefix eval since eval h and h have the same semantics with the initial definition of constructors with equations. However, eval becomes useful when we convert equations into rewrite rules (see Section 4.1). The prefix eval is used to indicate whether a term has been evaluated or not. Even though we allow may-fail messages in term evaluations, since no construct binds may-fail variables in processes, only messages M and fail may in fact occur. In order to avoid distinguishing constructors and destructors in the definition of term evaluation, for f a constructor of arity n, we let $\mathrm{def}_\Sigma(\mathsf{f}) = \{\mathsf{f}(x_1, \ldots, x_n) \to \mathsf{f}(x_1, \ldots, x_n)\} \cup \{\mathsf{f}(u_1, \ldots, u_{i-1}, \mathsf{fail}, u_{i+1}, \ldots, u_n) \to \mathsf{fail} \mid i = 1, \ldots, n\}$. The second part of the union corresponds to the failure cases: the constructor fails if, and only if, one of its arguments fails.

Processes. At last, the syntax of processes corresponds exactly to [7]. A trailing 0 can be omitted after an input or an output. An else branch can be omitted when it is else 0.

Even if the condition if $M = N$ then P else Q is not included in our calculus, it can be defined as let $x = \mathsf{equals}(M, N)$ in P else Q, where x is a fresh variable and equals is a binary destructor with the rewrite rules $\{\mathsf{equals}(x, x) \to x, \mathsf{equals}(x, y) \to \mathsf{fail} \;||\; x \neq_\Sigma y, \mathsf{equals}(\mathsf{fail}, u) \to \mathsf{fail}, \mathsf{equals}(u, \mathsf{fail}) \to \mathsf{fail}\}$. The destructor equals succeeds if and only if its two arguments are equal messages modulo the equational theory and different from fail. We always include this destructor in the signature Σ. An evaluation context C is a closed context built from $[\,]$, $C \mid P$, $P \mid C$, and $(\nu a)C$.

Example 3. We consider a slightly simplified version of the private authentication protocol given in [2]. In this protocol, a participant A is willing to engage in communication and reveal its identity to a participant B, without revealing it to other participants. The cryptographic primitives used in this protocol are the asymmetric encryption and pairing. Expressed in ProVerif syntax, the participants A and B proceed as follows:

$$
\begin{aligned}
A(sk_a, sk_b) \quad &\stackrel{def}{=} (\nu n_a)\overline{c}\langle \mathsf{aenc}(\langle n_a, \mathsf{pk}(sk_a)\rangle, \mathsf{pk}(sk_b))\rangle.c(x).0\\
B(sk_b, sk_a) \quad &\stackrel{def}{=} (\nu n_b)c(y).\mathsf{let}\ x = \mathsf{adec}(y, sk_b)\ \mathsf{in}\\
&\qquad \mathsf{let}\ xn_a = \mathsf{proj}_1(x)\ \mathsf{in}\\
&\qquad\quad \mathsf{let}\ z = \mathsf{equals}(\mathsf{proj}_2(x), \mathsf{pk}(sk_a))\ \mathsf{in}\\
&\qquad\qquad \overline{c}\langle \mathsf{aenc}(\langle xn_a, \langle n_b, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(sk_a))\rangle.0\\
&\qquad\quad \mathsf{else}\ \overline{c}\langle \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))\rangle.0\\
&\qquad \mathsf{else}\ \overline{c}\langle \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))\rangle.0\\
&\quad \mathsf{else}\ \overline{c}\langle \mathsf{senc}(n_b, \mathsf{pk}(sk_b))\rangle.0\\
System(sk_a, sk_b) \stackrel{def}{=}\ & A(sk_a, sk_b) \mid B(sk_b, sk_a)
\end{aligned}
$$

where sk_a and sk_b are the respective private keys of A and B, proj_1 and proj_2 are the two projections of a pairing denoted by $\langle\ ,\ \rangle$, aenc and adec are the asymmetric encryption and decryption, and $\mathsf{pk}(sk)$ is the public key associated to the private key sk.

In other words, A first sends to B a nonce n_a and its own public key $\mathsf{pk}(sk_a)$ encrypted with the public key of B, $\mathsf{pk}(sk_b)$. After receiving this message, B checks that the message is of the correct form and that it contains the public key of A. If so, B sends back to A the "correct" message composed of the nonce n_a he received, n_b a freshly generated nonce, and his own public key ($\mathsf{pk}(sk_b)$), all this encrypted with the public key of A. Otherwise, B sends back a "dummy" message, $\mathsf{aenc}(n_b, \mathsf{pk}(sk_b))$. From the point of view of the attacker, this dummy message is indistinguishable from the "correct" one since the private keys sk_a and sk_b are unknown to the attacker, so the attacker should not be able to tell whether A or another participant is talking to B. This is what we are going to prove formally.

2.2 Semantics

The semantics of processes and term evaluations is summarised in Fig. 2. The formula $D \Downarrow_\Sigma U$ means that D evaluates to U. When the term evaluation corresponds to a function h preceded by eval, the evaluation proceeds recursively by evaluating the arguments of the function and then by applying the rewrite rules of h in $\mathrm{def}_\Sigma(\mathsf{h})$ to compute U, taking into account the side-conditions in ϕ.

The semantics of processes in ProVerif is defined by a *structural equivalence*, denoted $\equiv$, and some *internal reductions*. The structural equivalence $\equiv$ is the smallest equivalence relation on extended processes that is closed under α-conversion of names and variables, by application of evaluation contexts, and

$U \Downarrow_\Sigma U$
eval $\mathsf{h}(D_1, \ldots, D_n) \Downarrow_\Sigma \sigma U$
if $\mathsf{h}(U_1, \ldots, U_n) \to U \mid\mid \phi$ is in $\mathrm{def}_\Sigma(\mathsf{h})$ and σ is such that for all i, $D_i \Downarrow_\Sigma V_i$, $V_i =_\Sigma \sigma U_i$ and $\sigma \vDash \phi$

$$\overline{N}\langle M\rangle.Q \mid N'(x).P \rightarrow_\Sigma Q \mid P\{^M/_x\} \quad \text{if } N =_\Sigma N' \quad \text{(Red I/O)}$$
$$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \rightarrow_\Sigma P\{^M/_x\} \quad \text{if } D \Downarrow_\Sigma M \quad \text{(Red Fun 1)}$$
$$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \rightarrow_\Sigma Q \quad \text{if } D \Downarrow_\Sigma \mathsf{fail} \quad \text{(Red Fun 2)}$$
$$!\,P \rightarrow_\Sigma P \mid !\,P \quad \text{(Red Repl)}$$
$$P \rightarrow_\Sigma Q \Rightarrow P \mid R \rightarrow_\Sigma Q \mid R \quad \text{(Red Par)}$$
$$P \rightarrow_\Sigma Q \Rightarrow (\nu a)P \rightarrow_\Sigma (\nu a)Q \quad \text{(Red Res)}$$
$$P' \equiv P, P \rightarrow_\Sigma Q, Q \equiv Q' \Rightarrow P' \rightarrow_\Sigma Q' \quad \text{(Red } \equiv\text{)}$$

