Automated Verification of Selected Equivalences for Security Protocols

Bruno Blanchet
CNRS, École Normale Supérieure, Paris

Martín Abadi
University of California, Santa Cruz

Cédric Fournet
Microsoft Research, Cambridge

Abstract

In the analysis of security protocols, methods and tools for reasoning about protocol behaviors have been quite effective. We aim to expand the scope of those methods and tools. We focus on proving equivalences $P \approx Q$ in which P and Q are two processes that differ only in the choice of some terms. These equivalences arise often in applications. We show how to treat them as predicates on the behaviors of a process that represents P and Q at the same time. We develop our techniques in the context of the applied pi calculus and implement them in the tool ProVerif.

1. Introduction

Many security properties can be expressed as predicates on system behaviors. These properties include some kinds of secrecy properties (for instance, "the system never broadcasts the key k"). They also include correspondence properties (for instance, "if the system deletes file f, then the administrator must have requested it"). Such predicates on system behaviors are the focus of many successful methods for security analysis. In recent years, several tools have made it possible to prove many such predicates automatically or semi-automatically, even for infinite-state systems.

Our goal in this work is to expand the scope of those methods and tools. We aim to apply them to important security properties that have been hard to prove and that cannot be easily phrased as predicates on system behaviors. Many such properties can be written as equivalences. For instance, the secrecy of a boolean parameter x of a protocol $P(x)$ may be written as the equivalence $P(\mathrm{true}) \approx P(\mathrm{false})$. Similarly, as is common in theoretical cryptography, we may wish to express the correctness of a construction P by comparing it to an ideal functionality Q, writing $P \approx Q$. Here the relation $\approx$ represents observational equivalence: $P \approx Q$ means that no context (that is, no attacker) can distinguish P and Q. A priori, $P \approx Q$ is not a simple predicate on the behaviors of P or Q.

We focus on proving equivalences $P \approx Q$ in which P and Q are two variants of the same process obtained by selecting different terms on the left and on the right. In particular, $P(\mathrm{true}) \approx P(\mathrm{false})$ is such an equivalence, since $P(\mathrm{true})$ and $P(\mathrm{false})$ differ only in the choice of value for the parameter x. Both $P(\mathrm{true})$ and $P(\mathrm{false})$ are variants of a process that we may write $P(\mathsf{diff}[\mathrm{true}, \mathrm{false}])$; the two variants are obtained by giving different interpretations to $\mathsf{diff}[\mathrm{true}, \mathrm{false}]$, making it select either true or false.

Although the notation diff can be viewed as a simple informal abbreviation, we find that there is some value in giving it a formal status. We define a calculus that supports diff. By carefully defining the operational semantics of this calculus, we can establish the equivalence $P(\mathrm{true}) \approx P(\mathrm{false})$ by reasoning about behaviors of $P(\mathsf{diff}[\mathrm{true}, \mathrm{false}])$.

In this operational semantics, $P(\mathsf{diff}[\mathrm{true}, \mathrm{false}])$ behaves like both $P(\mathrm{true})$ and $P(\mathrm{false})$ from the point of view of the attacker, as long as the attacker cannot distinguish $P(\mathrm{true})$ and $P(\mathrm{false})$. The semantics requires that the results of reducing $P(\mathrm{true})$ and $P(\mathrm{false})$ can be written as a process with subexpressions of the form $\mathsf{diff}[M_1, M_2]$. On the other hand, when $P(\mathrm{true})$ and $P(\mathrm{false})$ would do something that may differentiate them, the semantics specifies that the execution of $P(\mathsf{diff}[\mathrm{true}, \mathrm{false}])$ gets stuck. Hence, if no behavior of $P(\mathsf{diff}[\mathrm{true}, \mathrm{false}])$ ever gets stuck, then $P(\mathrm{true}) \approx P(\mathrm{false})$. Thus, we can prove equivalences by reasoning about behaviors, though not the behaviors of the original processes in isolation.

This technique applies not only to an equivalence $P(\mathrm{true}) \approx P(\mathrm{false})$ that represents the concealment of a boolean parameter, but to a much broader class of equivalences that arise in security analysis and that go beyond secrecy properties. In principle, every equivalence could be rewritten as an equivalence in our class: we might try to prove $P \approx Q$ by examining the behaviors of *if* $\mathsf{diff}[\mathrm{true}, \mathrm{false}] = \mathrm{true}$ *then* P *else* Q. This observation suggests that we should not expect completeness for an automatic technique. Accordingly, we are primarily concerned with soundness and usefulness, and (in contrast with some related work [5, 14–17, 20]) we emphasize simplicity and automation over generality. We believe, however, that the use of diff is not "just a hack", because diff is amenable to a rigorous treatment and because operators much like diff have already proved useful in other contexts—in particular, in elegant soundness proofs of information-flow type systems [22, 23].

We implement our technique in the tool ProVerif [11]. This tool is a protocol analyzer for protocols written in the applied pi calculus [4], an extension of the pi calculus with function symbols that may represent cryptographic operations. Internally, ProVerif translates protocols to Horn clauses in classical logic, and uses resolution on these clauses. We extend the translation into Horn clauses and also the manipulation of these Horn clauses.

While the implementation in ProVerif requires a nontrivial development of theory and code, it is rather fruitful. It enables us to treat, automatically, interesting proofs of equivalences. In particular, it does not require that all systems under consideration be finite-state. We demonstrate these points through small examples and larger applications.

Specifically, we use ProVerif for an infinite-state analysis of the important Encrypted Key Exchange (EKE) protocol [9, 10]. (Password-based protocols such as EKE have attracted much attention in recent years, partly because of the difficulty of reasoning about them.) We also use ProVerif for checking certain equivalences that express authenticity properties in an example from the literature [6]. In other applications, automated proofs of equivalences serve as lemmas for manual proofs of other results. We illustrate this combination by revisiting proofs for the JFK protocol [7]. The total runtime of all our proof scripts is 45 s on a Pentium M 1.8 GHz.

One of the main features of the approach presented in this paper is that it is compatible with the inclusion of equational theories on function symbols. We devote considerable attention to their proper, sound integration. Those equational theories serve in modelling properties of the underlying cryptographic operations; they are virtually necessary in many applications. For instance, an equational theory may describe a decryption function that returns "junk" when its input is not a ciphertext under the expected key. Without equational theories, we may be able to model decryption only as a destructor that fails when there is a mismatch between ciphertext and key. Because the failure of decryption would be observable, it can result in false indications of attacks. Our approach overcomes this problem.

In contrast, a recent method for proving equivalences with ProVerif [13] does not address equivalences that depend on equational theories. Moreover, that method applies only to pairs of processes in which the terms that differ are global constants, not arbitrary terms. In these respects, the approach presented in this paper constitutes a clear advance. It enables significant proofs that were previously beyond the reach of automated techniques.

The next section describes the process calculus that serves as setting for this work. Section 3 defines and studies observational equivalence. Section 4 contains some examples. Section 5 deals with equational theories. Section 6 explains how ProVerif maps protocols with diff to Horn clauses. Section 7 describes applications. These applications constitute a substantial part of our work; the brevity of their presentation is due only to space constraints. Section 8 mentions other related work and concludes.

