Automatic Proof of Strong Secrecy for Security Protocols

Bruno Blanchet
CNRS, École Normale Supérieure, Paris
and Max-Planck-Institut für Informatik, Saarbrücken
Bruno.Blanchet@ens.fr

Abstract

We present a new automatic technique for proving strong secrecy *for security protocols. Strong secrecy means that an adversary cannot see any difference when the value of the secret changes. Our technique relies on an automatic translation of the protocol into Horn clauses, and a resolution algorithm on the clauses. It requires important extensions with respect to previous work for the proof of (standard) secrecy and authenticity. This technique can handle a wide range of cryptographic primitives, and yields proofs valid for an unbounded number of sessions and an unbounded message space; it is also flexible and efficient. We have proved its correctness, implemented it, and tested it on several examples of protocols including JFK [11] (a proposed replacement for IPsec).*

1. Introduction

Secrecy is a basic and important security notion in the analysis of cryptographic protocols. It formalizes that an attacker must not be able to find out sensitive data. Even for such a basic notion, several different definitions can be given. We focus here on definitions of secrecy in formal models of cryptography, which do not consider bitstrings, but terms built from abstract perfect cryptographic primitives. As discussed in [2], in such models, two main ways of defining secrecy can be distinguished:

- The first one, which we name *standard secrecy*, means that an attacker must not be able to obtain the value of the secret itself. This is the one most frequently checked in cryptographic protocol verification, using techniques such as model-checking [34], theorem proving [39], typing and logic programming [3] for instance.
- The second one, which we name *strong secrecy*, means that an attacker must see no difference when the value of the secret changes. This notion of secrecy is stronger than the first one. It is similar to the notion of non-interference, which has been widely used in language-based security, but in general without cryptography [43]. Exceptions include [25], which defines a similar notion of secrecy for a property instead of a value, [1], which presents a type system for verifying strong secrecy for the spi calculus, and [7], which uses manual process equivalence proofs for deriving strong secrecy. This definition of secrecy is sometimes considered too restrictive. However, we believe that, in the case of cryptographic protocols, this notion of secrecy is not overly restrictive, since encryption can be used to hide information, so that sending a secret encrypted does not reveal it.

Strong secrecy is interesting for several reasons:

- It takes into account implicit flows. For instance, an implicit flow occurs when a test yields different results depending on the value of a secret and the outcome of the test is subsequently known to the adversary. This is particularly important when the secrets can take only a few values (see Section 6 for more details on this point).
- Strong secrecy is a particular case of process equivalence. Process equivalences provide a simple and elegant way of formalizing security properties. They have nice compositionality properties. This makes it easy to combine manual proofs of some lemmas with automatic proofs of others.
- Even though it is based on a formal model, strong secrecy is closer to the computational definition of secrecy, for example semantic security of a key. Indeed, intuitively, semantic security means that a polynomial-time adversary has negligible probability to distinguish between a process that gives it the correct key and a process that gives it a random key. Our notion of secrecy also means that the adversary cannot distinguish between different values of the secret. This link

is formally illustrated by [8, 10], which show that, for passive attacks and symmetric encryption, plus additional technical restrictions, process equivalences imply computational equivalences.

In this paper, we present an automatic technique for verifying strong secrecy. In our technique, the protocol is given as input under the form of a process in an extension of the pi calculus with a generic treatment of cryptographic primitives. This process is then automatically translated into a set of Horn clauses, which yields an abstract representation of the protocol. This translation extends the translation given in [3] for standard secrecy and in [16] for authenticity. The main new idea of this translation is to collect all tests that the adversary can do: in particular, a test can be the application of a primitive that can succeed or fail (such as a decryption or an equality test). The set of all possible tests is infinite, but can be finitely described by Horn clauses. The main challenge is then to make sure, without enumerating this infinite set of tests, that none of them reveals information about the secrets. We achieve this goal by adding a new predicate "testunif" in the clauses, which is not defined by Horn clauses, but by specific simplification steps in the solving algorithm. Taking into account the meaning of testunif, we can show that, if a certain fact bad is not derivable from the clauses, then the protocol satisfies the desired strong secrecy property. We use a resolution-based algorithm to determine derivability of facts from the clauses. This algorithm is also an extension of the algorithm previously used for standard secrecy. The major novelty of this algorithm is the introduction of the simplification steps for testunif. We have proved our technique correct (including the translation and the solving algorithm), implemented it, and tested it on several examples of protocols, including JFK [11] (a proposed replacement for IPsec).

We have also proved the termination of the algorithm for the large class of *tagged protocols*, thus extending [17] to strong secrecy. (Experimentally, the algorithm also terminates on many non-tagged protocols.) Intuitively, a tagged protocol is a protocol in which each encryption, signature, ... is distinguished from others by a constant tag. For instance, to encrypt m under k, we write $sencrypt((t, m), k)$ instead of $sencrypt(m, k)$, where t is a constant tag. Different encryptions in the protocol use different tags, and the receiver of a message always checks the tags.

Related work There is a long line of research on information flow for programs (without cryptography), using finite-state exploration tools such as CoSeC [28] or type systems [43]. These type systems are often very restrictive. We refer to [40] for a detailed survey. Adding cryptography considerably complicates the proof of non-interference, since one should be allowed to publish the encryption of a secret, although it depends on the secret, and therefore would be reported as a violation by these type systems. One might use explicit downgrading [37] to solve this problem, allowing the secret to be downgraded into a public value at specific places in the program. Obviously, downgrading is not safe in general, so one must control that it does not leak more information than intended, for example as in [44]. The current techniques for controlling downgrading do not take into account the algebraic properties of the cryptographic primitives, which are key to determine which messages one should be allowed to publish. So our work relies on a different approach: we use exclusively the algebraic properties of the cryptographic primitives to prove strong secrecy, without any downgrading indications.

In the same spirit as explicit downgrading, Dam and Giambiagi [24, 30] use the technique of admissible flows to check that flows in the implementation of a cryptographic protocol satisfy a dependency specification, which gives tight control on which arguments are allowed for encryption, and which messages may be sent. However, the technique does not answer the question of whether a dependency specification guarantees that some piece of data is kept secret. Our work essentially aims at answering this question.

The concept of process equivalence was the basis for the first manual proofs in the spi calculus [6, 7]. Other works have continued in this line of research, for example [18–20, 29, 36], with different calculi and notions of process equivalences. Manual proof techniques for equivalences have also been designed for a lambda calculus extended with cryptographic primitives [41, 42]. Our main contribution with respect to these works is to offer a fully automatic proof technique for a particular case of process equivalence.

Durante, Cisto, and Valenzano [27] introduce a model-checking technique for proving testing equivalence of spi calculus processes. The main limitation of this technique is that, to have finite processes, they interpret replication as a finite composition of n parallel processes, where n is chosen by the user, instead of the unbounded composition of parallel processes which is normally the semantics of replication. We do not have this limitation: we consider replication as an unbounded parallel composition of processes.

The work closest to ours is [1]. It introduces a type system for proving strong secrecy for spi calculus processes. Simon Gay has implemented a type-checker for this system. Type inference does not seem particularly difficult, but has not been implemented yet as far as we know. This system is not very flexible: in particular, it is limited to probabilistic shared-key encryption as the only cryptographic primitive; it requires a specific ordering of components of messages and of ciphertexts according to their secrecy status. Our work removes these restrictions: it allows generic definitions of cryptographic primitives by reduction rules; it gives full freedom for building messages by applying the primitives.

Following Lowe [35], several works [22, 23] aim to automatically verify protocols subject to guessing attacks. These works use a two-phase scenario to model a guessing attack: first, the attacker interacts actively with the protocol, to obtain a set of messages; second, it guesses the secret (presumably by exhaustive enumeration) and checks its guess off-line using the messages gathered in the first, on-line phase. Our notion of secrecy is stronger: it also fails when the attacker can check its guess only by on-line interaction with the protocol. Which notion is more appropriate depends on the practical situation. When the secret can take a very small number of values, an attack with on-line checking may already be harmful. When the set of possible values is larger, on-line checking might be too difficult or too long for the adversary, so the off-line guessing attacks of [22, 23, 35] might be more appropriate.

Although most works on automatic verification of protocols are based on formal models, Laud [33] presents a tool for verifying protocols in a computational model. This tool is limited to passive attacks and symmetric encryption, much like the results of [8, 10]. Backes and Pfitzmann [12, 13] have developed a semantics for information flow in the presence of cryptography, under arbitrary active attacks, in a computational model. Backes, Pfitzmann, and Waidner [14] have shown the soundness of an abstract cryptographic library including public-key encryption, signatures, and nonces (but not symmetric encryption) with respect to computational primitives, under arbitrary active attacks. We could adapt our primitives to provide exactly the same basic operations as their ideal library; however, relating the definitions of secrecy would still need important work and the technical differences in the presentation of the models would complicate this task. More generally, in the line of [8, 10], these results might be used as a basis for future results establishing, with suitable assumptions and restrictions, the soundness of formal definitions such as ours with respect to the computational view of cryptography.

Outline Section 2 presents an overview of the process calculus. Section 3 formally defines strong secrecy and gives a first criterion to prove it. Section 4 presents the translation into Horn clauses and Section 5 the solving algorithm. Section 6 sketches extensions. Section 7 presents some of our experimental results and Section 8 concludes. By lack of space, the termination result is omitted and proofs are sketched.

2. The Process Calculus

This section presents the process calculus that we use to represent protocols. It is essentially the calculus of [16], with the addition of probabilistic cryptographic primitives. Figure 1 gives the syntax of terms (messages) and processes (programs) of our calculus. It assumes infinite sets of names

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
0	nil
$P \mid Q$	parallel composition
$!P$	replication
$(\nu a)P$	restriction
$\overline{M}\langle N\rangle.P$	output
$M(x).P$	input
$let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q$	destructor application
$let\ x = M\ in\ P$	local binding
$if\ M = N\ then\ P\ else\ Q$	conditional

Figure 1. Syntax of the process calculus

and variables, and sets of symbols for constructors and destructors; we use a, b, c, k for names, x, y, z for variables, f for a constructor, and g for a destructor.