Fig. 2. Semantics of terms and processes

satisfying some further basic structural rules such as $P \mid 0 \equiv P$, associativity and commutativity of $\mid$, and scope extrusion. However, this structural equivalence does not substitute terms equal modulo the equational theory and does not model the replication. Both properties are in fact modelled as internal reduction rules for processes. This semantics differs from [7] by the rule (Red Fun 2) which previously corresponded to the case where the term evaluation D could not be reduced whereas D is reduced to fail in our semantics.

Both relations $\equiv$ and $\rightarrow_\Sigma$ are defined only on closed processes. We denote by $\rightarrow_\Sigma^*$ the reflexive and transitive closure of $\rightarrow_\Sigma$, and by $\rightarrow_\Sigma^* \equiv$ its composition with $\equiv$. When Σ is clear from the context, we abbreviate $\rightarrow_\Sigma$ to $\rightarrow$ and $\Downarrow_\Sigma$ to $\Downarrow$.

3 Using Biprocesses to Prove Observational Equivalence

In this section, we recall the notions of observational equivalence and biprocesses introduced in [7].

Definition 1. *We say that the process P* emits on *M ($P\downarrow_M$) if and only if $P \rightarrow_\Sigma^* \equiv C[\overline{M'}\langle N\rangle.R]$ for some evaluation context C that does not bind $fn(M)$ and $M =_\Sigma M'$.*

Observational equivalence*, denoted $\approx$, is the largest symmetric relation $\mathcal{R}$ between closed processes such that $P\ \mathcal{R}\ Q$ implies:*

1. *if $P\downarrow_M$, then $Q\downarrow_M$;*
2. *if $P \rightarrow_\Sigma^* P'$, then $Q \rightarrow_\Sigma^* Q'$ and $P'\ \mathcal{R}\ Q'$ for some Q';*
3. *$C[P]\ \mathcal{R}\ C[Q]$ for all closed evaluation contexts C.*

Intuitively, an evaluation context may represent an adversary, and two processes are observationally equivalent when no adversary can distinguish them. One of the most difficult parts of deciding the observational equivalence between two processes directly comes from the second item of Definition 1. Indeed, this

$$\overline{N}\langle M\rangle.Q \mid N'(x).P \;\to\; Q \mid P\{^{M}/_{x}\} \qquad \text{(Red I/O)}$$
$$\text{if } \mathsf{fst}(N) =_\Sigma \mathsf{fst}(N') \text{ and } \mathsf{snd}(N) =_\Sigma \mathsf{snd}(N')$$
$$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \to P\{^{\mathsf{diff}[M_1,M_2]}/_{x}\} \qquad \text{(Red Fun 1)}$$
$$\text{if } \mathsf{fst}(D) \Downarrow_\Sigma M_1 \text{ and } \mathsf{snd}(D) \Downarrow_\Sigma M_2$$
$$\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \to Q \qquad \text{(Red Fun 2)}$$
$$\text{if } \mathsf{fst}(D) \Downarrow_\Sigma \mathsf{fail} \text{ and } \mathsf{snd}(D) \Downarrow_\Sigma \mathsf{fail}$$

Fig. 3. Generalized rules for biprocesses

condition indicates that each reduction of a process has to be matched in the second process. However, we consider a process algebra with replication, hence there are usually infinitely many candidates for this mapping.

To solve this problem, [7] introduces a calculus that represents pairs of processes, called *biprocesses*, that have the same structure and differ only by the terms and term evaluations that they contain. The grammar of the calculus is a simple extension of the grammar of Fig. 1 with additional cases so that $\mathsf{diff}[M, M']$ is a term and $\mathsf{diff}[D, D']$ is a term evaluation.

Given a biprocess P, we define two processes $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$, as follows: $\mathsf{fst}(P)$ is obtained by replacing all occurrences of $\mathsf{diff}[M, M']$ with M and $\mathsf{diff}[D, D']$ with D in P, and similarly, $\mathsf{snd}(P)$ is obtained by replacing $\mathsf{diff}[M, M']$ with M' and $\mathsf{diff}[D, D']$ with D' in P. We define $\mathsf{fst}(D)$, $\mathsf{fst}(M)$, $\mathsf{snd}(D)$, and $\mathsf{snd}(M)$ similarly. A process or context is said to be *plain* when it does not contain diff.

Definition 2. *Let P be a closed biprocess. We say that P* satisfies observational equivalence *when $\mathsf{fst}(P) \approx \mathsf{snd}(P)$.*

The semantics of biprocesses is defined as in Fig. 2 with generalized rules (Red I/O), (Red Fun 1), and (Red Fun 2) given in Fig. 3.

The semantics of biprocesses is such that a biprocess reduces if and only if both sides of the biprocess reduce in the same way: a communication succeeds on both sides; a term evaluation succeeds on both sides or fails on both sides. When the two sides of the biprocess reduce in different ways, the biprocess blocks. The following lemma shows that, when both sides of a biprocess always reduce in the same way, then that biprocess satisfies observational equivalence.

Lemma 2. *Let P_0 be a closed biprocess. Suppose that, for all plain evaluation contexts C, all evaluation contexts C', and all reductions $C[P_0] \to^* P$,*

1. *if $P \equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ then $\mathsf{fst}(N) =_\Sigma \mathsf{fst}(N')$ if and only if $\mathsf{snd}(N) =_\Sigma \mathsf{snd}(N')$;*
2. *if $P \equiv C'[\mathsf{let}\ x = D\ \mathsf{in}\ Q\ \mathsf{else}\ R]$ then $\mathsf{fst}(D) \Downarrow_\Sigma \mathsf{fail}$ if and only if $\mathsf{snd}(D) \Downarrow_\Sigma \mathsf{fail}$.*

Then P_0 satisfies observational equivalence.