$M, N ::=$	terms
$\quad x, y, z$	variable
$\quad a, b, c, k, s$	name
$\quad f(M_1, \ldots, M_n)$	constructor application
$D ::=$	term evaluations
$\quad M$	term
$\quad$ eval $h(D_1, \ldots, D_n)$	function evaluation
$P, Q, R ::=$	processes
$\quad M(x).P$	input
$\quad \overline{M}\langle N \rangle.P$	output
$\quad \mathbf{0}$	nil
$\quad P \mid Q$	parallel composition
$\quad !P$	replication
$\quad (\nu a)P$	restriction
$\quad let\ x = D\ in\ P\ else\ Q$	term evaluation

Figure 1. Syntax for terms and processes

An extended version of this paper, the tool ProVerif, and the proof scripts for all examples and applications are available at `http://www.di.ens.fr/~blanchet/obsequi/`.

2. The process calculus

This section introduces our process calculus, by giving its syntax and its operational semantics. This calculus is a combination of the original applied pi calculus [4] with one of its dialects [13]. This choice of calculus gives us the richness of the original applied pi calculus (in particular, with equational theories) while enabling us to leverage ProVerif.

Syntax and informal semantics Figure 1 summarizes the syntax of our calculus. It defines a category of terms (data) and processes (programs). It assumes an infinite set of names and an infinite set of variables; a, b, c, k, s, and similar identifiers range over names, and x, y, and z range over variables. It also assumes a signature Σ (a set of function symbols, with arities and with associated definitions as explained below). We distinguish two categories of function symbols: constructors and destructors. We often write f for a constructor, g for a destructor, and h for a constructor or a destructor. Constructors are used for building terms. Thus, the terms $M, N, \ldots$ are variables, names, and constructor applications of the form $f(M_1, \ldots, M_n)$.

As in the applied pi calculus [4], terms are subject to an equational theory. Identifying an equational theory with its signature Σ, we write $\Sigma \vdash M = N$ for an equality modulo the equational theory, and $\Sigma \vdash M \neq N$ an inequality modulo the equational theory. (We write $M = N$ and $M \neq N$ for syntactic equality and inequality, respectively.) The equational theory is defined by a finite set of equations $\Sigma \vdash M_i = N_i$, where M_i and N_i are terms that contain only constructors and variables. The equational theory is then obtained from this set of equations by reflexive, symmetric, and transitive closure, closure by substitution (for any substitution σ, if $\Sigma \vdash M = N$ then $\Sigma \vdash \sigma M = \sigma N$), and closure by context application (if $\Sigma \vdash M = N$ then $\Sigma \vdash M'\{M/x\} = M'\{N/x\}$, where $\{M/x\}$ is the substitution that replaces x with M). We assume that there exist M and N such that $\Sigma \vdash M \neq N$.

As in [13], destructors are partial, non-deterministic operations on terms that processes can apply. More precisely, the semantics of a destructor g of arity n is given by a finite set $\mathrm{def}_\Sigma(g)$ of rewrite rules $g(M'_1, \ldots, M'_n) \rightarrow M'$, where $M'_1, \ldots, M'_n, M'$ are terms that contain only constructors and variables, the variables of M' are bound in $M'_1, \ldots, M'_n$, and variables are subject to renaming. Then $g(M_1, \ldots, M_n)$ is defined if and only if there exists a substitution σ and a rewrite rule $g(M'_1, \ldots, M'_n) \rightarrow M'$ in $\mathrm{def}_\Sigma(g)$ such that $M_i = \sigma M'_i$ for all $i \in \{1, \ldots, n\}$, and in this case $g(M_1, \ldots, M_n) \rightarrow \sigma M'$. In order to make function evaluation more uniform, we define $\mathrm{def}_\Sigma(f)$ as $\{f(x_1, \ldots, x_n) \rightarrow f(x_1, \ldots, x_n)\}$ when f is a constructor of arity n.

The process $let\ x = D\ in\ P\ else\ Q$ tries to evaluate D; if this succeeds, then x is bound to the result and P is executed, else Q is executed. Here one may ignore the prefix eval which may occur in D, since eval f and f have the same semantics when f is a constructor, and destructors are used only with eval. In Section 5, we distinguish eval f and f in order to keep track of when terms are evaluated.

Using constructors, destructors, and equations, we can model various data structures (tuples, lists, ...) and cryptographic primitives (shared-key encryption, public-key encryption, signatures, ...). Typically, destructors represent primitives that can visibly succeed or fail, while equations apply to primitives that always succeed but may sometimes return "junk". For instance, suppose that one can detect whether shared-key decryption succeeds or fails; then we would use a constructor enc, a destructor dec, and the rewrite rule $dec(enc(x, y), y) \rightarrow x$. Otherwise, we would use two constructors enc and dec, and the equations $dec(enc(x, y), y) = x$ and $enc(dec(x, y), y) = x$. (The second equation, which is not standard, aims to prevent that the equality test $enc(dec(M, N), N) = M$ reveal that M must be a ciphertext under N.) We refer to previous work [4, 13] for additional explanations and examples.

The rest of the syntax of Figure 1 is fairly standard pi calculus. The input process $M(x).P$ inputs a message on channel M, and executes P with x bound to the input message. The output process $\overline{M}\langle N\rangle.P$ outputs the message N on channel M and then executes P. (We allow M to be an arbitrary term; we could require that M be a name, as is frequently done, and adapt other definitions accordingly.) The nil process $\mathbf{0}$ does nothing. The process $P \mid Q$ is the parallel composition of P and Q. The replication $!P$ represents an unbounded number of copies of P in parallel. The restriction $(\nu a)P$ creates a new name a, and then executes P. The syntax does not include the conditional $if\ M = N\ then\ P\ else\ Q$, which can be defined as $let\ x = \mathrm{equals}(M, N)\ in\ P\ else\ Q$ where x is a fresh variable and equals is a destructor with the rewrite rule $\mathrm{equals}(x, x) \rightarrow x$.

We write $fn(P)$ and $fv(P)$ for the sets of names and variables free in P, respectively, which are defined as usual. A process is closed if it has no free variables; it may have free names. We identify processes up to renaming of bound names and variables. An evaluation context C is a closed context built from $[\,]$, $C \mid P$, $P \mid C$, and $(\nu a)C$.