Constructors are used to build terms. So the terms are variables, names, and constructor applications $f(M_1, \ldots, M_n)$. Destructors do not appear in terms, but manipulate terms in processes. They are partial functions on terms that processes can apply. The process $let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q$ tries to evaluate $g(M_1, \ldots, M_n)$; if this succeeds, x is bound to the result and P is executed; otherwise, Q is executed. A destructor g is defined by a finite set $\mathrm{def}(g)$ of reductions $g(N_1, \ldots, N_n) \to N$, where $N_1, \ldots, N_n, N$ are terms without free names, such that the variables of N also occur in $N_1, \ldots, N_n$. Then $g(M_1, \ldots, M_n)$ is defined if and only if there exists a substitution σ and a reduction $g(N_1, \ldots, N_n) \to N$ in $\mathrm{def}(g)$ such that $M_i = \sigma N_i$ for all $i \in \{1, \ldots, n\}$, and in this case $g(M_1, \ldots, M_n) \to \sigma N$.

The calculus includes standard pi calculus constructs: 0 does nothing; $P \mid Q$ 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 runs P (a new name is a secret that the adversary cannot guess; it may obtain it by computation from public messages, however); the process $M(x).P$ inputs a message on channel M, and runs P with x bound to the input message; the process $\overline{M}\langle N\rangle.P$ outputs the message N on the channel M and then runs P. A channel can be any term M, but the process blocks if M does not reduce to a name at runtime. The local binding $let\ x = M\ in\ P$ is syntactic sugar for $P\{M/x\}$, where $\{M/x\}$ is the substitution that replaces x with M. The conditional $if\ M = N\ then\ P\ else\ Q$ is in fact a particular

Tuples:
C: tuple $(x_1, \ldots, x_n)$
D: projections $ith_n((x_1, \ldots, x_n)) \rightarrow x_i$
Shared-key encryption:
C: encryption of x under the key y, $sencrypt(x, y)$
D: decryption $sdecrypt(sencrypt(x, y), y) \rightarrow y$
Probabilistic shared-key encryption:
C: encryption of x under the key y with random coins r, $sencrypt_{prob}(x, y, r)$
D: decryption $sdecrypt_{prob}(sencrypt_{prob}(x, y, r), y) \rightarrow x$
Probabilistic public-key encryption:
C: encryption of x under the public key y with random coins r, $pencrypt_{prob}(x, y, r)$
C: public key generation from a secret key y, $pk(y)$
D: decryption $pdecrypt_{prob}(pencrypt_{prob}(x, pk(y), r), y) \rightarrow x$
Signatures:
C: signature of x with the secret key y, $sign(x, y)$
D: signature verification $checksign(sign(x, y), pk(y)) \rightarrow y$
D: message without signature $getmess(sign(x, y)) \rightarrow x$
One-way hash functions:
C: hash function $H(x)$

Figure 2. Constructors and destructors (C means Constructor, D means destructor)

case of destructor application. Let eq be the destructor defined by $eq(x, x) \rightarrow x$. Then $if\ M = N\ then\ P\ else\ Q$ is equivalent to $let\ x = eq(M, N)\ in\ P\ else\ Q$ where x is not free in P. As usual, we omit an $else$ clause when it consists of 0.

Using constructors and destructors, we can represent data structures and cryptographic operations. Some examples are given in Figure 2. For instance, $sencrypt_{prob}(M, N, R)$ is the symmetric (shared-key) probabilistic encryption of the message M under the key N using the random coins (i.e. confounder) R, where $sencrypt_{prob}$ is a constructor. The corresponding destructor $sdecrypt_{prob}(M', N)$ returns the decryption of M' if M' is a message encrypted under N. (We assume perfect encryption, so one can decrypt only when one has the key.) Probabilistic encryption should be used as follows: $(\nu r) let\ x = sencrypt_{prob}(m, k, r)\ in \ldots$ where the coins r are fresh and used in a single encryption. We could model both deterministic and probabilistic public-key encryption, as we do for shared-key encryption. However, a deterministic function cannot satisfy the definition of semantic security for public-key encryption [31] (in the computational model of cryptography), because if encryption is deterministic, giving $\{m\}_{pk}$ to the adversary allows it to compare m with any message m' it has, by testing the equality $\{m\}_{pk} = \{m'\}_{pk}$ since the adversary has the public keys. This attack also appears in our model, and is detected by our verifier: giving $\{m\}_{pk}$ to the adversary does not preserve the strong secrecy of m. So we focus on probabilistic public-key encryption. Signatures can be both deterministic and probabilistic. The figure gives the deterministic version.

We write $fn(P)$ and $fv(P)$ for the sets of names and variables free in P, respectively, defined as usual. A process is closed if it has no free variables. We identify processes up to renaming of bound names and variables.

As usual, the semantics for this calculus is defined by a reduction relation $\rightarrow$ and a structural equivalence relation $\equiv$, which is used to transform processes so that the reduction rules can be applied. These relations are defined in [16], for example.

As an example, we consider the following protocol inspired by the corrected version of the Denning-Sacco key distribution protocol [9]:

Message 1. $A \rightarrow B : \quad \{\{pk_A, pk_B, k\}_{sk_A}\}_{pk_B}$ k fresh
Message 2. $B \rightarrow A : \quad \{x\}_k$

In this protocol, two participants A and B wish to establish a shared key k, and B sends to A a secret x under this key. To establish the key, A creates a fresh key k, groups it with the public keys of A and B, pk_A and pk_B (to express that the key is intended for sharing between A and B), signs the message with its secret key sk_A, and encrypts it under the public key of B. On receipt of this message, B can decrypt, check the signature and the identities. It is then convinced that the key was generated by A (because of the signature) and intended to talk to B. This protocol can be represented by the following processes:

$$P_A(pk_A, sk_A) = c(x_{pk_B}).(\nu k)(\nu r)$$
$$\overline{c}\langle pencrypt_{prob}(sign((pk_A, x_{pk_B}, k), sk_A), x_{pk_B}, r)\rangle.$$
$$c(x').let\ s = sdecrypt(x', k)\ in\ 0$$
$$P_B(pk_B, sk_B, pk_A) = c(y).$$
$$let\ y' = pdecrypt_{prob}(y, sk_B)\ in$$
$$let\ (= pk_A, = pk_B, k) = checksign(y', pk_A)\ in$$
$$\overline{c}\langle sencrypt(x, k)\rangle$$
$$P_0 = (\nu sk_A)(\nu sk_B) let\ pk_A = pk(sk_A)\ in$$
$$let\ pk_B = pk(sk_B)\ in\ \overline{c}\langle pk_A\rangle\overline{c}\langle pk_B\rangle.$$
$$(!P_A(pk_A, sk_A)\ |!P_B(pk_B, sk_B, pk_A))$$

The process P_0 first creates the secret keys of A and B, computes the corresponding public keys, and publishes them by sending them on the public channel c. Then, it launches an unbounded number of copies of P_A and P_B, representing the roles of A and B in the protocol. We assume that the adversary chooses to whom each participant talks. So the process P_A first inputs on channel c a key x_{pk_B}, and then starts a session with the participant of pub-

lic key x_{pk_B}. Then it creates the key k, sends the first message, inputs the second one to obtain the value of the secret s. The process P_B inputs message 1, tries to decrypt it and check the signature. We use a pattern-matching notation $let\ (= pk_A, = pk_B, k) = \ldots$ to abbreviate a series of destructor applications and equality tests. (The pattern $= M$ matches only a term equal to M.) If the checks succeed, it outputs the secret x encrypted under k.

Our first goal is to show that an adversary cannot tell the difference between different values of the secret x. (In this process, x is a free variable, which can be substituted by any term.) We present more subtle results in Section 7.

3. Strong Secrecy

Intuitively, two processes are observationally equivalent when an adversary cannot distinguish them. This can be formalized by the following standard definition, already used in [5].

Definition 1 (Observational equivalence) *An* evaluation context C *is a context built from* $[\,]$, $C \mid P$, $P \mid C$, $(\nu a)C$.

P emits on m, $P \Downarrow m$, *if and only if* $P \ (\rightarrow \cup \equiv)^* \ C[\overline{m}\langle N\rangle.R]$ *where* C *is an evaluation context that does not bind* m.

Observational equivalence $\approx$ *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* $P \rightarrow P'$ *then there exists* Q' *such that* $Q \rightarrow^* Q'$ *and* $P'\,\mathcal{R}\,Q'$;
3. $C[P]\,\mathcal{R}\,C[Q]$ *for all closing evaluation contexts* C.

A context is a process with a "hole" $[\,]$. An evaluation context has only restrictions and parallel compositions above the hole. A process P emits on m when, after any number of reduction steps, it sends a message on channel m, and the adversary has m (i.e. m is not bound in P). So the output on channel m is observable by the adversary. This fact is exploited in point 1 of the definition of observational equivalence: for P and Q to be observationally equivalent, if P emits on m, then Q must also emit on m, otherwise the adversary would distinguish them.

Point 2 means that observational equivalence is preserved by reduction. It makes the branching structure of the process reductions observable to the adversary: when P and Q are observationally equivalent, if P can reduce nondeterministically into two non-equivalent processes P' and P'', then Q must also reduce into two processes Q' and Q'', observationally equivalent to P' and P'' respectively.

Point 3 takes into account the adversary, represented by any closing evaluation context, that is, an evaluation context such that $C[P]$ and $C[Q]$ are closed. The main point is that we allow parallel composition with an arbitrary process representing a Dolev-Yao adversary [26] which can eavesdrop, compute, and send messages. Allowing restrictions in the context is not essential here; we allow any evaluation context as usual in the theory of process calculi.

Although the observation allowed in point 1 is very weak, the universal quantification over all contexts in point 3 leads to a strong definition of observational equivalence. For instance, if $P = (\nu a)(\nu b)(\overline{c}\langle a\rangle \mid \overline{d}\langle b\rangle)$ and $Q = (\nu a)(\nu b)(\overline{c}\langle a\rangle \mid \overline{d}\langle a\rangle)$, P and Q both emit on the same channels c and d, but they are not observationally equivalent, because the following context can distinguish them: $C = [\,] \mid c(x).d(y).if\ x = y\ then\ \overline{e}\langle x\rangle$. The process $C[Q]$ emits on e, while $C[P]$ does not.

As usual in formal models of cryptographic protocols, many details are abstracted away. For instance, this definition does not consider timing attacks and it abstracts the length of messages. In the real world, an adversary can distinguish the encryption of messages of different length, by comparing the length of the ciphertexts. Here, we consider that a ciphertext that the adversary cannot decrypt gives no information on the cleartext. This problem could be solved by defining a fixed-length format for the messages in the protocol, so that the length of a message is public information, and does not reveal any secret.

