I N F O R M A T I K

Automatic Proof of Strong Secrecy for Security Protocols

Bruno Blanchet

MPI–I–2004–NWG1-001 July 2004

FORSCHUNGSBERICHT RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Stuhlsatzenhausweg 85 66123 Saarbrücken Germany

Author's Address

Bruno Blanchet
CNRS
Département d'Informatique
École Normale Supérieure
45 rue d'Ulm
F-75005 Paris, France
email: Bruno.Blanchet@ens.fr

and

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
D-66123 Saarbrücken, Germany
email: Bruno.Blanchet@mpi-sb.mpg.de

Publication Notes

An extended abstract of this report was published in the proceedings of the 2004 IEEE Symposium on Security and Privacy, Oakland, California, May 2004.

Acknowledgements

We would like to thank Martín Abadi, Jérôme Feret, David Monniaux, Andrew Myers, and the anonymous reviewers of the IEEE Symposium on Security and Privacy for very helpful comments on this work. We also thank Hans de Nivelle for his careful reading of this report and his helpful comments.

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. Applying this technique to strong secrecy 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 [10] (a proposed replacement for IKE in IPsec).

Keywords

Cryptographic Protocols; Verification; Secrecy; Horn Clauses; Resolution

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 call *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 [36], theorem proving [42], typing and logic programming [3] for instance.

- The second one, which we call *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 [46]. 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 [6], 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 6.1 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 him the correct key and a process that gives him 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 [7, 9], 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 [10] (a proposed replacement for IKE in IPsec). We have also proved the termination of the algorithm for the large class of *tagged protocols*, thus extending [17] to strong secrecy.

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 [46]. These type systems are often very restrictive. We refer to [43] 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 [40] 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 [47]. 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 [5, 6]. Other works have continued in this line of research, for example [18–20, 29, 39], 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 [44, 45]. 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 rewrite rules; it gives full freedom for building messages by applying the primitives.

Following Lowe [37], 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, he guesses the secret (presumably by exhaustive enumeration) and checks his 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 his 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, 37] might be more appropriate.

Although most works on automatic verification of protocols are based on formal models, Laud [34] 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 [7, 9]. Recently, Laud presented an extension to active attacks, with a bounded number of sessions [35]. 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 [7, 9], 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 states our termination result. Section 8 presents some of our experimental results and Section 9 concludes. Proofs are given in the appendix because of their length.

2 The Process Calculus

This section presents the process calculus that we use to represent protocols. It is essentially the calculus of [16], with two minor additions: functions can be private (not known by the adversary) and we model probabilistic cryptographic primitives.

$M, N ::=$	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
$\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$		destructor application
$\mathit{let}\ x = M\ \mathit{in}\ P$		local binding
$\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$		conditional

Figure 1: Syntax of the process calculus

2.1 Syntax and informal semantics

Figure 1 gives the syntax of terms (messages) and processes (programs) of our calculus. It assumes infinite sets of names 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 of the form $f(M_1, \ldots, M_n)$. Destructors do not appear in terms, but manipulate terms in processes. They are partial functions on terms that processes can apply. The process $\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$ 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 rewrite rules $g(N_1, \ldots, N_n) \to N$, 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 exist a substitution σ and a rewrite rule $g(N_1, \ldots, N_n) \to N$ 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; the adversary may obtain it by computation from public messages, however); the process $M(x).P$ waits for 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 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) 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.) The random coins R enable us to model a probabilistic encryption algorithm using a deterministic function $sencrypt_{prob}$. Probabilistic encryption should be used as follows: (νr)*let* $x = sencrypt_{prob}(m, k, r)$ *in* $\ldots$ where the coins r are fresh and used in a single encryption. We can 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 the encryption of m under pk, $\{m\}_{pk}$, to the adversary allows him to compare m with any message m' he 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.

Constructors and destructors can be public or private. The public ones can be used by the adversary, which is the case when not stated otherwise. The private ones can be used only by honest participants. They are useful in practice to model tables of keys stored in a server, for instance. A public constructor *host* computes a host name from a long-term secret key, and a private destructor *getkey* returns the key from the host name, and simulates a lookup in a table of pairs (host name, key). Using a public constructor *host* allows the adversary to create and register any number of host names and keys. However, since *getkey* is private, the adversary cannot compute a key from the host name, which would break all protocols: host names are public while keys of honest participants are secret.