$M \Downarrow M$
eval $h(D_1, \ldots, D_n) \Downarrow \sigma N$
  if $h(N_1, \ldots, N_n) \rightarrow N \in \mathrm{def}_\Sigma(h)$,
  and σ is such that for all i, $D_i \Downarrow M_i$ and $\Sigma \vdash M_i = \sigma N_i$

$P \mid 0 \equiv P$	$P \equiv P$
$P \mid Q \equiv Q \mid P$	$Q \equiv P \ \Rightarrow\ P \equiv Q$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	$P \equiv Q,\ Q \equiv R \ \Rightarrow\ P \equiv R$
$(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$	$P \equiv Q \ \Rightarrow\ P \mid R \equiv Q \mid R$
$(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q$ if $a \notin fn(P)$	$P \equiv Q \ \Rightarrow\ (\nu a)P \equiv (\nu a)Q$

$\overline{N}\langle M\rangle.Q \mid N'(x).P \ \rightarrow\ Q \mid P\{M/x\}$
  if $\Sigma \vdash N = N'$ (Red I/O)

$let\ x = D\ in\ P\ else\ Q \rightarrow P\{M/x\}$
  if $D \Downarrow M$ (Red Fun 1)

$let\ x = D\ in\ P\ else\ Q \rightarrow Q$
  if there is no M such that $D \Downarrow M$ (Red Fun 2)

$!P \ \rightarrow\ P \mid !P$ (Red Repl)
$P \ \rightarrow\ Q \ \Rightarrow\ P \mid R \ \rightarrow\ Q \mid R$ (Red Par)
$P \ \rightarrow\ Q \ \Rightarrow\ (\nu a)P \ \rightarrow\ (\nu a)Q$ (Red Res)
$P' \equiv P,\ P \ \rightarrow\ Q,\ Q \equiv Q' \ \Rightarrow\ P' \ \rightarrow\ Q'$ (Red $\equiv$)

Figure 2. Semantics for terms and processes

Formal semantics The rules of Figure 2 axiomatize the reduction relation for processes ($\rightarrow_\Sigma$), thus defining the operational semantics of our calculus. Auxiliary rules define term evaluation ($\Downarrow_\Sigma$) and the structural congruence relation ($\equiv$); this relation is useful for transforming processes so that the reduction rules can be applied. Both $\equiv$ and $\rightarrow_\Sigma$ are defined only on closed processes. We write $\rightarrow^*_\Sigma\equiv$ for the reflexive and transitive closure of $\rightarrow_\Sigma$ union $\equiv$. When Σ is clear from the context, we abbreviate $\rightarrow_\Sigma$ and $\Downarrow_\Sigma$ to $\rightarrow$ and $\Downarrow$, respectively.

3. Observational equivalence

In this section we introduce diff formally and establish a sufficient condition for observational equivalence. We first recall the standard definition of observational equivalence from the pi calculus:

Definition 1 (Equivalence) The process P *emits on* M ($P \downarrow_M$) if and only if $P \equiv C[\overline{M'}\langle N\rangle.R]$ for some evaluation context C that does not bind $fn(M)$ and $\Sigma \vdash M = M'$. *(Strong) observational equivalence* $\sim$ is the largest symmetric relation $\mathcal{R}$ on closed processes such that $P \mathrel{\mathcal{R}} Q$ implies

1. if $P \downarrow_M$ then $Q \downarrow_M$;
2. if $P \rightarrow P'$ then $Q \rightarrow Q'$ and $P' \mathrel{\mathcal{R}} Q'$ for some Q';
3. $C[P] \mathrel{\mathcal{R}} C[Q]$ for all evaluation contexts C.

Weak observational equivalence $\approx$ is defined similarly, with $\rightarrow^* \downarrow_M$ instead of $\downarrow_M$ and $\rightarrow^*$ instead of $\rightarrow$.

Intuitively, a context may represent an adversary, and two processes are observationally equivalent when no adversary can distinguish them.

Next we introduce a new calculus that can represent pairs of processes that have the same structure and differ only by the terms and term evaluations that they contain. We call such a pair of processes a *biprocess*. The grammar for the calculus is a simple extension of the grammar of Figure 1, with additional cases so that $\mathsf{diff}[M, M']$ is a term and $\mathsf{diff}[D, D']$ is a term evaluation. We also extend the definition of contexts to permit the use of diff, and sometimes refer to contexts without diff as plain contexts.

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. Our goal is to show that the processes $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$ are observationally equivalent:

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

Figure 3 gives the semantics of biprocesses. Reductions for biprocesses bundle those for processes: if $P \rightarrow Q$ then $\mathsf{fst}(P) \rightarrow \mathsf{fst}(Q)$ and $\mathsf{snd}(P) \rightarrow \mathsf{snd}(Q)$. Conversely, however, reductions in $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$ need not correspond to any biprocess reduction, in particular when they do not match up. Our first theorem shows that the processes are equivalent when this does not happen.

Definition 3 We say that the biprocess P is *uniform* when $\mathsf{fst}(P) \rightarrow Q_1$ implies that $P \rightarrow Q$ for some biprocess Q with $\mathsf{fst}(Q) \equiv Q_1$, and symmetrically for $\mathsf{snd}(P) \rightarrow Q_2$.

The semantics for biprocesses is defined as in Figure 2 with generalized rules for I/O and Fun:

$\overline{N}\langle M\rangle.Q \mid N'(x).P \;\rightarrow\; Q \mid P\{M/x\}$ (Red I/O)
  if $\Sigma \vdash \mathsf{fst}(N) = \mathsf{fst}(N')$ and $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$

$let\ x = D\ in\ P\ else\ Q \rightarrow P\{\mathsf{diff}[M_1, M_2]/x\}$
  if $\mathsf{fst}(D) \Downarrow M_1$ and $\mathsf{snd}(D) \Downarrow M_2$ (Red Fun 1)
$let\ x = D\ in\ P\ else\ Q \rightarrow Q$
  if there is no M_1 such that $\mathsf{fst}(D) \Downarrow M_1$ and
  there is no M_2 such that $\mathsf{snd}(D) \Downarrow M_2$ (Red Fun 2)

Figure 3. Semantics for biprocesses

Theorem 1 *Let P_0 be a closed biprocess. If, for all plain evaluation contexts C and reductions $C[P_0] \rightarrow^* P$, the biprocess P is uniform, then P_0 satisfies observational equivalence.*

Our plan is to establish the hypothesis of Theorem 1 by automatically verifying that all the biprocesses P meet conditions that imply uniformity. The next corollary details those conditions, which guarantee that a communication and an evaluation, respectively, succeed in $\mathsf{fst}(P)$ if and only if they succeed in $\mathsf{snd}(P)$:

Corollary 1 *Let P_0 be a closed biprocess. Suppose that, for all plain evaluation contexts C, all evaluation contexts C', and all reductions $C[P_0] \rightarrow^* P$,*

1. *if $P \equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$, then $\Sigma \vdash \mathsf{fst}(N) = \mathsf{fst}(N')$ if and only if $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$,*
2. *if $P \equiv C'[let\ x = D\ in\ Q\ else\ R]$, then there exists M_1 such that $\mathsf{fst}(D) \Downarrow M_1$ if and only if there exists M_2 such that $\mathsf{snd}(D) \Downarrow M_2$.*

Then P_0 satisfies observational equivalence.

4. Examples in the applied pi calculus

This section illustrates our approach by revisiting examples of observational equivalences presented with the applied pi calculus [4]. Interestingly, all those equivalences can be formulated using biprocesses, proved via Theorem 1 and, it turns out, verified automatically by ProVerif. Section 7 sketches more complex examples.

Example 1 We begin with an equivalence expressed with a biprocess that performs a single output, of the form $(\nu a_1, \ldots, a_k)\overline{c}\langle M\rangle$ where c is a name that does not occur in $a_1, \ldots, a_k$ or in M. Intuitively, such equivalences state that no environment can differentiate $\mathsf{fst}(M)$ from $\mathsf{snd}(M)$ without knowing some name in a_1, …, a_k. Such equivalences on terms under restrictions are called static equivalences [4]. They play a central role in the extension of proof techniques from the pure pi calculus to the applied pi calculus.