Intuitively, the semantics for biprocesses forces that each reduction of a process has to be matched by the same reduction in the second process. Hence, verifying the second item of Definition 1 becomes less problematic since we reduce to one the number of possible candidates Q'.

Example 4. Coming back to the private authentication protocol detailed in Example 3, we want to verify the anonymity of the participant A. Intuitively, this protocol preserves anonymity if an attacker cannot distinguish whether B is talking to A or to another participant A', assuming that A, A', and B are honest participants and furthermore assuming that the intruder knows the public keys of A, A', and B. Hence, the anonymity property is modelled by an observational equivalence between two instances of the protocol: one where B is talking to A and the other where B is talking to A', which is modelled as follows:

$$\begin{aligned}&(\nu sk_a)(\nu sk'_a)(\nu sk_b)\overline{c}\langle\mathsf{pk}(sk_a)\rangle.\overline{c}\langle\mathsf{pk}(sk'_a)\rangle.\overline{c}\langle\mathsf{pk}(sk_b)\rangle.System(sk_a, sk_b)\\ \approx\ &(\nu sk_a)(\nu sk'_a)(\nu sk_b)\overline{c}\langle\mathsf{pk}(sk_a)\rangle.\overline{c}\langle\mathsf{pk}(sk'_a)\rangle.\overline{c}\langle\mathsf{pk}(sk_b)\rangle.System(sk'_a, sk_b)\end{aligned}$$

Since the "dummy" message and the "correct" one are indistinguishable from the point of view of the attacker, this equivalence holds. To prove this equivalence using ProVerif, we first have to transform this equivalence into a biprocess. This is easily done since only the private keys ska and ska' change between the two processes. Hence, we define the biprocess P_0 as follows:

$$(\nu sk_a)(\nu sk'_a)(\nu sk_b)\overline{c}\langle\mathsf{pk}(sk_a)\rangle.\overline{c}\langle\mathsf{pk}(sk'_a)\rangle.\overline{c}\langle\mathsf{pk}(sk_b)\rangle.System(\mathsf{diff}[sk_a, sk'_a], sk_b)$$

Note that $\mathsf{fst}(P_0)$ and $\mathsf{snd}(P_0)$ correspond to the two protocols of the equivalence. For simplicity, we only consider two sessions in this example but our results also apply to an unbounded number of sessions (for the definition of anonymity of [5]).

4 Clause Generation

In [7], observational equivalence is verified by translating the considered biprocess into a set of Horn clauses, and using a resolution algorithm on these clauses. We adapt this translation to our new destructors.

4.1 From Equational Theories to Rewrite Rules

Equational theories are a very powerful tool for modeling cryptographic primitives. However, for a practical algorithm, it is easier to work with rewrite rules rather than with equational theories. Hence in [7], a signature Σ with an equational theory is transformed into a signature Σ' with rewrite rules that models Σ, when Σ has the finite variant property [11]. These rewrite rules may rewrite a term M into several irreducible forms (the variants), which are all equal modulo Σ, and such that, when M and M' are equal modulo Σ, M and M' rewrite to at least one common irreducible form. We reuse the algorithm from [7] for generating Σ', adapting it to our formalism by just completing the rewrite rules of constructors with rewrite rules that reduce to fail when an argument is fail.

4.2 Patterns and Facts

In the clauses, the messages are represented by patterns, with the following grammar:

$p ::=$	pattern	$mp ::=$	may-fail pattern
x, y, z, i	variables	p	pattern
$f(p_1, \ldots, p_n)$	constructor application	u, v	may-fail variables
$a[p_1, \ldots, p_n]$	name	fail	failure

The patterns p are the same as in [7]. The variable i represents a session identifier for each replication of a process. A pattern $a[p_1, \ldots, p_n]$ is assigned to each name of a process P. The arguments $p_1, \ldots, p_n$ allow one to model that a fresh name a is created at execution of (νa). For example, in the process $!\, c'(x).(\nu a)P$, each name created by (νa) is represented by $a[i, x]$ where i is the session identifier for the replication and x is the message received as input in $c'(x)$. Hence, the name a is represented as a function of i and x. In two different sessions, (i, x) takes two different values, so the two created instances of a ($a[i, x]$) are different.

Since we introduced may-fail messages to represent the possible failure of a destructor, we also define *may-fail patterns* to represent the failure in clauses. As in messages and may-fail messages, a may-fail variable u can be instantiated by a pattern or fail, whereas a variable x cannot be instantiated by fail.

Clauses are built from the following predicates:

$F ::=$	facts
$\mathrm{att}'(mp, mp')$	attacker knowledge
$\mathrm{msg}'(p_1, p_2, p_1', p_2')$	output message p_2 on p_1 (resp. p_2' on p_1')
$\mathrm{input}'(p, p')$	input on p (resp. p')
$\mathrm{formula}(\bigwedge_i \forall \tilde{z}_i.p_i \neq_\Sigma p_i')$	formula
bad	bad

Intuitively, $\mathrm{att}'(mp, mp')$ means that the attacker may obtain mp in $\mathsf{fst}(P)$ and mp' in $\mathsf{snd}(P)$ by the same operations; the fact $\mathrm{msg}'(p_1, p_2, p_1', p_2')$ means that message p_2 may be output on channel p_1 by the process $\mathsf{fst}(P)$ while p_2' may be output on channel p_1' by the process $\mathsf{snd}(P)$ after the same reductions; $\mathrm{input}'(p, p')$ means that an input is possible on channel p in $\mathsf{fst}(P)$ and on channel p' in $\mathsf{snd}(P)$. Note that both facts msg' and input' contain only patterns and not may-fail patterns. Hence channels and sent terms are necessarily messages and so cannot be fail. The fact $\mathrm{formula}(\phi)$ means that ϕ has to be satisfied. At last, bad serves in detecting violations of observational equivalence: when bad is not derivable, we have observational equivalence.