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 vari-

Tuples:
Constructor: tuple $(M_1, \ldots, M_n)$
Destructors: projections $ith_n((M_1, \ldots, M_n)) \rightarrow M_i$
Shared-key encryption:
Constructor: encryption of M under key N, $sencrypt(M, N)$
Destructor: decryption $sdecrypt(sencrypt(M, N), N) \rightarrow M$
Probabilistic shared-key encryption:
Constructor: encryption of M under key N with random coins R,
$sencrypt_{prob}(M, N, R)$
Destructor: decryption $sdecrypt_{prob}(sencrypt_{prob}(M, N, R), N) \rightarrow M$
Probabilistic public-key encryption:
Constructors: encryption of M under public key N with random coins R,
$pencrypt_{prob}(M, N, R)$
public key generation from a secret key N, $pk(N)$
Destructor: decryption $pdecrypt_{prob}(pencrypt_{prob}(M, pk(N), R), N) \rightarrow M$
Signatures:
Constructor: signature of M with the secret key N, $sign(M, N)$
Destructors: verification $checksign(sign(M, N), pk(N)) \rightarrow M$
message without signature $getmess(sign(M, N)) \rightarrow M$
Non-message-revealing signatures:
Constructors: signature of M with the secret key N, $nmrsign(M, N)$
constant $true$
Destructor: verification $nmrchecksign(nmrsign(M, N), pk(N), M) \rightarrow true$
One-way hash functions:
Constructor: hash function $H(M)$
Message authentication codes, keyed hash functions:
Constructor: mac of M with key N, $mac(M, N)$
Table of host names and keys
Constructor: host name from key $host(M)$
Private destructor: key from host name $getkey(host(M)) \rightarrow M$

Figure 2: Constructors and destructors

$$
\begin{array}{ll}
P \mid 0 \equiv P & P \equiv Q \;\Rightarrow\; P \mid R \equiv Q \mid R \\
P \mid Q \equiv Q \mid P & P \equiv Q \;\Rightarrow\; !P \equiv !Q \\
(P \mid Q) \mid R \equiv P \mid (Q \mid R) & P \equiv Q \;\Rightarrow\; (\nu a)P \equiv (\nu a)Q \\
!P \equiv P \mid !P & P \equiv P \\
(\nu a_1)(\nu a_2)P \equiv (\nu a_2)(\nu a_1)P & Q \equiv P \;\Rightarrow\; P \equiv Q \\
(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q \;\text{ if } a \notin fn(P) & P \equiv Q,\, Q \equiv R \;\Rightarrow\; P \equiv R
\end{array}
$$

$$
\begin{array}{ll}
\overline{a}\langle M\rangle.Q \mid a(x).P \;\to\; Q \mid P\{M/x\} & \text{(Red I/O)} \\
\mathit{let}\ x = g(M_1,\ldots,M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \to P\{M'/x\} & \\
\quad \text{if } g(M_1,\ldots,M_n) \to M' & \text{(Red Destr 1)} \\
\mathit{let}\ x = g(M_1,\ldots,M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \to Q & \\
\quad \text{if there is no } M' \text{ such that } g(M_1,\ldots,M_n) \to M' & \text{(Red Destr 2)} \\
P \;\to\; Q \;\Rightarrow\; P \mid R \;\to\; Q \mid R & \text{(Red Par)} \\
P \;\to\; Q \;\Rightarrow\; (\nu a)P \;\to\; (\nu a)Q & \text{(Red Res)} \\
P' \equiv P,\, P \;\to\; Q,\, Q \equiv Q' \;\Rightarrow\; P' \;\to\; Q' & \text{(Red } \equiv\text{)}
\end{array}
$$

Figure 3: Structural equivalence and reduction

ables. Similarly, a closed (or ground) term is a term without variables and a closed substitution is a subsitution whose image consists of closed terms. We identify processes up to renaming of bound names and variables.

2.2 Formal semantics

As usual, the semantics for this calculus is defined by a reduction relation $\to$ and a structural equivalence relation $\equiv$, which is used to transform processes so that the reduction rules can be applied. These standard relations are recalled in Figure 3. Both $\equiv$ and $\to$ are defined only on closed processes.