We can now define strong secrecy by requiring that the adversary cannot distinguish processes that differ by the values of the secrets.

Definition 2 (Strong Secrecy) *The process* P_0 *preserves the strong secrecy of its free variables if and only if for all closed substitutions* σ *and* σ' *of domain* $fv(P_0)$, $\sigma P_0 \approx \sigma' P_0$. *(The substitution avoids name captures by first alpha renaming* P_0 *if necessary.)*

Intuitively, if a secret leaks, then there exists a context C (i.e. an adversary) that can exploit the leak to execute different outputs depending on the value of the secret, so that $C[\sigma P_0]$ and $C[\sigma' P_0]$ do not satisfy the point 1 of the definition of observational equivalence.

Next, we give a stronger criterion that we are going to use in order to prove strong secrecy. Intuitively, we require that each reduction step proceeds uniformly for all values of the secrets. For a communication step, this means that the channel must not be a secret. For a destructor application, this means that the destructor must succeed or fail independently of the values of the secrets.

Proposition 1 *Let* $Secr$ *be a set of names containing as many elements as* $fv(P_0)$ *and such that* $fn(P_0) \cap Secr = \emptyset$. *Let* σ_0 *be a substitution mapping all free variables of* P_0 *to*

distinct elements of $Secr$. Assume that, for all Q such that $fn(Q) \cap Secr = \emptyset$,

1. *if $\sigma_0 P_0 \mid Q \ (\rightarrow \cup \equiv)^* \ C[\overline{M}\langle N\rangle.R]$ or $\sigma_0 P_0 \mid Q \ (\rightarrow \cup \equiv)^* \ C[M(x).R]$ and C is an evaluation context that does not bind names in $Secr$ then $M \notin Secr$.*
2. *if $\sigma_0 P_0 \mid Q \ (\rightarrow \cup \equiv)^* \ C[let \ y = g(M_1, \ldots, M_n) \ in \ Q' \ else \ R']$, θ is a closed substitution with $dom(\theta) = Secr$, C is an evaluation context that does not bind names in $Secr$ or in the image of θ, $g(N_1, \ldots, N_n) \rightarrow N$ is in $\mathrm{def}(g)$, and $\theta(M_1, \ldots, M_n)$ is an instance of $(N_1, \ldots, N_n)$, then $(M_1, \ldots, M_n)$ is an instance of $(N_1, \ldots, N_n)$.*

Then P_0 preserves the strong secrecy of its free variables.

The process Q represents any adversary. We use a process rather than a context here to be closer to our previous results [3, 16], which we use in the following of the paper. The second hypothesis means that, if the destructor application $g(M_1, \ldots, M_n)$ succeeds for the values of the secrets given by θ (that is, $\theta(M_1, \ldots, M_n)$ is an instance of $(N_1, \ldots, N_n)$), then it must succeed for any value of the secrets (which is true when $(M_1, \ldots, M_n)$ is an instance of $(N_1, \ldots, N_n)$).

Proof sketch We first reformulate the hypotheses 1 and 2 using contexts to represent the adversary. Let C be an evaluation context. Let $Secr'$ be a set of names not free or bound in C and not free in P_0, containing as many elements as $fv(P_0)$. Let σ_0' be a substitution mapping the free variables of P_0 to distinct elements of $Secr'$.

1. If $C[\sigma_0' P_0] \ (\rightarrow \cup \equiv)^* \ C'[\overline{M}\langle N\rangle.R]$ or $C[\sigma_0' P_0] \ (\rightarrow \cup \equiv)^* \ C'[M(x).R]$ and C' is an evaluation context that does not bind names in $Secr'$ then $M \notin Secr'$.
2. If $C[\sigma_0' P_0] \ (\rightarrow \cup \equiv)^* \ C'[let \ y = g(M_1, \ldots, M_n) \ in \ Q' \ else \ R']$, θ is a closed substitution with $dom(\theta) = Secr'$, C' is an evaluation context that does not bind names in $Secr'$ or in the image of θ, $g(N_1, \ldots, N_n) \rightarrow N$ is in $\mathrm{def}(g)$, and $\theta(M_1, \ldots, M_n)$ is an instance of $(N_1, \ldots, N_n)$, then $(M_1, \ldots, M_n)$ is an instance of $(N_1, \ldots, N_n)$.

This part of the proof is delicate, because we have to avoid name captures of elements of $Secr$ by the context C, that is why we rename them into elements of $Secr'$.

Let us define a relation $\mathcal{R}$ by $P \mathcal{R} P'$ if and only if there exist 1) an evaluation context C, 2) a set $Secr'$ of names not free or bound in C and not free in P_0, containing as many elements as $fv(P_0)$, 3) a substitution σ_0' mapping the free variables of P_0 to distinct elements of $Secr'$, 4) substitutions θ and θ' of domain $Secr'$, and 5) a closed process P'' such that $P = \theta P''$, $P' = \theta' P''$, and $C[\sigma_0' P_0] \ (\rightarrow \cup \equiv)^* \ P''$. The relation $\mathcal{R}$ is obviously symmetric. Next, we prove that $\mathcal{R}$ satisfies the three points of the definition of observational equivalence.

- The proof of point 2 is by cases on the reduction. The case of a communication uses hypothesis 1 to show that the channel is unchanged by θ and θ', so that $P = \theta P''$, $P' = \theta' P''$, and P'' can all perform the communication. The case of a destructor application uses hypothesis 2: if a destructor application succeeds in $P = \theta P''$, then by hypothesis 2, it also succeeds in P'', and so in $P' = \theta' P''$; if a destructor application fails in P, then it also fails in P'' and P', since, if it succeeded in $P' = \theta' P''$, hypothesis 2 would imply its success in P'', which implies its success in $P = \theta P''$.
- The proof of point 1 also uses hypothesis 1 to show that the channel is unchanged by θ and θ'.
- The proof of point 3 relies on the definition of $\mathcal{R}$. The delicate point is that we have to rename names in $Secr'$ to avoid their capture by the added context.

Then $\mathcal{R} \subseteq \approx$. Moreover, for all σ and σ' substitutions of domain $fv(P_0)$, $\sigma P_0 \, \mathcal{R} \, \sigma' P_0$, by taking $C = [\,]$, $Secr' = Secr$, $\sigma_0' = \sigma_0$, θ and θ' such that $\theta\sigma_0 = \sigma$ and $\theta'\sigma_0 = \sigma'$, $P'' = \sigma_0 P_0$. Therefore, P_0 preserves the strong secrecy of its free variables. □

As a sanity check, we can notice that, if P_0 in the presence of an adversary outputs a secret, then Proposition 1 does not apply. Indeed, the adversary obtaining the secret can execute a communication with the secret as channel, thus violating the first hypothesis, or execute a destructor application whose success depends on the value of the secret, thus violating the second hypothesis. More formally, if $\sigma_0 P_0 \mid Q \ (\rightarrow \cup \equiv)^* \ C[\overline{c}\langle \mathsf{x}\rangle.R]$ for some $\mathsf{x} \in Secr$ and x and c are not bound in C, then $\sigma_0 P_0 \mid (Q \mid c(y).\overline{y}\langle d\rangle)$ violates hypothesis 1 of Proposition 1, since it can execute an output on $y = \mathsf{x}$.

The converse of Proposition 1 is obviously wrong: for instance, if after a destructor application, the processes executed in the success and failure cases are not distinguishable by the adversary, then strong secrecy can be true even if the success or failure of the destructor application depends on the value of the secret. In other words, we have performed a sound approximation by using the above criterion to prove strong secrecy. This approximation is useful to have a simple enough and automatable technique. Except works that rely on manual proofs, previous works that handle cryptography, such as [1], were at least as approximate.

Because of the way non-determinism is handled in point 2 of the definition of observational equivalence, a non-deterministic process may satisfy a strong secrecy property, while a process that has less possible behaviors does not. In other words, a refinement of a process may be found less secure. However, this problem is reduced by the approximations done in Proposition 1: if

we prove strong secrecy of a process P_0 using Proposition 1, and the notion of refinement guarantees that the refined process P_0' does not execute more destructor applications than P_0, then P_0' also satisfies strong secrecy.

4. Protocol Verifier

In this section, we extend the protocol verifier of [3, 16] to verify strong secrecy. The verifier builds a set of Horn clauses (rules) from a closed process P_0. We assume that each restriction $(\nu a)P$ in P_0 has a different name a, and that this name is different from any free name of P_0.

We first associate to each replication in P_0 a *session identifier*. Session identifiers are variables taking values in a countable set of terms, and different values for each copy of the replicated process. They make it possible to distinguish different copies of the process; in particular, we use them below to distinguish names created by the same restriction in different copies of the process.

The terms of the rules are called *patterns* p and are generated by the following grammar:

$p ::=$	patterns
x, y, z	variable
e	element of $EVar$
$a[p_1, \ldots, p_n]$	name
$f(p_1, \ldots, p_n)$	constructor application

The set $EVar$ contains special constants e, used below in arguments of the $\mathsf{testunif}$ predicate. (Intuitively, these constants stand for existentially quantified variables. They are instantiated by substitutions in Definition 3 below. We write constants e in $EVar$ and names x in $Secr$ in sanserif font, to distinguish them from variables.) A restriction in a process creates a new name each time it is executed. When the restriction is under a replication, this leads to the generation of an unbounded number of names, which would immediately lead to non-termination if we represented it directly. Instead, we encode names as functions. For example, in the process P_A of Section 2, the names created by the restriction (νk) are represented in the verifier as $k[i_A, x_{pk_B}]$, where k is a function symbol and i_A is the session identifier of the replication above P_A. Indeed, the names created by (νk) after receiving different messages x_{pk_B} or in different copies of the process P_A numbered by i_A are certainly different, so these names depend on i_A and x_{pk_B}; thus, we represent them by a function of i_A and x_{pk_B}. More generally, for each name a in P_0, we have a corresponding pattern construct $a[p_1, \ldots, p_n]$, where a is called a *name function symbol*. If a is a free name, then the arity of this function is 0. If a is bound by a restriction $(\nu a)P$ in P_0, then this arity is the total number of input statements, destructor applications, and replications above the restriction $(\nu a)P$ in the abstract syntax tree of P_0. Thus, in the verifier, a new name is represented by a function of the inputs and destructor applications that take place before its creation, and of the session identifiers of replications. Thanks to the sessions identifiers, we distinguish names created in different copies of processes, so different names in the process calculus are represented by different patterns in the verifier.