4.3 Clauses for the Attacker

The capabilities of the attacker are represented by clauses adapted from the ones in [7] to fit our new formalism. We give below the clauses that differ from [7].

$$\mathrm{att}'(\mathsf{fail}, \mathsf{fail}) \tag{Rfail}$$

For each function h, for each pair of rewrite rules
$\mathsf{h}(U_1, \ldots, U_n) \to U \mid\mid \phi$ and $\mathsf{h}(U_1', \ldots, U_n') \to U' \mid\mid \phi'$
in $\mathrm{def}_{\Sigma'}(\mathsf{h})$ (after renaming of variables),
$$\mathrm{att}'(U_1, U_1') \wedge \ldots \wedge \mathrm{att}'(U_n, U_n') \wedge \mathrm{formula}(\phi \wedge \phi') \to \mathrm{att}'(U, U') \tag{Rf}$$

$$\text{input}'(x, x') \wedge \text{msg}'(x, z, y', z') \wedge \text{formula}(x' \neq_\Sigma y') \rightarrow \text{bad} \quad \text{(Rcom)}$$
$$\text{att}'(x, \mathsf{fail}) \rightarrow \text{bad} \quad \text{(Rfailure)}$$

plus the symmetric clauses (Rcom′) and (Rfailure′) obtained from (Rcom) and (Rfailure) by swapping the first and second arguments of att′ and input′, and the first and third arguments of msg′.

Clauses (Rf) apply a constructor or a destructor on the attacker's knowledge, given the rewrite rules in $\text{def}_{\Sigma'}(\mathsf{h})$. Since our destructors may return fail, by combining (Rf) with (Rfailure) or (Rfailure′), we can detect when a destructor succeeds in one variant of the biprocess and not in the other. We stress that, in clauses (Rfailure) and (Rcom), x, x', y, y' are message variables and so they cannot be instantiated by fail. (The messages sent on the network and the channels are never fail.)

4.4 Clauses for the Protocol

To translate the protocol into clauses, we first need to define evaluation on open terms, as a relation $D \Downarrow' (U, \sigma, \phi)$, where σ collects instantiations of D obtained by unification and ϕ collects the side conditions of destructor applications. More formally, the relation $D \Downarrow' (U, \sigma, \phi)$ specifies how instances of D evaluate: if $D \Downarrow' (U, \sigma, \phi)$, then for any substitution σ' such that $\sigma' \vDash \phi$, we have $\sigma'\sigma D \Downarrow_{\Sigma'} \sigma' U$. There may be several (U, σ, ϕ) such that $D \Downarrow' (U, \sigma, \phi)$ in case several instances of D reduce in a different way. This relation is defined as follows:

$$U \Downarrow' (U, \emptyset, \top)$$

$$\text{eval } \mathsf{h}(D_1, \ldots, D_n) \Downarrow' (\sigma_u V, \sigma_u \sigma', \sigma_u \phi' \wedge \sigma_u \phi)$$
$$\quad \text{if } (D_1, \ldots, D_n) \Downarrow' ((U_1, \ldots, U_n), \sigma', \phi'),$$
$$\quad \mathsf{h}(V_1, \ldots, V_n) \rightarrow V \mid\mid \phi \in \text{def}_{\Sigma'}(\mathsf{h}) \text{ and}$$
$$\quad \sigma_u \text{ is a most general unifier of } (U_1, V_1), \ldots, (U_n, V_n)$$

$$(D_1, \ldots, D_n) \Downarrow' ((\sigma_n U_1, \ldots, \sigma_n U_{n-1}, U_n), \sigma_n \sigma, \sigma_n \phi \wedge \phi_n)$$
$$\quad \text{if } (D_1, \ldots, D_{n-1}) \Downarrow' ((U_1, \ldots, U_{n-1}), \sigma, \phi) \text{ and } \sigma D_n \Downarrow' (U_n, \sigma_n, \phi_n)$$

The most general unifier of may-fail messages is computed similarly to the most general unifier of messages, even though specific cases hold due to may-fail variables and message variables: there is no unifier of M and fail, for any message M (including variables x, because these variables can be instantiated only by messages); the most general unifier of u and U is $\{^U/_u\}$; the most general unifier of fail and fail is the identity; finally, the most general unifier of M and M' is computed as usual.

The translation $[\![P]\!]\rho s H$ of a biprocess P is a set of clauses, where ρ is an environment that associates a pair of patterns with each name and variable, s is a sequence of patterns, and H is a sequence of facts. The empty sequence is written $\emptyset$; the concatenation of a pattern p to the sequence s is written s, p; the concatenation of a fact F to the sequence H is written $H \wedge F$. Intuitively, H represents the hypothesis of the clauses, ρ represents the names and variables

that are already associated with a pattern, and s represents the current values of session identifiers and inputs. When ρ associates a pair of patterns with each name and variable, and f is a constructor, we extend ρ as a substitution by $\rho(\mathsf{f}(M_1, \ldots, M_n)) = (\mathsf{f}(p_1, \ldots, p_n), \mathsf{f}(p'_1, \ldots, p'_n))$ where $\rho(M_i) = (p_i, p'_i)$ for all $i \in \{1, \ldots, n\}$. We denote by $\rho(M)_1$ and $\rho(M)_2$ the components of the pair $\rho(M)$. We let $\rho(\mathsf{diff}[M, M']) = (\rho(M)_1, \rho(M')_2)$.

The definition of $[\![P]\!]\rho sH$ is directly inspired from [7]. We only present below the case $[\![\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]\rho sH$.

$$
\begin{aligned}
&[\![\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]\rho sH = \\
&\quad \bigcup \{ [\![P]\!]((\sigma\rho)[x \mapsto (p, p')])(\sigma s, p, p')(\sigma H \wedge \mathrm{formula}(\phi)) \\
&\qquad \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((p, p'), \sigma, \phi) \} \\
&\quad \cup \bigcup \{ [\![Q]\!](\sigma\rho)(\sigma s)(\sigma H \wedge \mathrm{formula}(\phi)) \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((\mathsf{fail}, \mathsf{fail}), \sigma, \phi) \} \\
&\quad \cup \{ \sigma H \wedge \mathrm{formula}(\phi) \to \mathrm{bad} \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((p, \mathsf{fail}), \sigma, \phi) \} \\
&\quad \cup \{ \sigma H \wedge \mathrm{formula}(\phi) \to \mathrm{bad} \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((\mathsf{fail}, p'), \sigma, \phi) \}
\end{aligned}
$$

This formula is fairly similar to the one in [7]: when both $\rho(D)_1$ and $\rho(D)_2$ succeed, the process P is translated, instantiating terms with the substitution σ and taking into account the side-condition ϕ, to make sure that $\rho(D)_1$ and $\rho(D)_2$ succeed; when both fail, the process Q is translated; and at last when one of $\rho(D)_1$, $\rho(D)_2$ succeeds and the other fails, clauses deriving bad are generated. Since may-fail variables do not occur in D, we can show by induction on the computation of $\Downarrow'$ that, when $(\rho(D)_1, \rho(D)_2) \Downarrow' ((mp_1, mp_2), \sigma, \phi)$, mp_1 and mp_2 are either fail or a pattern, but cannot be a may-fail variable, so our definition of $[\![\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]\rho sH$ handles all cases.