This example concerns the Diffie-Hellman computations used in key agreement protocols. These computations can be expressed in terms of a constant g, a binary constructor ^, and the equation $(\mathtt{g}\hat{\ }x)\hat{\ }y = (\mathtt{g}\hat{\ }y)\hat{\ }x$ [3,4]. With this signature, we verify that

$$(\nu a_1, a_2, a_3)\overline{c}\langle(\mathtt{g}\hat{\ }a_1, \mathtt{g}\hat{\ }a_2, \mathsf{diff}[(\mathtt{g}\hat{\ }a_1)\hat{\ }a_2, \mathtt{g}\hat{\ }a_3])\rangle$$

satisfies equivalence. This equivalence closely corresponds to the Decisional Diffie Hellman assumption often made by cryptographers; it is also the main lemma in the proof of [4, Theorem 3]. Intuitively, even if the environment is given access to the exponentials $\mathtt{g}\hat{\ }a_1$ and $\mathtt{g}\hat{\ }a_2$, those values are (apparently) unrelated to the Diffie-Hellman secret $(\mathtt{g}\hat{\ }a_1)\hat{\ }a_2$, since the environment cannot distinguish this secret from the exponential of any fresh unrelated value a_3.

Example 2 Non-deterministic encryption is a variant of public-key encryption that further protects the secrecy of the plaintext by embedding some additional, fresh value in each encryption. It can be modelled using three functions for public-key decryption, public-key encryption, and public-key derivation, linked by the equation $dec(enc(x, pk(y), z), y) = x$, where z is the additional parameter for the encryption. A key property of non-deterministic encryption is that, without knowledge of the decryption key, ciphertexts appear to be unrelated to the plaintexts, even if the attacker knows the plaintexts and the encryption key. To express this property, we state that

$$(\nu s)(\overline{c}\langle pk(s)\rangle \mid !c'(x).(\nu a)\overline{c}\langle \mathsf{diff}[enc(x, pk(s), a), a]\rangle)$$

satisfies equivalence. This biprocess performs a first output to reveal the public key $pk(s)$ (but not s!), then repeatedly inputs a term x from the environment and either outputs its encryption under $pk(s)$ or outputs a fresh, unrelated name.

Example 3 Biprocesses can also be used for relating an abstract specification of a cryptographic primitive with its implementation in terms of lower-level functions. As an example, we consider the construction of message authentication codes (MACs) for messages of arbitrary lengths from block ciphers, as modelled in the applied pi calculus [4, Section 6]—we refer to that work for additional details and discussion. The usage of MACs was captured via a little protocol that generates MACs on demand and checks them:

$$\begin{aligned} P_0 = (\nu k)(&!c'(x).\overline{c}\langle x, mac(k, x)\rangle \\ &| c(x, y).\mathit{if}\ y = mac(k, x)\ \mathit{then}\ \overline{c''}\langle x\rangle) \end{aligned}$$

Let P be P_0 with $\mathsf{diff}[mac(k, x), impl(k, x)]$ instead of the two occurrences of $mac(k, x)$. For a given signature with no equation for mac, a function $impl$ correctly implements mac when P satisfies equivalence. With this formulation, we can verify the correctness of the second construction considered in [4], $impl(k, x) = f(k, f(k, x))$ where f is a keyed hash function with equation $f(k, (x, y)) = h(f(k, x), y)$. We can also confirm that the first construction considered in [4] is subject to an attack.

5. Modelling equations with rewrite rules

We handle equations by translating from a signature with equations to a signature without equations. This translation is designed to ease implementation: with it, resolution can continue to rely on ordinary syntactic unification, and remains very efficient. Although our technique is general and automatic, it does have limitations: it does not apply to some equational theories, in particular theories with associative symbols such as XOR. (It may be possible to handle some of those theories by shifting from syntactic unification to unification modulo the theory in question, at the cost of increased complexity.)

We consider an auxiliary rewriting system on terms, $\mathcal{S}$, that defines partial normal forms. The terms manipulated by $\mathcal{S}$ do not contain diff, but they may contain variables. The rules of $\mathcal{S}$ do not contain names and do not have a single variable on the left-hand side. We say that the set of terms $\mathcal{M}$ is in normal form relatively to $\mathcal{S}$ and Σ, and write $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M})$, if and only if all terms of $\mathcal{M}$ are irreducible by $\mathcal{S}$ and, for all N_1, N_2 subterms of terms of $\mathcal{M}$, if $\Sigma \vdash N_1 = N_2$ then $N_1 = N_2$. Intuitively, we require that $\mathcal{M}$ use irreducible forms consistently, allowing for the possibility that terms may have several irreducible forms (see Example 5 below). We extend this definition to sets of processes: $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P})$ if and only if the set of terms appearing in processes in $\mathcal{P}$ is in normal form.

For a signature Σ', we define evaluation on open terms as a relation $D \Downarrow' (M, \sigma)$, where σ collects instantiations of D obtained by unification:

$M \Downarrow' (M, \emptyset)$
eval $h(D_1, \dots, D_n) \Downarrow' (\sigma_u N, \sigma_u \sigma')$
  if $(D_1, \dots, D_n) \Downarrow' ((M_1, \dots, M_n), \sigma')$,
  $h(N_1, \dots, N_n) \rightarrow N$ is in $\mathrm{def}_{\Sigma'}(h)$ and
  σ_u is a most general unifier of $(M_1, N_1), \dots, (M_n, N_n)$
$(D_1, \dots, D_n) \Downarrow' ((\sigma_n M_1, \dots, \sigma_n M_{n-1}, M_n), \sigma_n \sigma)$
  if $(D_1, \dots, D_{n-1}) \Downarrow' ((M_1, \dots, M_{n-1}), \sigma)$
  and $\sigma D_n \Downarrow' (M_n, \sigma_n)$

We let $\mathrm{addeval}(M_1, \dots, M_n)$ be the tuple of term evaluations obtained by adding eval before each function symbol of $M_1, \dots, M_n$. Using these definitions, we describe when a signature Σ' with rewrite rules models another signature Σ with equations:

Definition 4 Let Σ and Σ' be signatures on the same function symbols. We say that Σ' models Σ if and only if

1. The equational theory of Σ' is syntactic equality: $\Sigma' \vdash M = N$ if and only if $M = N$.

2. The constructors of Σ' are the constructors of Σ; their definition $\mathrm{def}_{\Sigma'}(f)$ contains the rule $f(x_1, \ldots, x_n) \rightarrow f(x_1, \ldots, x_n)$, but perhaps also other rules such that there exists a rewriting system $\mathcal{S}$ on terms that satisfies the following properties:

   S1. If $M \rightarrow N$ is in $\mathcal{S}$, then $\Sigma \vdash M = N$.

   S2. For any $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M})$ and term M, there exists M' such that $\Sigma \vdash M' = M$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$.

   S3. If $f(N_1, \ldots, N_n) \rightarrow N$ is in $\mathrm{def}_{\Sigma'}(f)$, then $\Sigma \vdash f(N_1, \ldots, N_n) = N$.

   S4. If $\Sigma \vdash f(M_1, \ldots, M_n) = M$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M_1, \ldots, M_n, M\})$, then there exist σ and $f(N_1, \ldots, N_n) \rightarrow N$ in $\mathrm{def}_{\Sigma'}(f)$ such that $M = \sigma N$ and $M_i = \sigma N_i$ for all $i \in \{1, \ldots, n\}$.