The rules use the following facts:

$F ::=$	facts
$\mathsf{att}(p)$	attacker knowledge
$\mathsf{mess}(p, p')$	channel messages
$\mathsf{com}(p)$	communication on p
$\mathsf{testunif}(p, p')$	unification test
bad	bad

The fact $\mathsf{att}(p)$ means that the attacker may have p. The fact $\mathsf{mess}(p, p')$ means that the message p' may appear on channel p. The fact $\mathsf{com}(p)$ means that the attacker may be able to perform a communication on channel p, which makes it possible for it to test whether p is a name or not. The fact $\mathsf{testunif}(p, p')$ is defined as follows. We say that a pattern or a set of patterns *contains bound names* when it contains subterms of the form $a[\ldots]$ where a corresponds to a restriction in the process. In the following, we set $Secr$ and σ_0 as in Proposition 1.

Definition 3 *Let p, p' be closed patterns. We say that* $\mathsf{testunif}(p, p')$ is true *if and only if there exists a σ closed substitution of domain $Secr \cup EVar$ such that $\sigma Secr$ does not contain bound names and $\sigma p = \sigma p'$, and there exists no σ' closed substitution of domain $EVar$ such that $\sigma' p = \sigma' p'$.*

In the particular case when p and p' are closed patterns containing no element of $EVar$, $\mathsf{testunif}(p, p')$ is true when p and p' unify for some values of the secrets, but $p \neq p'$, so there exist other values of the secrets that make p and p' different. In other words, testing equality of p and p' gives some information on the values of the secrets. For instance, $\mathsf{testunif}(sencrypt(c_0, k), sencrypt(\mathsf{x}, k))$ is true, because equality of $sencrypt(c_0, k)$ and $sencrypt(\mathsf{x}, k)$ tells whether the secret x is c_0.

In the general case, $\mathsf{testunif}(p, p')$ is true when the answer to the question "do there exist values of elements of $EVar$ that make p and p' unify?" depends on the values of the secrets. As we explain next, the predicate $\mathsf{testunif}$ can be used to determine when the success or failure of a destructor application may give some information on the values of the secrets. We define $EVar(p)$ as the pattern p after substituting distinct new elements of $EVar$ for variables. Consider a destructor g defined by $g(N_1, \ldots, N_n) \rightarrow N$. The destructor application $g(M_1, \ldots, M_n)$ succeeds for some, but not all, values of the secrets, if and only if $\mathsf{testunif}((M_1, \ldots, M_n), EVar((N_1, \ldots, N_n)))$ is true.

For example, consider the destructor $pdecrypt_{prob}$ defined by $pdecrypt_{prob}(pencrypt_{prob}(x, pk(y), z), y) \rightarrow x$. The fact $\mathsf{testunif}((\mathsf{x}, sk_B[]), (pencrypt_{prob}(\mathsf{e}, pk(\mathsf{e}'), \mathsf{e}''), \mathsf{e}'))$ is true, because the unification succeeds for some values of e, e', and e'', only when x is of the form $pencrypt_{prob}(p_1, pk(sk_B[]), p_2)$. That is, the destructor application $pdecrypt_{prob}(\mathsf{x}, sk_B[])$ succeeds when x is of the form $pencrypt_{prob}(p_1, pk(sk_B[]), p_2)$, which gives some information on the value of x. The predicate $\mathsf{testunif}$ is not defined by clauses, but by special simplification steps in the solver, defined in Section 5.

The predicate bad serves in detecting violations of strong secrecy: When bad is not derivable, strong secrecy is true.

Next we define the rules representing the abilities of the attacker and the actions of the honest participants.

Rules for the attacker The abilities of the attacker are represented by the following rules. These rules encode the worst actions that can be performed by any process Q representing the attacker.

$$\text{For each } a \in fn(P_0), \mathsf{att}(a[]) \quad \text{(Init)}$$

$$\mathsf{att}(i) \Rightarrow \mathsf{att}(b[i]) \text{ where } b \notin fn(P_0) \text{ and } (\nu b) \text{ does not occur in } P_0 \quad \text{(Rn)}$$

$$\text{For each constructor } f \text{ of arity } n, \quad \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \Rightarrow \mathsf{att}(f(x_1, \ldots, x_n)) \quad \text{(Rf)}$$

$$\text{For each destructor } g, \text{ for each reduction } g(N_1, \ldots, N_n) \rightarrow N \text{ in } \mathrm{def}(g),$$

$$\mathsf{att}(N_1) \wedge \ldots \wedge \mathsf{att}(N_n) \Rightarrow \mathsf{att}(N) \quad \text{(Rg)}$$

$$\mathsf{testunif}((x_1, \ldots, x_n), EVar((N_1, \ldots, N_n))) \wedge \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \Rightarrow \mathsf{bad} \quad \text{(Rt)}$$

$$\mathsf{mess}(x, y) \wedge \mathsf{att}(x) \Rightarrow \mathsf{att}(y) \quad \text{(Rl)}$$

$$\mathsf{att}(x) \wedge \mathsf{att}(y) \Rightarrow \mathsf{mess}(x, y) \quad \text{(Rs)}$$

$$\mathsf{att}(x') \Rightarrow \mathsf{com}(x') \quad \text{(Rc)}$$

$$\text{For each } \mathsf{x} \in Secr, \mathsf{com}(\mathsf{x}) \Rightarrow \mathsf{bad} \quad \text{(Rcom)}$$

Rule (Rc) means that, if the attacker has x', then it can initiate a communication on x' (thereby testing if x' is a name). Rule (Rcom) checks that $\mathsf{com}(\mathsf{x})$ is not derivable for $\mathsf{x} \in Secr$, that is, no communication is done on channels in $Secr$, i.e. hypothesis 1 of Proposition 1. When all communications of the protocol are done on a constant public channel, this is equivalent to $\mathsf{att}(\mathsf{x})$ not derivable. This is the criterion used by the verifier to prove standard secrecy of x.

Rule (Rt) means that, if the attacker has $x_1, \ldots, x_n$, then it can test whether the destructor application $g(x_1, \ldots, x_n)$ succeeds. Hence, if $\mathsf{testunif}((x_1, \ldots, x_n), EVar((M_1, \ldots, M_n)))$ is true, that is, if for some values of the secrets $(x_1, \ldots, x_n)$ is an instance of $(M_1, \ldots, M_n)$ but not for others, then we may have an attack against strong secrecy. This rule serves in proving hypothesis 2 of Proposition 1.

Rule (Rn) means that the attacker can generate an unbounded number of new names $b[i]$. It differs from the similar rule of [16] by the presence of the hypothesis $\mathsf{att}(i)$. This hypothesis is useful to guarantee that all variables present in the conclusion of a clause are also present in its hypothesis. (This is useful, for example, in Lemma 3 below.) It implies that the values of session identifiers i are taken among the infinite set of terms that the adversary has, for instance $H(\ldots H(a[])\ldots)$ for some $a \in fn(P_0)$.

The other rules come from [3, 16]. The rule (Init) indicates that the attacker initially has all free names of P_0. The rules (Rf) and (Rg) mean that the attacker can apply all operations to all terms it has, (Rf) for constructors, (Rg) for destructors. Rule (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.

For example, the clauses (Rg) and (Rt) for public-key decryption are

$$\mathsf{att}(pencrypt_{prob}(x, pk(y), z)) \wedge \mathsf{att}(y) \Rightarrow \mathsf{att}(x)$$
$$\mathsf{testunif}((x, y), (pencrypt_{prob}(\mathsf{e}, pk(\mathsf{e}'), \mathsf{e}''), \mathsf{e}')) \wedge \mathsf{att}(x) \wedge \mathsf{att}(y) \Rightarrow \mathsf{bad}$$

The first clause expresses that the adversary can decrypt when it has the secret key y corresponding to the public key $pk(y)$. The second one that, when it has x and y, it can test whether the decryption of x by y succeeds, that is, whether (x, y) is of the form $(pencrypt_{prob}(M, pk(M'), M''), M')$ for some terms M, M', M''.

Rules for the protocol The translation $[\![P]\!]\rho sh$ of a process P is a set of rules, where ρ is a function which associates a pattern with each name and variable, s is a sequence of patterns, and h is a sequence of facts. The empty sequence is denoted by $\emptyset$; the concatenation of a pattern p to the sequence s is denoted by s, p; the concatenation of a fact F to the sequence h is denoted by $h \wedge F$. When a function ρ associates a pattern with each name and variable, and f is a constructor, we extend ρ as a substitution by $\rho(f(M_1, \ldots, M_n)) = f(\rho(M_1), \ldots, \rho(M_n))$.

$$[\![0]\!]\rho sh = \emptyset$$
$$[\![!P]\!]\rho sh = [\![P]\!](\rho[i \mapsto i'])(s, i')(h \wedge \mathsf{att}(i'))$$
$$[\![P \mid Q]\!]\rho sh = [\![P]\!]\rho sh \cup [\![Q]\!]\rho sh$$
$$[\![(\nu a)P]\!]\rho sh = [\![P]\!](\rho[a \mapsto a[s]])sh$$
$$[\![M(x).P]\!]\rho sh = \{h \Rightarrow \mathsf{com}(\rho(M))\} \cup [\![P]\!](\rho[x \mapsto x'])(s, x')(h \wedge \mathsf{mess}(\rho(M), x'))$$
$$[\![\overline{M}\langle N\rangle.P]\!]\rho sh = \{h \Rightarrow \mathsf{mess}(\rho(M), \rho(N)), h \Rightarrow \mathsf{com}(\rho(M))\} \cup [\![P]\!]\rho sh$$
$$[\![let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q]\!]\rho sh =$$
$$\cup \{[\![P]\!]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma s, \sigma' p')(\sigma h) \mid g(N_1, \ldots, N_n) \rightarrow N \text{ is in } \mathrm{def}(g) \text{ and } (\sigma, \sigma') \text{ is a most general pair of substitutions such that } \sigma\rho(M_1) = \sigma' N_1, \ldots, \sigma\rho(M_n) = \sigma' N_n\}$$
$$\cup \{h \wedge \mathsf{testunif}((\rho(M_1), \ldots, \rho(M_n)), EVar((N_1, \ldots, N_n))) \Rightarrow \mathsf{bad} \mid g(N_1, \ldots, N_n) \rightarrow N \text{ is in } \mathrm{def}(g)\} \cup [\![Q]\!]\rho sh$$

The environment ρ maps names and variables to their corresponding pattern. The sequence s contains all input messages, session identifiers, and results of destructor applications above the current program point. It is then used in the restriction case $(\nu a)P$, to build the pattern $a[s]$ corresponding to the name a. The sequence h contains all facts that must be true to reach the current program point.