4.5 Proving Equivalences

Let $\rho_0 = \{a \mapsto (a[\,], a[\,]) \mid a \in fn(P_0)\}$. We define the set of clauses that corresponds to biprocess P_0 as $\mathcal{R}_{P_0} = [\![P_0]\!]\rho_0\emptyset\emptyset \cup \{(\mathrm{Rfail}), \ldots, (\mathrm{Rfailure}')\}$. The following theorem enables us to prove equivalences from these clauses.

Theorem 1. *If* bad *is not a logical consequence of $\mathcal{R}_{P_0}$, then P_0 satisfies observational equivalence.*

This theorem shows the soundness of the translation. The proof of this theorem is adapted from the proof of Theorem 3 of [7]. Furthermore, since we use almost the same patterns and facts as in [7], we also use the algorithm proposed in [7] to automatically check if bad is a logical consequence of $\mathcal{R}_{P_0}$, with the only change that we use the unification algorithm for may-fail patterns.

5 Automatic Modification of the Protocol

In this section, we first present the kind of false attack that we want to avoid and then propose an algorithm to automatically generate, from a biprocess P, equivalent biprocesses on which ProVerif will avoid this kind of false attack.

5.1 Targeted False Attacks

We present a false attack on the anonymity of the private authentication protocol due to structural conditional branching.

Example 5. Coming back to the private authentication protocol (see Example 4), we obtained a biprocess P_0 on which we would ask ProVerif to check the equivalence. Unfortunately, ProVerif is unable to prove the equivalence of P_0 and yields a false attack. Indeed, consider the evaluation context C defined as follows:

$$C \stackrel{def}{=} \_ \mid (\nu n_i)c(x_{sk_a}).c(x_{sk_{a'}}).c(x_{sk_b}).\overline{c}\langle \mathsf{aenc}(\langle n_i, x_{sk_a}\rangle, x_{sk_b})\rangle$$

The biprocess $C[P_0]$ can be reduced as follows:

$$\begin{aligned} C[P_0] \rightarrow_\Sigma^* & (\nu n_i)(\nu sk_a)(\nu sk_{a'})(\nu sk_b) \\ & \big(\overline{c}\langle \mathsf{aenc}(\langle n_i, \mathsf{pk}(sk_a)\rangle, \mathsf{pk}(sk_b))\rangle \mid System(\mathsf{diff}[sk_a, sk_{a'}], sk_b)\big) \\ \rightarrow_\Sigma^* & (\nu n_i)(\nu sk_a)(\nu sk_{a'})(\nu sk_b)(A(\mathsf{diff}[sk_a, sk_{a'}], sk_b) \mid \\ & \quad \mathsf{let}\ z = \mathsf{equals}(\mathsf{proj}_2(\langle n_i, \mathsf{pk}(sk_a)\rangle)), \mathsf{pk}(\mathsf{diff}[sk_a, sk_{a'}]))\ \mathsf{in} \\ & \quad\ \overline{c}\langle \mathsf{aenc}(\langle n_i, \langle n_b, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(\mathsf{diff}[sk_a, sk_{a'}]))\rangle \\ & \quad\ \mathsf{else}\ \overline{c}\langle \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))\rangle) \end{aligned}$$

However from this point, the biprocess gets stuck, i.e., no internal reduction rule is applicable. More specifically, neither the internal rule (Red Fun 1) nor (Red Fun 2) is applicable. Indeed, if we denote $D = \mathsf{equals}(\mathsf{proj}_2(\langle n_i, sk_a\rangle)), \mathsf{pk}(\mathsf{diff}[sk_a, sk_{a'}]))$, we have that $\mathsf{snd}(D) \Downarrow_\Sigma \mathsf{fail}$ and $\mathsf{fst}(D) \Downarrow_\Sigma \mathsf{pk}(sk_a)$, which contradicts Item 2 of Lemma 2. So ProVerif cannot prove the equivalence. But, although a different branch of the let is taken, the process outputs the message $\mathsf{aenc}(\langle n_b, \langle n_a, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(sk_a))$ in the first variant (in branch of the let) and the message $\mathsf{aenc}(n_b, \mathsf{pk}(sk_b))$ in the second variant (else branch of the let). Intuitively, these two messages are indistinguishable, so in fact the attacker will not be able to determine which branch of the let is taken, and observational equivalence still holds.

In order to avoid the false attacks similar to Example 5, we transform term evaluations $\mathsf{let}\ x = D\ \mathsf{in}\ \overline{c}\langle M_1\rangle\ \mathsf{else}\ \overline{c}\langle M_2\rangle$ into a computation that always succeeds $\mathsf{let}\ x = D'\ \mathsf{in}\ \mathsf{let}\ m = D''\ \mathsf{in}\ \overline{c}\langle m\rangle$. The term evaluation D' will correspond to the value of the evaluation of D when the latter succeeds and a new constant $\mathsf{c_{fail}}$ when D fails. Thus we ensure that D' never fails. Moreover, the term evaluation D'' computes either M_1 or M_2 depending on the value of D', i.e., depending on whether D succeeds or not. The omitted else 0 branches are never taken. Since the same branch is always taken, the false attack disappears. To do that, we introduce three new destructors catchfail, letin, notfail and a constant $\mathsf{c_{fail}}$, which rely on the side conditions that we have added to destructors. These new destructors are defined as follows:

$$\begin{array}{lll} \mathrm{def}_\Sigma(\mathsf{catchfail}) = & \mathrm{def}_\Sigma(\mathsf{letin}) = & \mathrm{def}_\Sigma(\mathsf{notfail}) = \\ \quad \mathsf{catchfail}(x) \rightarrow x & \quad \mathsf{letin}(x, u, v) \rightarrow u \mid\mid x \neq_\Sigma \mathsf{c_{fail}} & \quad \mathsf{notfail}(x) \rightarrow \mathsf{fail} \\ \quad \mathsf{catchfail}(\mathsf{fail}) \rightarrow \mathsf{c_{fail}} & \quad \mathsf{letin}(\mathsf{c_{fail}}, u, v) \rightarrow v & \quad \mathsf{notfail}(\mathsf{fail}) \rightarrow \mathsf{c_{fail}} \\ & \quad \mathsf{letin}(\mathsf{fail}, u, v) \rightarrow \mathsf{fail} & \end{array}$$