3. The destructors of Σ' are the destructors of Σ, with a rewrite rule $g(M'_1, \ldots, M'_n) \rightarrow M'$ in $\mathrm{def}_{\Sigma'}(g)$ for each $g(M_1, \ldots, M_n) \rightarrow M$ in $\mathrm{def}_{\Sigma}(g)$ and each $\mathrm{addeval}(M_1, \ldots, M_n, M) \Downarrow' ((M'_1, \ldots, M'_n, M'), \sigma)$.

Properties S1 and S2 concern the definition of $\mathcal{S}$: S1 guarantees that all rewrite rules of $\mathcal{S}$ are acceptable (that is, they do not generate terms different modulo Σ), and S2 guarantees that there are enough normal forms relatively to $\mathcal{S}$ and Σ. Properties S3 and S4 concern the definition of constructors in Σ': S3 guarantees that their definition contains only acceptable rewrite rules, and S4 that it contains enough rewrite rules: essentially, that when $M_1, \ldots, M_n$ are in normal form, all normal forms of $f(M_1, \ldots, M_n)$ can be generated by applying the rewrite rules of f in Σ' to $f(M_1, \ldots, M_n)$. Hence we deal with any equations on f in Σ by evaluating f once in Σ'. (We use eval markers in expressions accordingly: $\mathrm{eval}\, f$ and f represent f before and after this evaluation, respectively.) The definition of destructors in Σ' is computed by applying the rewrite rules of constructors in Σ' to the definition of destructors in Σ.

We have designed and implemented semi-algorithms for finding a rewriting system $\mathcal{S}$ and a signature Σ' such that Σ' models Σ. We leave them for the full paper because of space constraints. Here we show only two examples that ProVerif handles automatically.

Example 4 Suppose that Σ has the constructors enc and dec with the equations $dec(enc(x, y), y) = x$ and $enc(dec(x, y), y) = x$. In Σ', we adopt the rewrite rules:

$$\begin{array}{ll} dec(x,y) \rightarrow dec(x,y) & enc(x,y) \rightarrow enc(x,y) \\ dec(enc(x,y),y) \rightarrow x & enc(dec(x,y),y) \rightarrow x \end{array}$$

We have that Σ' models Σ for the rewriting system $\mathcal{S}$ with rules $dec(enc(x, y), y) \rightarrow x$ and $enc(dec(x, y), y) \rightarrow x$, and a single normal form for every term.

Example 5 In order to model the Diffie-Hellman equation of Example 1, we define Σ' with three rewrite rules:

$$\mathsf{g} \rightarrow \mathsf{g} \qquad x\hat{\ }y \rightarrow x\hat{\ }y \qquad (\mathsf{g}\hat{\ }x)\hat{\ }y \rightarrow (\mathsf{g}\hat{\ }y)\hat{\ }x$$

and use an empty $\mathcal{S}$. Intuitively, applying $\hat{\ }$ to $(\mathsf{g}\hat{\ }x)$ and y yields both possible forms of $(\mathsf{g}\hat{\ }x)\hat{\ }y$ modulo the equational theory, $(\mathsf{g}\hat{\ }x)\hat{\ }y$ and $(\mathsf{g}\hat{\ }y)\hat{\ }x$. Hence, a term M may have several irreducible forms M' that satisfy $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M'\})$ and $\Sigma \vdash M' = M$.

From this point on, we assume that Σ' models Σ. We extend equality modulo Σ to biprocesses: $\Sigma \vdash P = P'$ if and only if P' can be obtained from P by replacing some of its subterms M (not containing diff or eval) with subterms equal modulo Σ. We define $P \rightarrow_{\Sigma',\Sigma} P'$ as $P \rightarrow_{\Sigma} P'$ except that signature Σ' is used for reduction rules (Red I/O) and (Red Fun 1)—signature Σ is still used for (Red Fun 2).

We say that a biprocess P_0 is unevaluated when every term in P_0 is either a variable or $\mathsf{diff}[a, a]$ for some name a. Hence, every function symbol in P_0 must be in a term evaluation and prefixed by eval. For any biprocess P, we can build an unevaluated biprocess $\mathrm{unevaluated}(P)$ by introducing a term evaluation for every non-trivial term and a diff for every name (with $P \approx \mathrm{unevaluated}(P)$).

Lemma 1 *Let P_0 be a closed, unevaluated biprocess. If $P_0 \rightarrow^*_{\Sigma}\equiv P'_0$, $\Sigma \vdash P'_0 = P'$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{P'\})$, then $P_0 \rightarrow^*_{\Sigma',\Sigma}\equiv P'$. Conversely, if $P_0 \rightarrow^*_{\Sigma',\Sigma}\equiv P'$ then there exists P'_0 such that $\Sigma \vdash P'_0 = P'$ and $P_0 \rightarrow^*_{\Sigma}\equiv P'_0$.*

This lemma gives an operational correspondence between $\rightarrow_{\Sigma}$ and $\rightarrow_{\Sigma',\Sigma}$. A similar lemma holds for processes instead of biprocesses, and can be used for extending previous proof techniques for secrecy [2] and correspondence [12] properties, so that they apply under equational theories. These extensions are implemented in ProVerif. We do not detail them further since we focus on equivalences in this paper. Using Lemma 1, we obtain:

Lemma 2 *Let P_0 be a closed biprocess. P_0 satisfies the conditions of Corollary 1 if and only if, for all plain evaluation contexts C, all evaluation contexts C', and all reductions $\mathrm{unevaluated}(C[P_0]) \rightarrow^*_{\Sigma',\Sigma} P$, we have*

1. *if $P \equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ and $\mathsf{fst}(N) = \mathsf{fst}(N')$, then $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$,*

2. *if $P \equiv C'[let\ x = D\ in\ Q\ else\ R]$ and $\mathsf{fst}(D) \Downarrow_{\Sigma'} M_1$ for some M_1, then $\mathsf{snd}(D) \Downarrow_{\Sigma} M_2$ for some M_2,*

as well as the symmetric properties where we swap fst and snd.

6. Clause generation

Given a closed biprocess P_0, our protocol verifier builds a set of Horn clauses. This section explains the generation of the clauses, substantially extending to biprocesses previous work at the level of processes.

Patterns and facts In the clauses, the terms of processes are represented by patterns, with the following grammar:

$p ::=$	patterns
$\quad x, y, z, i$	variable
$\quad f(p_1, \ldots, p_n)$	constructor application
$\quad a[p_1, \ldots, p_n]$	name
$\quad \mathsf{g}$	element of $GVar$

We assign a distinct, fresh session identifier variable i to each replication of P_0. (We will use a distinct value for i for each copy of the replicated process.) We assign a pattern $a[p_1, \ldots, p_n]$ to each name a of P_0. We treat a as a function symbol, and write $a[p_1, \ldots, p_n]$ rather than $a(p_1, \ldots, p_n)$ only for clarity. We sometimes write a for $a[\,]$. If a is a free name, then its pattern is $a[\,]$. If a is bound by a restriction $(\nu a)P$ in P_0, then its pattern takes as arguments the terms received as inputs, the results of term evaluations, and the session identifiers of replications in the context that encloses the restriction. 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 $c'(x)$. We assume that each restriction $(\nu a)P$ in P_0 has a different name a, distinct from any free name of P_0. Moreover, session identifiers enable us to distinguish names created in different copies of processes. Hence, each name created in the process calculus is represented by a different pattern in the verifier.