The generated clauses are similar to those of [16], but clauses are added to indicate which tests the adversary can perform. Intuitively, when the protocol outputs a message M on channel M' after receiving messages $M_1, \ldots, M_n$ on channels $M'_1, \ldots, M'_n$, respectively, we generate a clause $h \Rightarrow \mathsf{mess}(p', p)$ with $h = \mathsf{mess}(p'_1, p_1) \wedge \ldots \wedge \mathsf{mess}(p'_n, p_n)$ where p, p', p_i, p'_i are the patterns corresponding to the terms M, M', M_i, M'_i respectively. This clause means that, if messages $p_1, \ldots, p_n$ are sent on channels $p'_1, \ldots, p'_n$ respectively, then the protocol may output p on channel p'. The mapping from terms to patterns is given by the environment ρ, which is extended by adding the image of the session identifier in the case of replication, the image of the new name in the case of restriction, and the image of the new variable in the case of destructor application. The hypothesis h is extended by adding the input message in the case of input. The clause $h \Rightarrow \mathsf{mess}(p', p)$ is generated in the output case. Similarly, when the protocol makes a communication (input or output) on channel M' after receiving the same messages, we generate a clause $h \Rightarrow \mathsf{com}(p')$, as can be seen on the input and output cases. At last, when the protocol executes a destructor application $g(M''_1, \ldots, M''_{n''})$ after receiving the same messages, we generate a clause $h \wedge \mathsf{testunif}((p''_1, \ldots, p''_{n''}), EVar((N_1, \ldots, N_{n''}))) \Rightarrow \mathsf{bad}$ for each $g(N_1, \ldots, N_{n''}) \rightarrow N$ in $\mathrm{def}(g)$, where p''_i is the pattern corresponding to the term M''_i. This clause means that, when the conditions in h are true, the attacker may know whether the destructor application succeeds. This clause serves in proving hypothesis 2 of Proposition 1.

The replication creates a new session identifier i, and its corresponding pattern i'. The hypothesis $\mathsf{att}(i')$ is added to h, to make sure that all variables in the conclusion of a clause are also in its hypothesis, as in Rule (Rn). (This detail was omitted in the intuitive explanation above.) The replication is otherwise ignored, because Horn clauses can be applied any number of times, so they are implicitly replicated.

For example, the destructor application $let\ y' = pdecrypt_{prob}(y, sk_B)$ in P_B in the example of Section 2 generates the clause:

$$\mathsf{testunif}((y, sk_B[\,]), (pencrypt_{prob}(\mathsf{e}, pk(\mathsf{e}'), \mathsf{e}''), \mathsf{e}')) \wedge \mathsf{mess}(c[\,], y) \Rightarrow \mathsf{bad}$$

When a message y is sent on c, it can be received by B, and the attacker might observe whether the decryption of y by sk_B succeeds. The second destructor application of P_B generates similar clauses. The last output of P_B generates the clauses

$$\mathsf{mess}(c[\,], pencrypt_{prob}(sign((pk(sk_A[\,]), pk(sk_B[\,]), x), sk_A[\,]), pk(sk_B[\,]))) \Rightarrow \mathsf{mess}(c[\,], sencrypt(\mathsf{x}, x))$$
$$\mathsf{mess}(c[\,], pencrypt_{prob}(sign((pk(sk_A[\,]), pk(sk_B[\,]), x), sk_A[\,]), pk(sk_B[\,]))) \Rightarrow \mathsf{com}(c[\,])$$

The first clause means that if $\{\{pk_A, pk_B, x\}_{sk_A}\}_{pk_B}$ is sent on channel c, it may be input by B, which is then going to reply with $\{\mathsf{x}\}_x$ on channel c. The second clause means that, in this case, a communication occurs on channel c. (When all communications occur on public channels, the clauses $h \Rightarrow \mathsf{com}(p)$ generated from the process are in fact useless.)

Proof of strong secrecy Let $\rho = \{a \mapsto a[\,] \mid a \in fn(P_0) \cup Secr\}$. Let σ_0 be a substitution mapping all free variables of P_0 to distinct elements of $Secr$, as in Proposition 1. We define the set of rules corresponding to process P_0 as:

$$\mathcal{R}_{P_0} = [\![\sigma_0 P_0]\!]\rho\emptyset\emptyset \cup \{(\text{Init}), (\text{Rn}), \ldots (\text{Rc}), (\text{Rcom})\}$$

We have proved the following result:

Proposition 2 *If* bad *is not derivable from $\mathcal{R}_{P_0}$, then P_0 preserves the strong secrecy of its free variables.*

Proof sketch We exploit the theory developed in [3, 16] to prove the hypotheses of Proposition 1. As in [16], we introduce *events* $\mathtt{end}(M)$ that can be used as witnesses that some part of a process has been executed. From the process P_0 and any adversary Q such that $fn(Q) \cap Secr = \emptyset$, we build processes P'_0 and Q' such that:

- If $\sigma_0 P_0 \mid Q$ can execute a destructor application $g(M_1, \ldots, M_n)$, then $P'_0 \mid Q'$ can execute the event $\mathtt{end}(\mathsf{test}_g(M_1, \ldots, M_n))$.
- If $\sigma_0 P_0 \mid Q$ can execute a communication (input or output) on channel M, then $P'_0 \mid Q'$ can execute the event $\mathtt{end}(\mathsf{com}(M))$.
- Q' is an adversary in the sense of [16], that is, it does not contain $\mathtt{end}$ events and $fn(Q') \cap Secr = \emptyset$.

We build P'_0 by adding in P_0 the required end events just before each destructor application and communication. Similarly, Q' is built from Q. Since Q' cannot execute events, Q' sends a message describing the end event to execute on a special channel before each destructor application and communication, and P'_0 includes a relay process that executes the corresponding end events when receiving the message.

Similarly to what has been done in [16], we can build Horn clauses $\mathcal{R}'$ such that if $P'_0 \mid Q'$ executes the event $\mathtt{end}(M)$ then a fact $\mathtt{end}(p)$ is derivable from the clauses, where p is the pattern corresponding to the term M after encoding names by functions. (The proof of the soundness of

the clauses is done as in [3], using a generic type system to express the soundness invariant, and a subject reduction theorem to show that the invariant is indeed preserved.) Moreover, the clauses $\mathcal{R}'$ are closely related to $\mathcal{R}_{P_0}$, namely:

- For each rule $h \Rightarrow \mathsf{end}(\mathsf{test}_g(p_1, \ldots, p_n))$ in $\mathcal{R}'$, for each $g(N_1, \ldots, N_n) \rightarrow N$ in $\mathrm{def}(g)$, there is a rule $h \wedge \mathsf{testunif}((p_1, \ldots, p_n), EVar((N_1, \ldots, N_n))) \Rightarrow \mathsf{bad}$ in $\mathcal{R}_{P_0}$.
- For each rule $h \Rightarrow \mathsf{end}(\mathsf{com}(p))$ in $\mathcal{R}'$, there is a rule $h \Rightarrow \mathsf{com}(p)$ in $\mathcal{R}_{P_0}$.
- The clauses that define the predicates att and mess (occurring in h) are the same in $\mathcal{R}_{P_0}$ and $\mathcal{R}'$.

Let us prove hypothesis 2 of Proposition 1. Assume that $\sigma_0 P_0 \mid Q$ executes a destructor application $g(M_1, \ldots, M_n)$ where $g(N_1, \ldots, N_n) \rightarrow N$ is in $\mathrm{def}(g)$. Then $P_0' \mid Q$ executes $\mathsf{end}(\mathsf{test}_g(M_1, \ldots, M_n))$, so $\mathsf{end}(\mathsf{test}_g(p_1, \ldots, p_n))$ is derivable from $\mathcal{R}'$, where $p_1, \ldots, p_n$ are the patterns corresponding to the terms $M_1, \ldots, M_n$ after encoding names by functions. If $\mathsf{testunif}((p_1, \ldots, p_n), EVar((N_1, \ldots, N_n)))$ was true, then bad would be derivable from $\mathcal{R}_{P_0}$, because of the relation between $\mathcal{R}'$ and $\mathcal{R}_{P_0}$. So $\mathsf{testunif}((p_1, \ldots, p_n), EVar((N_1, \ldots, N_n)))$ is false. From the definition of testunif and the relation between terms and patterns, we can then prove hypothesis 2 of Proposition 1. The proof of hypothesis 1 is similar, and we conclude by applying Proposition 1. □

5. Solving Algorithm

To determine whether a fact is derivable from the clauses, we use an algorithm based on resolution with free selection extending the one of [17]. (We use the meta-variables R, H, C, F for rule, hypothesis, conclusion, fact, respectively.)

The algorithm infers new clauses by resolution as follows. From two clauses $R = H \Rightarrow C$ and $R' = F \wedge H' \Rightarrow C'$ (where F is any hypothesis of R'), it infers $R \circ_F R' = \sigma H \wedge \sigma H' \Rightarrow \sigma C'$, where C and F are unifiable and σ is the most general unifier of C and F:

$$\frac{H \Rightarrow C \qquad F \wedge H' \Rightarrow C'}{\sigma H \wedge \sigma H' \Rightarrow \sigma C'}$$

The clause $R \circ_F R'$ is the combination of R and R', in which R proves the hypothesis F of R'. The resolution is guided by a selection function sel: $\mathsf{sel}(R)$ returns a subset of the hypotheses of R, and the resolution step above is performed only when $\mathsf{sel}(R) = \emptyset$ and $F \in \mathsf{sel}(R')$.