One can easily check that $\mathrm{def}_\Sigma(\mathsf{catchfail})$, $\mathrm{def}_\Sigma(\mathsf{letin})$, and $\mathrm{def}_\Sigma(\mathsf{notfail})$ satisfy Properties P1 and P2. Intuitively, the destructor $\mathsf{catchfail}$ evaluates its argument and returns either the result of this evaluation when it did not fail or else returns the new constant $\mathsf{c_{fail}}$ instead of the failure constant fail. The destructor letin will get the result of $\mathsf{catchfail}$ as first argument and return its third argument if $\mathsf{catchfail}$ returned $\mathsf{c_{fail}}$, and its second argument otherwise. Importantly, $\mathsf{catchfail}$ never fails: it returns $\mathsf{c_{fail}}$ instead of fail. Hence, $\mathsf{let}\ x = D\ \mathsf{in}\ \overline{c}\langle M_1\rangle\ \mathsf{else}\ \overline{c}\langle M_2\rangle$ can be transformed into $\mathsf{let}\ x = \mathrm{eval}\ \mathsf{catchfail}(D)\ \mathsf{in}\ \mathsf{let}\ m = \mathrm{eval}\ \mathsf{letin}(x, M_1, M_2)\ \mathsf{in}\ \overline{c}\langle m\rangle$: if D succeeds, x has the same value as before, and $x \neq \mathsf{c_{fail}}$, so $\mathsf{letin}(x, M_1, M_2)$ returns M_1; if D fails, $x = \mathsf{c_{fail}}$ and $\mathsf{letin}(x, M_1, M_2)$ returns M_2. The destructor $\mathsf{notfail}$ inverts the status of a term evaluation: it fails if and only if its argument does not fail. This destructor will be used in the next section.

Example 6. Coming back to Example 5, the false attack occurs due to the following term evaluation:

$$\begin{array}{l}\mathsf{let}\ z = \mathsf{equals}(\mathsf{proj}_2(x), \mathsf{pk}(\mathsf{diff}[ska, ska'])) \ \mathsf{in}\\ \quad \overline{c}\langle \mathsf{aenc}(\langle n_i, \langle n_b, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(\mathsf{diff}[sk_a, sk_{a'}]))\rangle\\ \mathsf{else}\ \overline{c}\langle \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))\rangle\end{array}$$

We transform this term evaluation as explained above:

$$\mathsf{let}\ z = \mathsf{letin}(\mathsf{catchfail}(\mathsf{equals}(\mathsf{proj}_2(x), \mathsf{pk}(\mathsf{diff}[ska, ska']))), M, M')\ \mathsf{in}\ \overline{c}\langle z\rangle$$

where $M = \mathsf{aenc}(\langle n_i, \langle n_b, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(\mathsf{diff}[sk_a, sk_{a'}]))$, $M' = \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))$. Note that with $x = \langle n_i, \mathsf{pk}(sk_a)\rangle$ (see Example 5), if D is the term evaluation $D = \mathsf{letin}(\mathsf{catchfail}(\mathsf{equals}(\mathsf{proj}_2(x), \mathsf{pk}(\mathsf{diff}[ska, ska']))), M, M')$, we obtain that:

- $\mathsf{fst}(D) \Downarrow \mathsf{aenc}(\langle n_i, \langle n_b, \mathsf{pk}(sk_b)\rangle\rangle, \mathsf{pk}(sk_a))$
- $\mathsf{snd}(D) \Downarrow \mathsf{aenc}(n_b, \mathsf{pk}(sk_b))$

which corresponds to what $\mathsf{fst}(P_0)$ and $\mathsf{snd}(P_0)$ respectively output. Thanks to this, if we denote by P_0' our new biprocess, we obtain that $\mathsf{fst}(P_0) \approx \mathsf{fst}(P_0')$ and $\mathsf{snd}(P_0) \approx \mathsf{snd}(P_0')$. Furthermore, ProVerif will be able to prove that the biprocess P_0' satisfies equivalence, i.e., $\mathsf{fst}(P_0') \approx \mathsf{snd}(P_0')$ and so $\mathsf{fst}(P_0) \approx \mathsf{snd}(P_0)$.

The transformation proposed in the previous example can be generalised to term evaluations that perform actions other than just a single output. However, it is possible only if the success branch and the failure branch of the term evaluation both input and output the same number of terms. For example, the biprocess $\mathsf{let}\ x = D\ \mathsf{in}\ \overline{c}\langle M\rangle.\overline{c}\langle M'\rangle\ \mathsf{else}\ \overline{c}\langle N\rangle$ cannot be modified into a biprocess without else branch even with our new destructors. On the other hand, the success or failure of D can really be observed by the adversary, by tracking the number of outputs on the channel c, so the failure of the proof of equivalence corresponds to a real attack in this case.

5.2 Merging and Simplifying Biprocesses

To automatically detect and apply this transformation, we define two functions, *merge* and *simpl*. The function *merge*, defined in Fig. 4, is partial. It takes two biprocesses as arguments and detects if those two biprocesses can be merged into one biprocess. If the merging is possible, it returns the merged biprocess. This merged biprocess is expressed using a new operator branch, similar to diff: $\mathsf{branch}[D, D']$ is a term evaluation and we introduce functions fst' and snd' such that $\mathsf{fst}'(P)$ (resp. $\mathsf{snd}'(P)$) replaces each $\mathsf{branch}[D, D']$ with D (resp. D') in P.

Case (Mout) detects that both biprocesses output a message while case (Min) detects that both biprocesses input a message. We introduce a *let* for the channels and messages so that they can later be replaced by a term evaluation. Case (Mpar) uses the commutativity and associativity of parallel composition to increase the chances of success of *merge*. Cases (Mres) and (Mres′) use $Q \approx (\nu a)Q$ when $a \notin fn(Q)$ to allow merging processes even when a restriction occurs only on one side. Case (Mrepl2) is the basic merging of replicated processes, while Case (Mrepl1) allows merging $!!P$ with $!P'$ (case $n = 0$) because $!P \approx !!P$, and furthermore allows restrictions between the two replications, using $Q \approx (\nu a)Q$. Case (Mlet1) merges two processes that both contain term evaluations, by merging their success branches together and their failure branches together. On the other hand, Cases (Mlet2), (Mlet2′) also merge two processes that contain term evaluations, by merging the success branch of one process with the failure branch of the other process. Cases (Mlet3), (Mlet3′), (Mlet4), (Mlet4′) allow merging a term evaluation with another process P', by merging P' with either the success branch or the failure branch of the term evaluation. This merging is useful when ProVerif can prove that the resulting process satisfies equivalence, hence when both sides of the obtained *let* succeed simultaneously. Therefore, rule (Mlet3) is helpful when D always succeeds, and rule (Mlet4) when D always fails. When no such case applies, merging fails.