Patterns include an infinite set of constants $GVar$. These constants are basically universally quantified variables, and occur only in arguments of the predicate nounif, defined below. We write $GVar(M)$ for the term obtained from M by replacing variables with new constants in the set $GVar$.

Clauses are built from the following predicates:

$F ::=$	facts
$\quad \mathrm{att}'(p, p')$	attacker knowledge
$\quad \mathrm{msg}'(p_1, p_2, p'_1, p'_2)$	channel messages
$\quad \mathrm{input}'(p, p')$	input on p (resp. p')
$\quad \mathrm{nounif}(p, p')$	impossible unification
$\quad \mathrm{bad}$	bad

Informally, $\mathrm{att}'(p, p')$ means that the attacker may obtain p in $\mathsf{fst}(P)$ and p' in $\mathsf{snd}(P)$ by the same operations; $\mathrm{msg}'(p_1, p_2, p'_1, p'_2)$ means that message p_2 may appear on channel p_1 in $\mathsf{fst}(P)$ and that message p'_2 may appear on channel p'_1 in $\mathsf{snd}(P)$ after the same reductions; $\mathrm{input}'(p, p')$ means that an input may be executed on channel p in $\mathsf{fst}(P)$ and on channel p' in $\mathsf{snd}(P)$, thus enabling the attacker to infer whether p (resp. p') is equal to another channel used for output; $\mathrm{nounif}(p, p')$ means that p and p' cannot be unified modulo Σ by substituting elements of $GVar$ with patterns; finally, bad serves in detecting violations of observational equivalence: when bad is not derivable, we have observational equivalence.

An evident difference with respect to previous translations from processes to clauses is that predicates have twice as many arguments: we use the binary predicate att' instead of the unary one att and the 4-ary predicate msg' instead of the binary one msg. This extension allows us to represent information for both versions of a biprocess.

The predicate nounif is not defined by clauses, but by special simplification steps in the solver. These steps are such that, for all closed patterns p and p', $\mathrm{nounif}(p, p')$ holds if and only if there is no closed substitution σ with domain $GVar$ such that $\Sigma \vdash \sigma p = \sigma p'$.

Clauses for the attacker The following clauses represent the capabilities of the attacker:

For each $a \in fn(P_0)$, $\mathrm{att}'(a[\,], a[\,])$ (Init)

For some b that does not occur in P_0,
$\mathrm{att}'(b[x], b[x])$ (Rn)

For each function h, for each pair of rewrite rules
$h(M_1, \ldots, M_n) \to M$ and $h(M'_1, \ldots, M'_n) \to M'$
in $\mathrm{def}_{\Sigma'}(h)$ (after renaming of variables),
$\mathrm{att}'(M_1, M'_1) \wedge \ldots \wedge \mathrm{att}'(M_n, M'_n) \to \mathrm{att}'(M, M')$ (Rf)

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

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

$\mathrm{att}'(x, x') \to \mathrm{input}'(x, x')$ (Ri)

$\mathrm{input}'(x, x') \wedge \mathrm{msg}'(x, z, y', z') \wedge \mathrm{nounif}(x', y') \to \mathrm{bad}$ (Rcom)

For each destructor g,
for each rewrite rule $g(M_1, \ldots, M_n) \to M$ in $\mathrm{def}_{\Sigma'}(g)$,

$$\bigwedge_{g(M'_1, \ldots, M'_n) \to M' \text{ in } \mathrm{def}_{\Sigma'}(g)} \mathrm{nounif}((x'_1, \ldots, x'_n), GVar((M'_1, \ldots, M'_n)))$$
$$\wedge\, \mathrm{att}'(M_1, x'_1) \wedge \ldots \wedge \mathrm{att}'(M_n, x'_n) \to \mathrm{bad} \qquad \text{(Rt)}$$

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

Clause (Ri) means that, if the attacker has x (resp. x'), then it can initiate an input on x (resp. x'), thereby testing whether it is equal to some other channel used for output. Clauses (Rcom) and (Rcom') detect when a communication can be done in one version of the biprocess and not in the other: the input and output channels are equal in one version and different in the other. These clauses check that condition 1 of Lemma 2 and its symmetric are true.

Clause (Rt) checks that for all applications of a destructor g, if this application succeeds in $\mathsf{fst}(P)$, then it succeeds in $\mathsf{snd}(P)$, possibly using another rule. Clause (Rt') checks the converse. These two clauses are essential for obtaining condition 2 of Lemma 2. Consider, for instance, the destructor equals. After a minor simplification, clause (Rt) becomes

$$\mathrm{att}'(x, y) \wedge \mathrm{att}'(x, y') \wedge \mathrm{nounif}(y, y') \rightarrow \mathrm{bad}$$

The other clauses are adapted from previous work [2, 12] by replacing unary (resp. binary) predicates with binary (resp. 4-ary) ones. Clause (Init) indicates that the attacker initially has all free names of P_0. Clause (Rn) means that the attacker can generate fresh names $b[x]$. Clause (Rf) mean that the attacker can apply all functions to all terms it has. In this clause, the rewrite rules $h(M_1, \dots, M_n) \rightarrow M$ and $h(M'_1, \dots, M'_n) \rightarrow M'$ may be different elements of $\mathrm{def}_{\Sigma'}(h)$; their variables are renamed so that $M_1, \dots, M_n, M$ on the one hand and $M'_1, \dots, M'_n, M'$ on the other hand do not share variables. Clause (Rl) means that the attacker can listen on all the channels it has, and (Rs) that it can send all the messages it has on all the channels it has.

Clauses for the protocol When a function ρ associates a pair of patterns with each name and variable, and f is a constructor, we extend ρ as a substitution by $\rho(f(M_1, \dots, M_n)) = (f(p_1, \dots, p_n), f(p'_1, \dots, p'_n))$ where $\rho(M_i) = (p_i, p'_i)$. We denote by $\rho(M)_1$ and $\rho(M)_2$ the components of the pair $\rho(M)$. We define $\rho(\mathsf{diff}[M, M']) = (\rho(M)_1, \rho(M')_2)$.