2.3 Example

As an example, we consider the following protocol inspired by the corrected version of the Denning-Sacco key distribution protocol [8]:

$$
\begin{array}{llll}
\text{Message 1.} & A \to B: & \{\{pk_A, pk_B, k\}_{sk_A}\}_{pk_B} & k \text{ fresh} \\
\text{Message 2.} & B \to A: & \{x\}_k &
\end{array}
$$

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 k as a fresh key, 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:

$$
\begin{aligned}
&P_A(pk_A, sk_A) \stackrel{\text{def}}{=} c(x\_pk_B).(\nu k)(\nu r) \\
&\quad \overline{c}\langle pencrypt_{prob}(sign((pk_A, x\_pk_B, k), sk_A), x\_pk_B, r)\rangle. \\
&\quad c(x').let\ s = sdecrypt(x', k)\ in\ 0
\end{aligned}
$$

$$
\begin{aligned}
&P_B(pk_B, sk_B, pk_A) \stackrel{\text{def}}{=} c(y).let\ y' = pdecrypt_{prob}(y, sk_B)\ in \\
&\quad let\ (= pk_A, = pk_B, k) = checksign(y', pk_A)\ in\ \overline{c}\langle sencrypt(x, k)\rangle
\end{aligned}
$$

$$
\begin{aligned}
&P_0 \stackrel{\text{def}}{=} (\nu sk_A)(\nu sk_B) let\ pk_A = pk(sk_A)\ in\ let\ pk_B = pk(sk_B)\ in \\
&\quad \overline{c}\langle pk_A\rangle \overline{c}\langle pk_B\rangle.(!P_A(pk_A, sk_A)\ |!P_B(pk_B, sk_B, pk_A))
\end{aligned}
$$

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 public 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 $let\ (= pk_A, = pk_B, k) = \dots$ 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 will present more subtle results in Section 8.

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 [4].

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 $P \to P'$ then there exists Q' such that $Q \to^* Q'$ 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 non-deterministically 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: $C = [\,] \mid c(x).d(y).\mathit{if}\ x = y\ \mathit{then}\ \overline{e}\langle x\rangle$. 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 assume that a ciphertext that the adversary cannot decrypt gives no information on the cleartext. This problem could be solved by concatenating random bitstrings of random length to the ciphertexts. It could also 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 $fv(P_0)$, $\sigma P_0 \approx \sigma' P_0$. (Substitutions avoid 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 obtain 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.

The following proposition gives a simple criterion 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 $\sigma_0 P_0 \mid Q \ (\rightarrow \cup \equiv)^* \ C[let\ y = g(M_1, \dots, M_n)\ in\ Q'\ else\ R']$, 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, $g(N_1, \dots, N_n) \rightarrow N$ is in $\mathrm{def}(g)$, and $\theta(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$, then $(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$.*

Then P_0 preserves the strong secrecy of its free variables.

This result is proved in Appendix A.

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 rest of the paper. The substitution σ_0 maps the free variables of P_0 (which represent the secrets) to names in *Secr*. The process $\sigma_0 P_0$ is then closed, and can therefore be reduced. (Recall that the reduction rules of the semantics apply only to closed processes.) The second hypothesis means that, if the destructor application $g(M_1, \dots, M_n)$ succeeds for the values of the secrets given by θ (that is, $\theta(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$), then it must succeed for any value of the secrets (which is true when $(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$).

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 $\mathsf{x} \in \mathit{Secr}$ 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 fewer 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. (See Section 6.3 for a formal statement of this result.)

The same result also holds using may testing equivalence instead of observational equivalence, as in [1]. Indeed, observational equivalence implies may testing equivalence.

4 From the Process Calculus to Horn Clauses

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 of 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 Horn clauses are called *patterns* p and are generated by the following grammar:

$p ::=$	patterns
$\quad x, y, z$	$\quad$ variable
$\quad \mathsf{e}$	$\quad$ element of $EVar$
$\quad a[p_1, \ldots, p_n]$	$\quad$ name
$\quad f(p_1, \ldots, p_n)$	$\quad$ 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 them 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 $a[p_1, \ldots, p_n]$. We treat a as a function symbol, which we call 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 session 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 Horn clauses use the following facts:

$F ::=$	facts
$\quad \mathsf{att}(p)$	$\quad$ the attacker knows p
$\quad \mathsf{mess}(p, p')$	$\quad$ the message p' may appear on channel p
$\quad \mathsf{com}(p)$	$\quad$ communication on p
$\quad \mathsf{testunif}(p, p')$	$\quad$ unification test
$\quad \mathsf{bad}$	$\quad$ 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 him 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 closed substitution σ with domain $\mathit{Secr} \cup \mathit{EVar}$ such that $\sigma \mathit{Secr}$ does not contain bound names and $\sigma p = \sigma p'$, and there exists no closed substitution σ' with 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, $\mathsf{testunif}(\mathit{sencrypt}(c_0, k), \mathit{sencrypt}(\mathsf{x}, k))$ is true, because equality of $\mathit{sencrypt}(c_0, k)$ and $\mathit{sencrypt}(\mathsf{x}, k)$ 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 reveal some information on the values of the secrets. We define $\mathit{EVar}(p)$ as the pattern p after substituting distinct new elements of EVar for variables. Consider a destructor g defined by $g(N_1, \ldots, N_n) \to N$. The destructor application $g(M_1, \ldots, M_n)$ succeeds for some, but not all, values of the secrets, if and only if $\mathsf{testunif}((M_1, \ldots, M_n), \mathit{EVar}((N_1, \ldots, N_n)))$ is true. For example, consider the destructor $\mathit{pdecrypt}_{\mathit{prob}}$ defined by the rewrite rule $\mathit{pdecrypt}_{\mathit{prob}}(\mathit{pencrypt}_{\mathit{prob}}(x, \mathit{pk}(y), z), y) \to x$. The corresponding fact $\mathsf{testunif}((\mathsf{x}, \mathit{sk}_B[\,]), (\mathit{pencrypt}_{\mathit{prob}}(\mathsf{e}, \mathit{pk}(\mathsf{e}'), \mathsf{e}''), \mathsf{e}'))$ is true, because the unification succeeds for some values of e, e', and e'', only when x is of the form $\mathit{pencrypt}_{\mathit{prob}}(p_1, \mathit{pk}(\mathit{sk}_B[\,]), p_2)$. That is, the destructor application $\mathit{pdecrypt}_{\mathit{prob}}(\mathsf{x}, \mathit{sk}_B[\,])$ succeeds when x is of the form $\mathit{pencrypt}_{\mathit{prob}}(p_1, \mathit{pk}(\mathit{sk}_B[\,]), p_2)$, 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, we have strong secrecy.

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

4.1 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.

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

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

For each public constructor f of arity n,

$$\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \Rightarrow \mathsf{att}(f(x_1, \ldots, x_n)) \tag{Rf}$$

For each public destructor g,

for each rewrite rule $g(N_1, \ldots, N_n) \to N$ in $\mathrm{def}(g)$,

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

$$\begin{aligned}&\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \wedge \\ &\quad \mathsf{testunif}((x_1, \ldots, x_n), \mathit{EVar}((N_1, \ldots, N_n))) \Rightarrow \mathsf{bad}\end{aligned} \tag{Rt}$$

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

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

$$\mathsf{att}(x) \Rightarrow \mathsf{com}(x) \tag{Rc}$$

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

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 $\mathsf{x} \in \mathit{Secr}$, that is, no communication is done on channels in Secr, which is hypothesis 1 of Proposition 1. When all communications of the protocol take place 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 or not. Hence, if $\mathsf{testunif}((x_1, \ldots, x_n), \mathit{EVar}((M_1, \ldots, M_n)))$ 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 1 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 $a \in \mathit{fn}(P_0)$.

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

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

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 $(\mathit{pencrypt}_{\mathit{prob}}(M, \mathit{pk}(M'), M''), M')$ for some terms M, M', M''.

4.2 Rules for the protocol

The translation $[\![P]\!]\rho s h$ 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 $\rho(f(M_1, \ldots, M_n)) = f(\rho(M_1), \ldots, \rho(M_n))$.

$$[\![0]\!]\rho s h \stackrel{\text{def}}{=} \emptyset$$

$$[\![!P]\!]\rho s h \stackrel{\text{def}}{=} [\![P]\!](\rho[i \mapsto i'])(s, i')(h \wedge \mathsf{att}(i'))$$

$$[\![P \mid Q]\!]\rho s h \stackrel{\text{def}}{=} [\![P]\!]\rho s h \cup [\![Q]\!]\rho s h$$

$$[\![(\nu a)P]\!]\rho s h \stackrel{\text{def}}{=} [\![P]\!](\rho[a \mapsto a[s]])s h$$

$$[\![M(x).P]\!]\rho s h \stackrel{\text{def}}{=} [\![P]\!](\rho[x \mapsto x'])(s, x')(h \wedge \mathsf{mess}(\rho(M), x'))$$
$$\quad \cup \{h \Rightarrow \mathsf{com}(\rho(M))\}$$