In this paper, we use the following selection function: When R is the clause

$$\mathsf{testunif}((x_1), (x_2)) \wedge \mathsf{att}(x_1) \wedge \mathsf{att}(x_2) \Rightarrow \mathsf{bad} \quad \text{(Ratteq)}$$

$\mathsf{sel}(R) = \{\mathsf{att}(x_1)\}$. (This clause is obtained from (Rt) for $g = eq$, by the simplification steps defined below.) For all other clauses, $\mathsf{sel}(R)$ is defined by

$$\mathsf{sel}(H \Rightarrow C) = \begin{cases} \emptyset & \text{if all elements of } H \text{ are of the form } \mathsf{att}(x), \\ & \quad x \text{ variable, or } \mathsf{testunif}(p, p') \\ \{F\} & \text{where } F \neq \mathsf{att}(x),\ F \neq \mathsf{testunif}(p, p'), \\ & \quad F \in H, \text{ otherwise} \end{cases}$$

In the last case, there may be several choices for F, we simply choose one. The selection function must never select testunif facts because they are not defined by clauses. It should not select $\mathsf{att}(x)$ because it would lead to nontermination ($\mathsf{att}(x)$ can be unified with all att facts, yielding many resolution steps). However, if $\mathsf{att}(x)$ was not selected in (Ratteq), this rule would appear unchanged after executing the algorithm, and we would not know whether bad is derivable using this rule or not.

The algorithm uses standard optimizations, such as the elimination of tautologies (performed by *elimtaut*), as well as more protocol-specific optimizations, such as the elimination of hypotheses $\mathsf{att}(x)$: *elimattx* removes hypotheses $\mathsf{att}(x)$ when x does not appear elsewhere in the clause. (These hypotheses are always true, since the attacker has at least one term.)

We need to extend the algorithm to determine whether $\mathsf{testunif}(p_1, p_2)$ is true or not. We achieve this goal by using a specific simplification technique. The simplifications have several goals: reduce the size of testunif facts, so that they do not grow indefinitely; make progress towards knowing the truth value of these facts, by instantiating the variables; simplify them out when their truth value is known. As an example, we consider the clause:

$$\begin{aligned} &\mathsf{testunif}((pencrypt_{prob}(z, x, r[z]), y), \\ &\quad (pencrypt_{prob}(\mathsf{e}, pk(\mathsf{e}'), \mathsf{e}''), \mathsf{e}')) \wedge \\ &\qquad \mathsf{att}(x) \wedge \mathsf{att}(y) \wedge \mathsf{att}(z) \Rightarrow \mathsf{bad} \end{aligned} \quad (1)$$

The following simplification steps take as input a set of clauses and return a set of clauses. Clauses to which the mentioned transformation does not apply are left unchanged.

- Unification: *unify* transforms clauses of the form $\mathsf{testunif}(p_1, p_2) \wedge H \Rightarrow C$ as follows. It tries to unify p_1 and p_2, considering elements of $Secr$ and $EVar$ as variables. If this unification fails, the clause is removed. Indeed, $\sigma p_1 \neq \sigma p_2$ for all values of σ, so $\mathsf{testunif}(p_1, p_2)$ is always false.

  Let σ_u be the most general unifier of p_1 and p_2. In this unification, σ_u is built such that all variables in its domain and its image are variables of p_1 and p_2, and the variables in its domain do not occur in its image.

If, for some $\mathsf{x} \in Secr$, $\sigma_u\mathsf{x}$ is a pattern that contains bound names, the clause is removed. Indeed, $\sigma\mathsf{x}$ can never be equal to such a pattern.

Otherwise, *unify* replaces the clause with

$$\mathsf{testunif}((x_1, \ldots, x_n), \sigma_u(x_1, \ldots, x_n)) \wedge H \Rightarrow C$$

where $x_1, \ldots, x_n$ are all variables modified by σ_u (these may include elements of $Secr$ and $EVar$). Indeed, an instance of the new testunif fact is true if and only if the same instance of the old one is, because for all σ of domain $Secr$ union $EVar$ union the variables, $\sigma p_1 = \sigma p_2$ if and only if $\sigma(x_1, \ldots, x_n) = \sigma\sigma_u(x_1, \ldots, x_n)$.

For example, *unify* transforms the clause (1) into the equivalent one:

$$\begin{array}{c}\mathsf{testunif}((z, x, \mathsf{e}'', y), (\mathsf{e}, pk(\mathsf{e}'), r[\mathsf{e}], \mathsf{e}')) \wedge \\ \mathsf{att}(x) \wedge \mathsf{att}(y) \wedge \mathsf{att}(z) \Rightarrow \mathsf{bad}\end{array} \quad (2)$$

- Swapping: *swap* transforms clauses of the form $\mathsf{testunif}((p_1, \ldots, p_n), (p'_1, \ldots, p'_n)) \wedge H \Rightarrow C$, obtained after *unify*. If $p_i \in Secr$ and $p'_i \in EVar$, it swaps p_i and p'_i everywhere in the testunif fact. Similarly, if p_i is a variable and $p'_i \in Secr \cup EVar$, it swaps p_i and p'_i everywhere in the testunif fact.

  Indeed, some instance of the new testunif fact is true if and only if the same instance of the old one is, since the unification constraints remain the same.

  For example, *swap* transforms the clause (2) into the following one, by swapping z and e, and y and e':

$$\begin{array}{c}\mathsf{testunif}((\mathsf{e}, x, \mathsf{e}'', \mathsf{e}'), (z, pk(y), r[z], y)) \wedge \\ \mathsf{att}(x) \wedge \mathsf{att}(y) \wedge \mathsf{att}(z) \Rightarrow \mathsf{bad}\end{array} \quad (3)$$

- Elimination of elements of $EVar$: *elimEVar* transforms clauses $\mathsf{testunif}((p_1, \ldots, p_n), (p'_1, \ldots, p'_n)) \wedge H \Rightarrow C$, obtained after *unify* and *swap*: if $p_i = \mathsf{e} \in EVar$, it eliminates the pair p_i, p'_i from the testunif fact.

  Indeed, e does not occur elsewhere in the testunif fact (this property comes from the result of *unify* and is preserved by *swap*). Then an instance of the new testunif fact is true if and only if the same instance of the old one is, by giving to e the same value as the value of p'_i.

  For example, *elimEVar* transforms the clause (3) into the following one, by removing the useless unification tests with elements of $EVar$:

$$\begin{array}{c}\mathsf{testunif}((x), (pk(y))) \wedge \\ \mathsf{att}(x) \wedge \mathsf{att}(y) \wedge \mathsf{att}(z) \Rightarrow \mathsf{bad}\end{array} \quad (4)$$

- Instantiation: *instantiate* transforms clauses of the form

$$\begin{array}{l}\mathsf{testunif}((\ldots, x, \ldots), (\ldots, f(p_1, \ldots, p_n), \ldots)) \\ \quad \wedge \mathsf{att}(x) \wedge H \Rightarrow \mathsf{bad}\end{array}$$

  where f is a constructor or a name function symbol, and x and $f(p_1, \ldots, p_n)$ are in matching positions in their respective tuples. It substitutes $f(x_1, \ldots, x_n)$ for x in the clause.

  Indeed, the fact $\mathsf{testunif}((\ldots, x, \ldots), (\ldots, f(p_1, \ldots, p_n), \ldots))$ can be true in two cases:

  1. When $\mathsf{att}(\mathsf{x})$ is derivable for some $\mathsf{x} \in Secr$, and we take $x = \mathsf{x}$ in the clause. This case is already detected since $\mathsf{att}(\mathsf{x})$ implies $\mathsf{com}(\mathsf{x})$ which implies bad.
  2. When x is of the form $f(p'_1, \ldots, p'_n)$. So we substitute $f(x_1, \ldots, x_n)$ for x in the clause.

  The whole simplification process will be repeated when *instantiate* modifies a clause, since new opportunities for simplifications appear. In particular, $f(x_1, \ldots, x_n)$ and $f(p_1, \ldots, p_n)$ will be replaced with $x_1, \ldots, x_n$ and $p_1, \ldots, p_n$ in the testunif fact by *unify*. Repeating the simplification does not yield an infinite loop, because the size of the testunif fact decreases.

  This transformation is important because it replaces a non-selectable fact $\mathsf{att}(x)$ with a selectable one $\mathsf{att}(f(x_1, \ldots, x_n))$. This is key for making sure that all clauses with conclusion bad and non-empty hypothesis have a selected hypothesis (see Lemma 3 below).

  For example, *instantiate* substitutes $pk(x')$ for x in the clause (4), yielding:

$$\begin{array}{c}\mathsf{testunif}((pk(x')), (pk(y))) \wedge \\ \mathsf{att}(pk(x')) \wedge \mathsf{att}(y) \wedge \mathsf{att}(z) \Rightarrow \mathsf{bad}\end{array} \quad (5)$$

  This clause will then be transformed into

$$\mathsf{testunif}((x'), (y)) \wedge \mathsf{att}(pk(x')) \wedge \mathsf{att}(y) \Rightarrow \mathsf{bad} \quad (6)$$

  by applying *unify* again and *elimattx*. The selected hypothesis will be $\mathsf{att}(pk(x'))$.

- Elimination of useless variables: *elimvar* transforms clauses of the form

$$\begin{array}{c}\mathsf{testunif}((\ldots, x_1, \ldots), (\ldots, x_2, \ldots)) \wedge \\ \mathsf{att}(x_1) \wedge \mathsf{att}(x_2) \wedge H \Rightarrow \mathsf{bad}\end{array}$$

  where x_1 and x_2 appear in matching positions in the tuples in testunif, and the clause is not (Ratteq). It substitutes x_1 for x_2 in the clause.

  Indeed, if the old rule derives bad, then either it derives bad with $x_1 = x_2$, in which case the new one derives bad, or it derives bad with $x_1 \neq x_2$, in which case (Ratteq) does. Conversely, if the new rule derives bad, then the old one does, by taking $x_1 = x_2$.

The whole simplification process will be repeated when *elimvar* modifies a clause, since new opportunities for simplifications appear. In particular, the matching occurrences of x_1 and x_2 (transformed into x_1 by *elimvar*) in the testunif fact will be removed by *unify*.

The function *elimvar* leaves (6) unchanged, since the hypothesis does not contain $\mathsf{att}(x')$.

- Detect testunif false: *elimtestuniffalse* removes clauses of the form $\mathsf{testunif}((),()) \wedge H \Rightarrow C$. Indeed, $\mathsf{testunif}((),())$ is false.
- Detect testunif true: *simptestuniftrue* transforms clauses of the form

$$\mathsf{testunif}((p_1,\ldots,p_n),(p'_1,\ldots,p'_n)) \wedge H \Rightarrow \mathsf{bad}$$

obtained after the previous simplification steps, where H contains hypotheses of the form $\mathsf{att}(x)$ and for all $i \in \{1,\ldots,n\}$, $p_i \in \mathit{Secr}$. It replaces the clause with the clause bad.