The function *simpl* is total. It takes one biprocess as argument and simplifies it by replacing all subprocesses of the form $\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ P'$, where $merge(P, P')$ succeeds, with

$$\mathsf{let}\ x = \mathrm{eval}\ \mathsf{catchfail}(D)\ \mathsf{in}\ Q\{^{\mathrm{eval}\ \mathsf{letin}(x,D_1,D_2)}/_{\mathsf{branch}[D_1,D_2]}\}\ \mathsf{else}\ 0$$

for some $Q = merge(P, P')$. This replacement is performed bottom up, so that P and P' have already been simplified when we transform $\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ P'$. The notation $Q\{^{\mathrm{eval}\ \mathsf{letin}(x,D_1,D_2)}/_{\mathsf{branch}[D_1,D_2]}\}$ means that we replace in Q every instance of $\mathsf{branch}[D_1, D_2]$, for some D_1, D_2, with $\mathrm{eval}\ \mathsf{letin}(x, D_1, D_2)$. The function *simpl* performs the transformation of term evaluations outlined in Section 5.1, when we can merge the success and failure branches.

Both functions are non-deterministic; the implementation may try all possibilities. In the current implementation of ProVerif, we apply the rules (Mlet3) and (Mlet4) only if the rules (Mlet1) and (Mlet2) are not applicable. Moreover, we never merge 0 with a process different from 0. This last restriction is crucial to reduce the number of biprocesses returned by *merge* and *simpl*. Typically, we avoid that 0 and $\mathsf{let}\ x = M\ \mathsf{in}\ P\ \mathsf{else}\ 0$ are merged by the rule (Mlet4).

$$merge(0,0) \stackrel{def}{=} 0 \tag{Mnil}$$

$$merge(\overline{M}\langle N\rangle.P, \overline{M'}\langle N'\rangle.P') \stackrel{def}{=} \mathsf{let}\ x = \mathsf{branch}[M, M']\ \mathsf{in}\ \mathsf{let}\ x' = \mathsf{branch}[N, N']\ \mathsf{in}\ \overline{x}\langle x'\rangle.merge(P, P') \tag{Mout}$$
where x and x' are fresh variables

$$merge(M(x).P, M'(x').P') \stackrel{def}{=} \mathsf{let}\ y = \mathsf{branch}[M, M']\ \mathsf{in}\ y(y').merge(P\{^{y'}/_{x}\}, P'\{^{y'}/_{x'}\}) \tag{Min}$$
where y and y' are fresh variables

$$merge(P_1 \mid \ldots \mid P_n, P'_1, \mid \ldots \mid P'_n) \stackrel{def}{=} Q_1 \mid \ldots \mid Q'_n \tag{Mpar}$$
if $(i_1, \ldots, i_n)$ is a permutation of $(1, \ldots, n)$
and for all $k \in \{1, \ldots, n\}$, $Q_k = merge(P_k, P'_{i_k})$

$$merge((\nu a)P, Q) \stackrel{def}{=} (\nu a) merge(P, Q) \tag{Mres}$$
after renaming a such that $a \notin fn(Q)$

$$merge(!\,(\nu a_1) \ldots (\nu a_n)!\,P, !\,P') \stackrel{def}{=} !\,(\nu a_1) \ldots (\nu a_n) merge(!\,P, !\,P') \tag{Mrepl1}$$
after renaming $a_1, \ldots, a_n$ such that $a_1, \ldots, a_n \notin fn(P')$

$$merge(!\,P, !\,P') \stackrel{def}{=} !\,merge(P, P') \tag{Mrepl2}$$
if there is no $P_1, a_1, \ldots, a_n$ such that $P = (\nu a_1) \ldots (\nu a_n)!\,P_1$
and no $P'_1, a'_1, \ldots, a'_m$ such that $P' = (\nu a'_1) \ldots (\nu a'_m)!\,P'_1$

$$merge(\mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2, \mathsf{let}\ x' = D'\ \mathsf{in}\ P'_1\ \mathsf{else}\ P'_2) \stackrel{def}{=} \mathsf{let}\ y = \mathsf{branch}[D, D']\ \mathsf{in}\ Q_1\ \mathsf{else}\ Q_2 \tag{Mlet1}$$
if y is a fresh variable, $Q_1 = merge(P_1\{^{y}/_{x}\}, P'_1\{^{y}/_{x'}\})$, and $Q_2 = merge(P_2, P'_2)$

$$merge(\mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2, \mathsf{let}\ x' = D'\ \mathsf{in}\ P'_1\ \mathsf{else}\ P'_2) \stackrel{def}{=} \mathsf{let}\ y = \mathsf{branch}[D, \mathsf{notfail}(D')]\ \mathsf{in}\ Q_1\ \mathsf{else}\ Q_2 \tag{Mlet2}$$
if y is a fresh variable, $x' \notin fv(P'_1)$, $Q_1 = merge(P_1\{^{y}/_{x}\}, P'_2)$, and $Q_2 = merge(P_2, P'_1)$

$$merge(\mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2, P') \stackrel{def}{=} \mathsf{let}\ y = \mathsf{branch}[D, \mathsf{c_{fail}}]\ \mathsf{in}\ Q\ \mathsf{else}\ P_2 \tag{Mlet3}$$
if y is a fresh variable and $Q = merge(P_1\{^{y}/_{x}\}, P')$

$$merge(\mathsf{let}\ x = D\ \mathsf{in}\ P_1\ \mathsf{else}\ P_2, P') \stackrel{def}{=} \mathsf{let}\ y = \mathsf{branch}[D, \mathsf{fail}]\ \mathsf{in}\ P_1\{^{y}/_{x}\}\ \mathsf{else}\ Q \tag{Mlet4}$$
if y is a fresh variable and $Q = merge(P_2, P')$

plus symmetric cases (Mres′), (Mrepl1′), (Mlet2′), (Mlet3′), and (Mlet4′) obtained from (Mres), (Mrepl1), (Mlet2), (Mlet3), and (Mlet4) by swapping the first and second arguments of *merge* and branch.