The translation $[\![P]\!]\rho s s' H$ 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 and s' are sequences 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 $H \wedge F$.

$$[\![0]\!]\rho s s' H = \emptyset$$

$$[\![!P]\!]\rho s s' H = [\![P]\!]\rho(s, i)(s', i)H$$

where i is a fresh variable

$$[\![P \mid Q]\!]\rho s s' H = [\![P]\!]\rho s s' H \cup [\![Q]\!]\rho s s' H$$

$$[\![(\nu a)P]\!]\rho s s' H = [\![P]\!](\rho[a \mapsto (a[s], a[s'])])s s' H$$

$$\begin{aligned}[\![M(x).P]\!]\rho s s' H = {} & [\![P]\!](\rho[x \mapsto (x', x'')])(s, x')(s', x'') \\ & (H \wedge \mathrm{msg}'(\rho(M)_1, x', \rho(M)_2, x'')) \\ & \cup \{H \rightarrow \mathrm{input}'(\rho(M)_1, \rho(M)_2)\}\end{aligned}$$

where x' and x'' are fresh variables

$$\begin{aligned}[\![\overline{M}\langle N\rangle.P]\!]\rho s s' H = {} & [\![P]\!]\rho s s' H \\ & \cup \{H \rightarrow \mathrm{msg}'(\rho(M)_1, \rho(N)_1, \rho(M)_2, \rho(N)_2)\}\end{aligned}$$

$$\begin{aligned}[\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho s s' H = {} & \\ \bigcup\{ & [\![P]\!]((\sigma\rho)[x \mapsto (p, p')])(\sigma s, p)(\sigma s', p')(\sigma H) \mid \\ & (\rho(D)_1, \rho(D)_2) \Downarrow' ((p, p'), \sigma)\} \\ \cup\ & [\![Q]\!]\rho s s' (H \wedge \mathrm{fails}(\rho(D)_1) \wedge \mathrm{fails}(\rho(D)_2) \\ \cup\ & \{\sigma H \wedge \mathrm{fails}(\sigma\rho(D)_2) \rightarrow \mathrm{bad} \mid \rho(D)_1 \Downarrow' (p, \sigma)\} \\ \cup\ & \{\sigma H \wedge \mathrm{fails}(\sigma\rho(D)_1) \rightarrow \mathrm{bad} \mid \rho(D)_2 \Downarrow' (p', \sigma)\}\end{aligned}$$

where $\mathrm{fails}(D) = \bigwedge_{\sigma \mid D \Downarrow' (p, \sigma)} \mathrm{nounif}(D, \mathit{GVar}(\sigma D))$

In the translation, the environment ρ maps names and variables to their corresponding pair of patterns—one pattern for each version of the biprocess. The sequences s and s' contain all input messages, session identifiers, and results of term evaluations in the enclosing context—one sequence for each version of the biprocess. They are used in the restriction case $(\nu a)P$, to build patterns $a[s]$ and $a[s']$ that correspond to the name a. The sequence H contains all facts that must be true to run the current process.

The clauses generated are similar to those of [12], but clauses are added to indicate which tests the adversary can perform, and predicates have twice as many arguments. Replication creates a new session identifier i, added to s and s'. Replication is otherwise ignored, since Horn clauses can be applied any number of times anyway.

In the translation of an input, the sequence H is extended with the input in question and the environment ρ with a binding of x to a new variable x' in version 1, x'' in version 2. Moreover, a new clause $H \rightarrow \mathrm{input}'(\rho(M)_1, \rho(M)_2)$ is added, indicating that when all conditions in H are true, an input on channel M may be executed. This input may enable the adversary to infer that M is equal to some channel used for output; Clauses (Rcom) or (Rcom') derive bad when this information may break equivalence. The output case adds a clause stating that message N may be sent on channel M. Finally, the clauses for a term evaluation are the union of clauses for the cases where the term evaluation succeeds on both sides (then we execute P), where the term evaluation fails on both sides (then we execute Q), and where the term evaluation fails on one side and succeeds on the other (then we derive bad). Indeed, in the last case, the adversary may get to know whether the term evaluation succeeds or fails (when the code executed in the success case is visibly different from the code executed in the failure case).

Example 6 The biprocess of Example 2 yields the clauses:

$$\mathrm{msg}'(c, pk(s), c, pk(s))$$

$$\begin{aligned}& \mathrm{msg}'(c', x, c', x') \\ & \quad \rightarrow \mathrm{msg}'(c, enc(x, pk(s), a[i, x]), c, a[i, x'])\end{aligned}$$

The first clause corresponds to the output of the public key $pk(s)$. The second clause corresponds to the other output: if a message x (resp. x') is received on channel c', then the message $enc(x, pk(s), a[i,x])$ in the first version (resp. $a[i,x']$ in the second version) is sent on channel c.

Example 7 The process $c(x).let\ y = \text{eval}\ dec(x,a)\ in\ \overline{c}\langle y\rangle$, where $dec(enc(x,y),y) \rightarrow x$, yields the clauses:

$$\text{msg}'(c, enc(y,a), c, x') \wedge \text{nounif}(x', enc(\mathsf{g}, a)) \rightarrow \text{bad}$$
$$\text{msg}'(c, x, c, enc(y',a)) \wedge \text{nounif}(x, enc(\mathsf{g}, a)) \rightarrow \text{bad}$$
$$\text{msg}'(c, enc(y,a), c, enc(y',a)) \rightarrow \text{msg}'(c, y, c, y')$$

In the first clause, a message received on c is of the form $enc(y,a)$ in the first version but not in the second version; decryption succeeds only in the first version, so the process is not uniform and we derive bad. The second clause is the symmetric case. In the third clause, decryption succeeds in both versions, and yields an output on channel c.

Proving equivalences Let $\rho = \{a \mapsto a[\,] \mid a \in fn(P_0)\}$. We define the set of clauses that corresponds to biprocess P_0 as: $\mathcal{R}_{P_0} = [\![\text{unevaluated}(P_0)]\!]\rho\emptyset\emptyset\emptyset \cup \{$(Init), (Rn), ..., (Rt), (Rt')$\}$. The following result shows the soundness of clause generation.

Theorem 2 *If* bad *is not a logical consequence of* $\mathcal{R}_{P_0}$*, then* P_0 *satisfies observational equivalence.*

To determine whether bad is a logical consequence of $\mathcal{R}_{P_0}$, we use a resolution-based solving algorithm, similar to one in previous work [13]. The main differences are as follows:

- Instead of nounif, that previous work uses a predicate testunif with a different semantics. We adapt the simplification steps of testunif accordingly. In particular, the semantics of nounif relies on unification modulo the equational theory of Σ, whereas testunif uses syntactic unification. To unify the patterns p and p' modulo the equational theory, we apply the rewrite rules of Σ' to every function symbol that appears in the patterns, then syntactically unify the results. Formally, the most general unifiers of p and p' modulo Σ are the substitutions $\sigma_u\sigma$ such that $\text{addeval}(p,p') \Downarrow' ((p_1, p'_1), \sigma)$ and σ_u is the most general unifier of p_1 and p'_1.
- Clauses that contain terms reducible by $\mathcal{S}$ can be eliminated. (Clauses in normal form relatively to $\mathcal{S}, \Sigma$ are sufficient for not missing derivations of bad.)

When bad is a logical consequence of $\mathcal{R}_{P_0}$, the derivation of bad from $\mathcal{R}_{P_0}$ can serve for reconstructing a violation of the hypothesis of Corollary 1, via an extension of recent techniques for secrecy analysis [8]. However, the translation of protocols to Horn clauses performs safe abstractions that sometimes result in false counterexamples.

7. Applications (summary)

Weak secrets Protocols that use passwords and other low-entropy secrets may be subject to off-line guessing attacks: the attacker first interacts with the protocol, then guesses a password (possibly by enumerating a dictionary), and checks its guess against the protocol messages, without further interaction with the protocol. As we show in the full paper, the absence of off-line guessing attacks can be defined as an equivalence in a simple extension of our formalism that supports several execution stages, and checked automatically with our technique. Following this approach, we have proved that four variants of EKE [9, 10] are resistant to off-line guessing attacks. The handling of equational theories is crucial for this application.