Indeed, we can derive bad as follows. Since the attacker has at least one term, we can choose the values of the variables to satisfy H. Let σ_1 be a substitution such that $\sigma_1 H$ is true. Let us show that $\sigma_1\mathsf{testunif}((p_1,\ldots,p_n),(p'_1,\ldots,p'_n)) = \mathsf{testunif}((p_1,\ldots,p_n),\sigma_1(p'_1,\ldots,p'_n))$ is true. We simply define σ by $\sigma p_i = \sigma_1 p'_i$ and σ maps all other elements of Secr and EVar to some constant. Then $\sigma(p_1,\ldots,p_n) = \sigma\sigma_1(p'_1,\ldots,p'_n)$. Moreover, there exists no σ' of domain EVar such that $(p_1,\ldots,p_n) = \sigma'\sigma_1(p'_1,\ldots,p'_n)$, because by the previous application of *swap*, p'_i is not in EVar, so $\sigma_1 p'_i$ is not is EVar: it is either a complex term or an element of Secr different from p_i, then so is $\sigma'\sigma_1 p'_i$. Then the testunif fact is true. Then all hypotheses of the clause are true, and thus we can derive bad.

We can finally group all simplifications together:

- We define the simplification function $\mathit{simplify} = \mathit{elimtaut} \circ \mathit{elimattx} \circ \mathit{simptestuniftrue} \circ \mathit{elimtestuniffalse} \circ \mathit{repeat}(\mathit{elimvar} \circ \mathit{instantiate} \circ \mathit{elimEVar} \circ \mathit{swap} \circ \mathit{unify})$. The expression $\mathit{repeat}(f)$ means that the application of function f is repeated until a fixpoint is obtained, that is, $f(\mathcal{R}) = \mathcal{R}$. It is enough to repeat the simplification only when *elimvar* or *instantiate* have modified the set of clauses. Indeed, no new simplification would be done in the other cases. The repetition never leads to an infinite loop, because the size of the testunif fact decreases at each iteration.
- $\mathit{condense}(\mathcal{R})$ applies *simplify* to $\mathcal{R}$ and then eliminates subsumed clauses. We say that $H_1 \Rightarrow C_1$ subsumes $H_2 \Rightarrow C_2$ if and only if there exists a substitution σ such that $\sigma C_1 = C_2$ and $\sigma H_1 \subseteq H_2$. If $\mathcal{R}$ contains clauses R and R', such that R subsumes R', R' is removed. (In that case, R can do all derivations that R' can do.)

We now define the algorithm $\mathsf{saturate}(\mathcal{R}_0)$. Starting from $\mathit{condense}(\mathcal{R}_0)$, the algorithm adds clauses inferred by resolution with the selection function sel and condenses the clause set at each iteration step until a fixed point is reached. When a fixpoint is reached, $\mathsf{saturate}(\mathcal{R}_0)$ is the set of clauses R in the obtained clause set such that $\mathsf{sel}(R) = \emptyset$.

By adapting the proof of [15] to this algorithm and proving the soundness of all simplifications for testunif, we have shown that, for any closed fact F, F is derivable from $\mathcal{R}_{P_0}$ if and only if it is derivable from $\mathsf{saturate}(\mathcal{R}_{P_0})$.

The following lemma shows that, at the end of the algorithm, the only remaining clause that can contain bad is bad itself. So bad is derivable from $\mathsf{saturate}(\mathcal{R}_{P_0})$ if and only if the clause bad is in $\mathsf{saturate}(\mathcal{R}_{P_0})$. This lemma strongly relies on the simplification process for testunif facts.

Lemma 3 *After simplification, if* $R = H \Rightarrow \mathsf{bad}$ *and* H *is not empty, then* $\mathsf{sel}(R) \neq \emptyset$.

Proposition 4 *The above algorithm generates the clause* bad *if and only if* bad *is derivable from* $\mathcal{R}_{P_0}$. *So, if it does not generate the clause* bad, *then* P_0 *preserves the strong secrecy of its free variables.*

Proof sketch We define a derivation of a closed fact F from a set of clauses $\mathcal{R}_{P_0}$ as a finite sequence of pairs (R_i, F_i) for $i \in \{1,\ldots,n\}$, such that $F_n = F$ and for all i, $R_i \in \mathcal{R}_{P_0}$, and by using R_i, we can derive F_i from the facts already derived before F_i. More formally, for some instance σR_i of R_i, the hypotheses of σR_i are $F_{j_1},\ldots,F_{j_m}$ with $j_1,\ldots,j_m < i$ or are true testunif facts, and the conclusion of σR_i is F_i. When $j_1,\ldots,j_m$ are minimal satisfying the previous condition, we define $\mathrm{measure}(D,i)$ as the multiset $\{j_1,\ldots,j_m\}$. The measures are ordered by the multiset ordering. So the measure is smaller when the hypotheses of R_i are proved earlier in the derivation. We prove the following results:

- Consider a derivation D containing a step $(R_{i'}, F_{i'})$. Let F_0 be an hypothesis of $R_{i'}$ (not a testunif fact). Then there exists a step (R_i, F_i) with $i < i'$, such that F_i is an instance of F_0, $R_i \circ_{F_0} R_{i'}$ is defined, one obtains a derivation D' by replacing the step $(R_{i'}, F_{i'})$ with the step $(R_i \circ_{F_0} R_{i'}, F_{i'})$ in D, and $\mathrm{measure}(D', i') < \mathrm{measure}(D, i')$.
- If D is a derivation whose step i is (R, F_i), one obtains a derivation D' by replacing R with a clause R' that subsumes R. Moreover $\mathrm{measure}(D', i) \leq \mathrm{measure}(D, i)$.

- Let f be any of the above simplification functions.

  If D is a derivation whose step i is (R, F_i), then there exists $R' \in f(\{R\}) \cup \{\text{(Ratteq)}\} \cup \{\mathsf{att}(\mathsf{x}) \Rightarrow \mathsf{bad} \mid \mathsf{x} \in Secr\}$ such that one obtains a derivation D' of the same fact by replacing R with R'. Moreover, $\text{measure}(D', i) \leq \text{measure}(D, i)$.

  Conversely, if D' contains a clause $R' \in f(\{R\})$, then there exists a derivation D formed from D' by replacing R' with R and perhaps adding rules in $\mathcal{R}_0$, that derives the same fact as D'.

  The proof of these results follows closely the intuition for correctness of each simplification function given above. The clause (Ratteq) is useful for $f = elimvar$ and $\mathsf{att}(\mathsf{x}) \Rightarrow \mathsf{bad}$ is useful for $f = instantiate$. By combining these results, we obtain the same result for $f = simplify$.

- F is derivable from $\mathcal{R}_0$ if and only if it is derivable from $\mathsf{saturate}(\mathcal{R}_0)$. The key idea of the proof of the direct part is to replace rules as allowed by the previous results. When the replacement terminates, we can show that all rules are in $\mathsf{saturate}(\mathcal{R}_0)$. We show the termination using the decrease of $\text{measure}(D, i)$, since the ordering of measures is well-founded. For the converse, we perform the replacement in the other direction.

The proposition is then easy to infer from the last result and Lemma 3 (since all clauses in $\mathsf{saturate}(\mathcal{R}_0)$ have empty selection). □

6. Extensions

Limiting the Set of Values of Secrets Let $fv(P_0) = \{x_1, \ldots, x_n\}$. We now assume that the secrets $x_1, \ldots, x_n$ are known to be in sets $S_1, \ldots, S_n$ respectively. That is, we want to prove that, for all σ and σ' such that $\sigma x_i \in S_i$ and $\sigma' x_i \in S_i$, we have $\sigma P_0 \approx \sigma' P_0$. When the sets S_i are small, using strong secrecy is particularly important, since the adversary might quickly obtain the value of the secret by exploiting the information obtained from tests. (Each test whose result depends on the secrets can leak one bit of information.) We use the same reasoning as before, but we update Definition 3, such that only substitutions mapping x_i to an element of S_i are considered, and we refine $simptestuniftrue$. When for all θ such that $\theta\mathsf{x}_i \in S_i$, θp_1 does not unify with θp_2 considering elements of $EVar$ as variables, we remove clauses containing $\mathsf{testunif}(p_1, p_2)$. When a clause is $\mathsf{testunif}((p_1, \ldots, p_{n'}), (p'_1, \ldots, p'_{n'})) \wedge H \Rightarrow \mathsf{bad}$ and for all j, $p_j \in Secr$, we replace this clause with clauses $\sigma_u H \Rightarrow \mathsf{bad}$, where σ_u is the most general unifier of $\theta(p_1, \ldots, p_{n'})$ and $\theta(p'_1, \ldots, p'_{n'})$ for some θ such that $\theta\mathsf{x}_i \in S_i$, considering elements of $EVar$ as variables.

We can also extend this to the case in which the secrets can take values among bound names. That is, we prove $(\nu a_1) \ldots (\nu a_n)\sigma P_0 \approx (\nu a_1) \ldots (\nu a_n)\sigma' P_0$, so that $a_1, \ldots, a_n$ may occur in the image of σ and σ'. We proceed as for proving $\sigma P_0 \approx \sigma' P_0$, except that the adversary does not have $a_1, \ldots, a_n$: the clause $\mathsf{att}(a_i[\,])$ for $i \in \{1, \ldots, n\}$ is removed.

Treatment of Equations We have extended our technique to the case when cryptographic primitives are defined by equations. For example, Diffie-Hellman key agreements can be modeled using two constructors f and g, with the equation $f(x, g(y)) = f(y, g(x))$ [5]. We encode equations with destructors, as in [15]. For instance, the previous equation can be modeled by having a destructor f defined by $f(x, g(y)) \to f'(y, g(x))$ and $f(x, y) \to f(x, y)$. This encoding is sound for strong secrecy. (This technique is limited to rather simple equations; more complex ones could be handled by unification modulo the equational theory.)

7. Experimental Results

The verifier shows automatically that the protocol of Section 2 preserves the strong secrecy of x. As further small examples, we can consider variants of this protocol. If we add a Message 2': $A \to B : \{x'\}_k$, the verifier finds an attack against strong secrecy of x and x'. Precisely, it finds that bad is derivable, and the derivation of bad describes an attack, which may in general be a false attack because of our approximations. In this example, it is a real attack. Indeed, an attacker can test whether $x = x'$ by comparing the ciphertexts $\{x\}_k$ and $\{x'\}_k$. In contrast, if we replace shared-key encryption with probabilistic shared-key encryption, the verifier now shows that the protocol preserves the strong secrecy of x and x'. If we replace Messages 2 and 2' with $\{c_0, x\}_k$ and $\{c'_0, x'\}_k$ respectively, the verifier shows that the protocol preserves the strong secrecy of x and x', both with deterministic and probabilistic shared-key encryption. Indeed, the ciphertexts are always different, even when $x = x'$.