Fig. 4. Definition of the function *merge*

5.3 Results

Lemma 3 below shows that observational equivalence is preserved by the functions *merge* and *simpl*. In this lemma, we consider biprocesses P and P' that are not necessarily closed. We say that a context C is closing for P when $C[P]$ is closed. Moreover, given two biprocesses P and Q, we say that $P \approx Q$ if, and only if, $\mathsf{fst}(P) \approx \mathsf{fst}(Q)$ and $\mathsf{snd}(P) \approx \mathsf{snd}(Q)$.

Lemma 3. *Let P and P' be two biprocesses. If $merge(P, P') = Q$ then, for all contexts C closing for P, $C[P] \approx C[\mathsf{fst}'(Q)]$ and, for all contexts C closing for P', $C[P'] \approx C[\mathsf{snd}'(Q)]$. For all contexts C closing for P, $C[P] \approx C[simpl(P)]$.*

From the previous lemma, we can derive the two main results of this section.

Theorem 2. *Let P be a closed biprocess. If $simpl(P)$ satisfies observational equivalence, then $\mathsf{fst}(P) \approx \mathsf{snd}(P)$.*

From Theorem 2, we can extract our algorithm. Given a biprocess P as input, we compute $simpl(P)$. Since *simpl* is total but non-deterministic, we may have several biprocesses as result for $simpl(P)$. If ProVerif proves equivalence on at least one of them, then we conclude that $\mathsf{fst}(P) \approx \mathsf{snd}(P)$.

Theorem 3. *Let P and P' be two closed processes that do not contain* diff. *Let $Q = merge(simpl(P), simpl(P'))$. If the biprocess $Q\{^{\mathsf{diff}[D,D']}/_{\mathsf{branch}[D,D']}\}$ satisfies observational equivalence, then $P \approx P'$.*

The previous version of ProVerif could only take a biprocess as input. However, transforming two processes into a biprocess is usually not as easy as in the private authentication example. Theorem 3 automates this transformation.

6 Conclusion

In this paper, we have extended ProVerif with destructors defined by rewrite rules with inequalities as side-conditions. We have proposed a procedure relying on these new rewrite rules to automatically transform a biprocess into equivalent biprocesses on which ProVerif avoids the false attacks due to conditional branchings. Our extension is implemented in ProVerif, which is available at http://proverif.inria.fr. Experimentation showed that the automatic transformation of a biprocess is efficient and returns few biprocesses. In particular, our extension automatically proves anonymity as defined in [5] for the private authentication protocol for an unbounded number of sessions.

However, ProVerif is still unable to prove the unlinkability of the UK e-passport protocol [5] even though we managed to avoid some previously existing false attacks. This is a consequence of the matching by ProVerif of traces with the same scheduling in the two variants of the biprocesses. Thus we would like to relax the matching of traces, e.g., by modifying the replication identifiers on the left and right parts of biprocesses. This would allow us to prove even more equivalences with ProVerif and in particular the e-passport protocol.

Another direction for future research would be to define equations with inequalities as side-conditions. It may be possible to convert such equations into rewrite rules with side-conditions, like we convert equations into rewrite rules.

Acknowledgments. This work has been partially supported by the ANR projects PROSE (decision ANR 2010-VERS-004) and JCJC VIP n° 11 JS02 006 01, as well as the grant DIGITEO API from Région Île-de-France. It was partly done while the authors were at Ecole Normale Supérieure, Paris.

References

1. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL'01. pp. 104–115. ACM, New York (2001)
2. Abadi, M., Fournet, C.: Private authentication. Theoretical Computer Science 322(3), 427–476 (2004)
3. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Information and Computation 148(1), 1–70 (1999)
4. Arapinis, M., Cheval, V., Delaune, S.: Verifying privacy-type properties in a modular way. In: CSF'12. pp. 95–109. IEEE, Los Alamitos (2012)
5. Arapinis, M., Chothia, T., Ritter, E., Ryan, M.: Analysing unlinkability and anonymity using the applied pi calculus. In: CSF'10. pp. 107–121. IEEE, Los Alamitos (2010)
6. Baudet, M.: Sécurité des protocoles cryptographiques: aspects logiques et calculatoires. Ph.D. thesis, LSV, ENS Cachan (2007)
7. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. Journal of Logic and Algebraic Programming 75(1), 3–51 (2008)
8. Borgström, J., Briais, S., Nestmann, U.: Symbolic bisimulation in the spi calculus. In: Gardner, P., Yoshida, N. (eds.) CONCUR'04. LNCS, vol. 3170, pp. 161–176. Springer, Heidelberg (2004)
9. Cheval, V., Comon-Lundh, H., Delaune, S.: Trace equivalence decision: Negative tests and non-determinism. In: CCS'11. pp. 321–330. ACM, New York (2011)
10. Ciobâcă, Ş.: Automated Verification of Security Protocols with Applications to Electronic Voting. Ph.D. thesis, LSV, ENS Cachan, France (2011)
11. Comon-Lundh, H., Delaune, S.: The finite variant property: How to get rid of some algebraic properties. In: Giesl, J. (ed.) RTA'05. LNCS, vol. 3467, pp. 294–307. Springer, Heidelberg (2005)
12. Delaune, S., Kremer, S., Ryan, M.D.: Symbolic bisimulation for the applied picalculus. In: Arvind, V., Prasad, S. (eds.) FSTTCS'07. LNCS, vol. 4855, pp. 133–145. Springer, Heidelberg (2007)
13. Durante, L., Sisto, R., Valenzano, A.: Automatic testing equivalence verification of spi calculus specifications. ACM TOSEM 12(2), 222–284 (2003)
14. Hüttel, H.: Deciding framed bisimilarity. In: INFINITY'02. pp. 1–20 (2002)
15. Liu, J., Lin, H.: A complete symbolic bisimulation for full applied pi calculus. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., Rumpe, B. (eds.) SOFSEM'10. LNCS, vol. 5901, pp. 552–563. Springer, Heidelberg (2010)
16. Nicola, R.D., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34, 83–133 (1984)
17. Tiu, A., Dawson, J.E.: Automating open bisimulation checking for the spi calculus. In: CSF'10. pp. 307–321. IEEE, Los Alamitos (2010)