Authenticity Abadi and Gordon [6] use equivalences for characterizing authenticity properties, and treat a variant of the Wide-Mouth-Frog protocol as an example. Essentially, authenticity is defined as an equivalence between the protocol and a specification. The technique presented in this paper automatically proves authenticity for the one-session version of this protocol [6, section 3.2.2], thereby eliminating the need for a laborious manual proof. It can also be used for simplifying the proof of authenticity for the multi-session version. (Authenticity properties are sometimes formulated as correspondence assertions on behaviors, rather than as equivalences. Previous work shows how to check those assertions with ProVerif [12].)

Complete sessions in JFK Finally, we show other ways in which automated proofs of equivalences can contribute to protocol analyses, specifically studying JFK, a modern session-establishment protocol for IP security [7].

In recent work [3], we modelled JFK in the applied pi calculus. We used processes for representing the reachable states of JFK, for any number of principals and sessions, and stated security properties as equivalences. Although we relied on ProVerif for reasoning about behaviors, our main proofs of equivalences were manual. Applying the techniques of this paper, we can revise and largely automate those proofs. The resulting proofs rely on equivalences on biprocesses, verified by ProVerif, composed with standard pi calculus equivalences that do not depend on the signature for terms.

In particular, a core property of JFK is that, once a session completes, its session key is (apparently) unrelated to the cryptographic material exchanged during the session, and all those values can be replaced by distinct fresh names [3, Theorem 2]. This property can be stated and proved in terms of a biprocess S that outputs either the actual results of JFK computations (in $\mathsf{fst}(S)$) or distinct fresh names (in $\mathsf{snd}(S)$), in parallel with the rest of the JFK system to account for any other sessions. The proof of this property goes as follows. The system is split into $S \approx C[S']$, where S' is similar to S but omits unimportant parts of JFK, collected in the evaluation context $C[\_]$. The proof that $S \approx C[S']$ is straightforward; it relies on pi calculus equivalences that eliminate communications on private channels introduced in the split. ProVerif shows that S' satisfies equivalence. Using the contextual property of equivalence, $C[S']$ satisfies equivalence, hence $\mathsf{fst}(S) \approx \mathsf{snd}(S)$.

8. Conclusion

In the last decade, there has been substantial research on other proof methods for security protocols. While many of those proof methods have focused on predicates on behaviors, others have addressed equivalences between systems (e.g., [1, 4–6, 14–21, 24]). Much of this research is concerned with obtaining sound and complete proof systems, often via sophisticated bisimulations, and eventually decision algorithms for restricted cases. In our opinion, these are important goals, and the results to date are significant.

In the present paper, we aim to contribute to this body of research with a different approach. We do not emphasize the development of bisimulation techniques. Rather, we leverage behavior-oriented techniques and tools (ProVerif, in particular) for equivalence proofs. We show how to derive equivalences by reasoning about behaviors—specifically, by reasoning about behaviors of applied pi calculus biprocesses. We also show how to translate those biprocesses to Horn clauses and how to reason about their behaviors by resolution. The resulting proof method is sound, although that is not simple to establish. We demonstrate the usefulness of the method through automated analyses of interesting, infinite-state systems.

Acknowledgments Martín Abadi's work was partly done at Microsoft Research, Silicon Valley, and also partly supported by the National Science Foundation under Grants CCR-0204162 and CCR-0208800. Bruno Blanchet's work was partly done at Max-Planck-Institut für Informatik, Saarbrücken. We are grateful to Harald Ganzinger for helpful discussions on the treatment of equational theories.

References

[1] M. Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM*, 46(5):749–786, Sept. 1999.
[2] M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. *Journal of the ACM*, 52(1):102–146, Jan. 2005.
[3] M. Abadi, B. Blanchet, and C. Fournet. Just Fast Keying in the Pi Calculus. In *ESOP'04*, volume 2986 of *LNCS*, pages 340–354. Springer, Mar. 2004.
[4] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *POPL'01*, pages 104–115. ACM Press, Jan. 2001.
[5] M. Abadi and A. D. Gordon. A Bisimulation Method for Cryptographic Protocols. *Nordic Journal of Computing*, 5(4):267–303, Winter 1998.
[6] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, Jan. 1999.
[7] W. Aiello, S. M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, K. Keromytis, and O. Reingold. Just Fast Keying: Key Agreement in a Hostile Internet. *ACM TISSEC*, 7(2):242–273, May 2004.
[8] X. Allamigeon and B. Blanchet. Reconstruction of Attacks against Cryptographic Protocols. In *CSFW-18*, June 2005.
[9] S. M. Bellovin and M. Merritt. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. In *Security and Privacy*, pages 72–84, May 1992.
[10] S. M. Bellovin and M. Merritt. Augmented Encrypted Key Exchange: a Password-Based Protocol Secure Against Dictionary Attacks and Password File Compromise. In *CCS'93*, pages 244–250, Nov. 1993.
[11] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *CSFW-14*, pages 82–96, June 2001.
[12] B. Blanchet. From Secrecy to Authenticity in Security Protocols. In *SAS'02*, volume 2477 of *LNCS*, pages 342–359. Springer, Sept. 2002.
[13] B. Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *Security and Privacy*, pages 86–100, May 2004.
[14] M. Boreale, R. De Nicola, and R. Pugliese. Proof Techniques for Cryptographic Processes. *SIAM Journal on Computing*, 31(3):947–986, 2002.
[15] J. Borgström, S. Briais, and U. Nestmann. Symbolic Bisimulation in the Spi Calculus. In *CONCUR 2004*, volume 3170 of *LNCS*, pages 161–176. Springer, Aug. 2004.
[16] J. Borgström and U. Nestmann. On Bisimulations for the Spi Calculus. In *AMAST 2002*, volume 2422 of *LNCS*, pages 287–303. Springer, Sept. 2002.
[17] V. Cortier. *Vérification Automatique des Protocoles Cryptographiques*. PhD thesis, ENS de Cachan, Mar. 2003.
[18] L. Durante, R. Sisto, and A. Valenzano. Automatic Testing Equivalence Verification of Spi Calculus Specifications. *ACM TOSEM*, 12(2):222–284, Apr. 2003.
[19] R. Focardi and R. Gorrieri. The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, Sept. 1997.
[20] H. Hüttel. Deciding Framed Bisimilarity. In *INFINITY'02*, pages 1–20, Aug. 2002.
[21] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic Polynomial-time Equivalence and Security Protocols. In *FM'99*, volume 1708 of *LNCS*, pages 776–793. Springer, Sept. 1999.
[22] F. Pottier. A Simple View of Type-Secure Information Flow in the π-Calculus. In *CSFW-15*, pages 320–330, June 2002.
[23] F. Pottier and V. Simonet. Information Flow Inference for ML. In *POPL'02*, pages 319–330, Jan. 2002.
[24] A. Ramanathan, J. Mitchell, A. Scedrov, and V. Teague. Probabilistic Bisimulation and Equivalence for Security Analysis of Network Protocols. In *FOSSACS 2004*, volume 2987 of *LNCS*, pages 468–483. Springer, Mar. 2004.