We have also proved strong secrecy results for the protocols of Otway-Rees [38], Yahalom [21], and Skeme [32]. The total runtime for these tests is less than 2 s on an Intel Xeon 1.7 GHz.

As a more substantial example, we have proved properties of the JFK protocol [11]. This protocol is a proposed replacement for IKE, the key exchange protocol of IPsec. It allows an initiator I to establish a security association (in particular, a session key) with a responder R. We have used the technique described in this paper for proving identity protection properties of JFK. The protocol has two variants, JFKi and JFKr, that differ by these properties:

1. JFKi guarantees that an active attacker cannot know the identity of the initiator (when it accepts connec-

tions only with honest principals). Accordingly, we show the observational equivalence of configurations in which two initiators I_1 and I_2 have their secret keys among sk_1 and sk_2 (so their public keys, which we identify with their identities, among $pk(sk_1)$ and $pk(sk_2)$). Hence, an adversary cannot distinguish a configuration in which an initiator uses sk_1 from one in which it uses sk_2, and it cannot tell whether two initiators use the same secret key. Technically, using the extension of Section 6, we show that $(\nu sk_1)(\nu sk_2)\sigma P_0 \approx (\nu sk_1)(\nu sk_2)\sigma' P_0$ where the process P_0 models the protocol, σ and σ' map the variables sk_{I_1} and sk_{I_2}, secret keys of I_1 and I_2 respectively, to elements of $\{sk_1, sk_2\}$.

Importantly, the attacker knows all public keys of the principals ($pk(sk_1)$, $pk(sk_2)$, ...), but we show that it does not know which ones the principals actually use. Such a property cannot be satisfactorily approximated by a standard secrecy property.

2. JFKr protects the identity of responders against active attacks. Accordingly, we show the observational equivalence of configurations in which two responders R_1 and R_2 have their secret keys among sk_1 and sk_2.

   JFKr also protects all identities against passive attacks. So, we show the observational equivalence of configurations in which two responders and two initiators can have their secret keys among $sk_1, \ldots, sk_4$, and the attacker can listen but not send messages.

These configurations of course contain other responders and initiators, with other keys. For simplicity, for JFKi, all initiators (resp. for JFKr, all responders) accept connections only from honest principals. These tests took 2 s for JFKi and 1 min 30 s for JFKr, on an Intel Xeon 1.7 GHz. The case study of the JFK protocol has been done in collaboration with Martín Abadi and Cédric Fournet [4].

The verifier is available at `http://www.di.ens.fr/~blanchet/crypto-eng.html` and more information on the study of the JFK protocol is available at `http://www.di.ens.fr/~blanchet/crypto/jfk.html`.

8. Conclusion

This work provides an efficient, automatic technique for proving a particular case of observational equivalence for cryptographic protocols, namely strong secrecy. It is much more flexible than the rare previous works that already tackled this problem, and has already been useful in the study of real protocols such as JFK.

For future work, it would be interesting to verify more general classes of process equivalences, since they can be used to formalize a wide variety of security properties. This is however a difficult task, especially when the equivalent processes have a different structure.

Acknowledgments

We would like to thank Martín Abadi, Jérôme Feret, David Monniaux, Andrew Myers, and the anonymous reviewers for very helpful comments on this work.

References

[1] M. Abadi. Secrecy by Typing in Security Protocols. In *Theoretical Aspects of Computer Software*, volume 1281 of *LNCS*, pages 611–638. Springer, 1997.

[2] M. Abadi. Security Protocols and their Properties. In *Foundations of Secure Computation*, NATO Science Series, pages 39–60. IOS Press, 2000. Volume for the 20th International Summer School on Foundations of Secure Computation, held in Marktoberdorf, Germany (1999).

[3] M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 33–44, Jan. 2002.

[4] M. Abadi, B. Blanchet, and C. Fournet. Just Fast Keying in the Pi Calculus. In *Programming Languages and Systems: 13th European Symposium on Programming (ESOP'04)*, LNCS. Springer, Mar. 2004.

[5] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, Jan. 2001.

[6] M. Abadi and A. D. Gordon. A Bisimulation Method for Cryptographic Protocols. *Nordic Journal of Computing*, 5(4):267–303, Winter 1998.

[7] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, Jan. 1999.

[8] M. Abadi and J. Jürjens. Formal Eavesdropping and its Computational Interpretation. In *Theoretical Aspects of Computer Software (TACS'01)*, volume 2215 of *LNCS*, pages 82–94. Springer, Oct. 2001.

[9] M. Abadi and R. Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.

[10] M. Abadi and P. Rogaway. Reconciling two views of cryptography (The computational soundness of formal encryption). In *First IFIP International Conference on Theoretical Computer Science*, volume 1872 of *LNCS*, pages 3–22. Springer, Aug. 2000.

[11] W. Aiello, S. Bellovin, M. Blaze, R. Canetti, J. Ionnidis, A. Keromytis, and O. Reingold. Efficient, DoS-resistant, Secure Key Exchange for Internet Protocols. In *ACM Conference on Computer and Communications Security (CCS'02)*, pages 48–58. ACM, Nov. 2002.

[12] M. Backes and B. Pfitzmann. Computational probabilistic non-interference. In *7th European Symposium on Research in Computer Security (ESORICS '02)*, volume 2502 of *LNCS*, pages 1–23. Springer, Oct. 2002.

[13] M. Backes and B. Pfitzmann. Intransitive Non-Interference for Cryptographic Purposes. In *24th IEEE Symposium on Security and Privacy*, pages 140–152. IEEE, May 2003.

[14] M. Backes, B. Pfitzmann, and M. Waidner. A Composable Cryptographic Library with Nested Operations. In *10th ACM conference on Computer and communication security (CCS'03)*, pages 220–230. ACM, Oct. 2003.

[15] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, June 2001.

[16] B. Blanchet. From Secrecy to Authenticity in Security Protocols. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 342–359. Springer, Sept. 2002.

[17] B. Blanchet and A. Podelski. Verification of Cryptographic Protocols: Tagging Enforces Termination. In *Foundations of Software Science and Computation Structures (FoSSaCS'03)*, volume 2620 of *LNCS*, pages 136–152. Springer, Apr. 2003.

[18] M. Boreale, R. De Nicola, and R. Pugliese. Proof Techniques for Cryptographic Processes. *SIAM Journal on Computing*, 31(3):947–986, 2002.

[19] J. Borgström and U. Nestmann. On Bisimulations for the Spi Calculus. In *Algebraic Methodology and Software Technology: 9th International Conference, AMAST 2002*, volume 2422 of *LNCS*, pages 287–303. Springer, Sept. 2002.

[20] M. Bugliesi, A. Ceccato, and S. Rossi. Non-Interference Proof Techniques for the Analysis of Cryptographic Protocols. In *Workshop on Issues in the Theory of Security (WITS'03)*, pages 42–51, Apr. 2003.

[21] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

[22] E. Cohen. Proving Protocols Safe from Guessing. In *Foundations of Computer Security*, July 2002.

[23] R. Corin, S. Malladi, J. Alves-Foss, and S. Etalle. Guess What? Here is a New Tool that Finds some New Guessing Attacks. In *Workshop on Issues in the Theory of Security (WITS'03)*, Apr. 2003.

[24] M. Dam and P. Giambiagi. Confidentiality for Mobile Code: The Case of a Simple Payment Protocol. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 233–244, July 2000.

[25] R. A. DeMillo, N. A. Lynch, and M. J. Merritt. Cryptographic Protocols. In *14th ACM Symposium on Theory of Computing*, pages 383–400. ACM, May 1982.

[26] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, Mar. 1983.

[27] L. Durante, R. Sisto, and A. Valenzano. A State-Exploration Technique for Spi-Calculus Testing-Equivalence Verification. In *Formal Techniques for Distributed System Development, FORTE/PSTV*, volume 183 of *IFIP Conference Proceedings*, pages 155–170. Kluwer, Oct. 2000.

[28] 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.

[29] R. Focardi, R. Gorrieri, and F. Martinelli. Non Interference for the Analysis of Cryptographic Protocols. In *Automata, Languages and Programming, 27th International Colloquium, ICALP'00*, volume 1853 of *LNCS*, pages 354–372. Springer, July 2000.

[30] P. Giambiagi and M. Dam. On the Secure Implementation of Security Protocols. In *Programming Languages and Systems: 12th European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 144–158. Springer, Apr. 2003.

[31] S. Goldwasser and S. Micali. Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information. In *14th Annual ACM Symposium on Theory of Computing*, pages 365–377. ACM, May 1982.

[32] H. Krawczyk. SKEME: A Versatile Secure Key Exchange Mechanism for Internet. In *Internet Society Symposium on Network and Distributed Systems Security*, Feb. 1996. Available at `http://bilbo.isu.edu/sndss/sndss96.html`.

[33] P. Laud. Handling Encryption in an Analysis for Secure Information Flow. In *Programming Languages and Systems, 12th European Symposium on Programming, ESOP'03*, volume 2618 of *LNCS*, pages 159–173. Springer, Apr. 2003.

[34] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.

[35] G. Lowe. Analyzing Protocols Subject to Guessing Attacks. In *Workshop on Issues in the Theory of Security (WITS'02)*, Jan. 2002.

[36] F. Martinelli, M. Petrocchi, and A. Vaccarelli. Analysing EMSS with Compositional Proof Rules for Non-Interference. In *Workshop on Issues in the Theory of Security (WITS'03)*, pages 52–53, Apr. 2003.

[37] A. C. Myers and B. Liskov. Complete, Safe Information Flow with Decentralized Labels. In *IEEE Symposium on Security and Privacy*, pages 186–197, May 1998.

[38] D. Otway and O. Rees. Efficient and Timely Mutual Authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[39] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[40] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[41] E. Sumii and B. C. Pierce. Logical Relations for Encryption. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 256–269, June 2001.

[42] E. Sumii and B. C. Pierce. A Bisimulation for Dynamic Sealing. In *31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 161–172, Jan. 2004.

[43] D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4:167–187, 1996.

[44] S. Zdancewic and A. Myers. Robust Declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 15–23, June 2001.