$$[\![\overline{M}\langle N\rangle.P]\!]\rho s h \stackrel{\text{def}}{=} [\![P]\!]\rho s h \cup \{h \Rightarrow \mathsf{mess}(\rho(M), \rho(N)), h \Rightarrow \mathsf{com}(\rho(M))\}$$

$$[\![\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho s h \stackrel{\text{def}}{=}$$
$$\quad \cup \{[\![P]\!]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma s, \sigma' p')(\sigma h)$$
$$\qquad \mid g(N_1, \ldots, N_n) \to N \text{ is in } \mathrm{def}(g) \text{ and } (\sigma, \sigma') \text{ is a most general pair of}$$
$$\qquad \text{substitutions such that } \sigma\rho(M_1) = \sigma' N_1, \ldots, \sigma\rho(M_n) = \sigma' N_n\}$$
$$\quad \cup [\![Q]\!]\rho s h$$
$$\quad \cup \{h \wedge \mathsf{testunif}((\rho(M_1), \ldots, \rho(M_n)), \mathit{EVar}((N_1, \ldots, N_n))) \Rightarrow \mathsf{bad}$$
$$\qquad \mid g(N_1, \ldots, N_n) \to N \text{ is in } \mathrm{def}(g)\}$$

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 $M_1, \dots, M_n$ on channels $M'_1, \dots, M'_n$, respectively, we generate a clause $h \Rightarrow \mathsf{mess}(p', p)$ with $h = \mathsf{mess}(p'_1, p_1) \wedge \dots \wedge \mathsf{mess}(p'_n, p_n)$ 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 $p_1, \dots, p_n$ are sent on channels $p'_1, \dots, p'_n$ 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 $g(M''_1, \dots, M''_{n''})$ after receiving the same messages, we generate a clause $h \wedge \mathsf{testunif}((p''_1, \dots, p''_{n''}), \mathit{EVar}((N_1, \dots, N_{n''}))) \Rightarrow \mathsf{bad}$ for each $g(N_1, \dots, N_{n''}) \to N$ 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* $y' = \mathit{pdecrypt}_{\mathit{prob}}(y, sk_B)$ in P_B in the example of Section 2 generates the clause:

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

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

$$\mathsf{mess}(c[\,], \mathit{pencrypt}_{\mathit{prob}}(\mathit{sign}((\mathit{pk}(\mathit{sk}_A[\,]), \mathit{pk}(\mathit{sk}_B[\,]), x), \mathit{sk}_A[\,]), \mathit{pk}(\mathit{sk}_B[\,])))$$
$$\Rightarrow \mathsf{mess}(c[\,], \mathit{sencrypt}(\mathsf{x}, x))$$
$$\mathsf{mess}(c[\,], \mathit{pencrypt}_{\mathit{prob}}(\mathit{sign}((\mathit{pk}(\mathit{sk}_A[\,]), \mathit{pk}(\mathit{sk}_B[\,]), x), \mathit{sk}_A[\,]), \mathit{pk}(\mathit{sk}_B[\,])))$$
$$\Rightarrow \mathsf{com}(c[\,])$$

The first clause means that if a message $\{\{pk_A, pk_B, x\}_{sk_A}\}_{pk_B}$ 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.)

4.3 Proof of strong secrecy

Let $\rho = \{a \mapsto a[\,] \mid a \in \mathit{fn}(P_0) \cup \mathit{Secr}\}$. 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:

$$\mathcal{R}_{P_0} \stackrel{\text{def}}{=} [\![\sigma_0 P_0]\!]\rho\emptyset\emptyset \cup \{(\text{Init}), (\text{Rn}), (\text{Rf}), (\text{Rg}), (\text{Rt}), (\text{Rl}), (\text{Rs}), (\text{Rc}), (\text{Rcom})\}$$

We obtain the following result, proved in Appendix A:

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

The translation into Horn clauses introduces a number of approximations. The Horn clause model ignores the number of repetitions of each event, as already mentioned in [16]. As a consequence of this approximation, the system finds false attacks when a protocol first needs to preserve a secret, and later in the protocol, publishes it. The verifier thinks that the attacker can reintroduce this secret in the beginning of the protocol, which is obviously not true when the secret is a nonce, for instance. We also consider that *else* clauses of destructors may always be executed, even if in fact the destructor succeeds. Outputs are considered as asynchronous. All these approximations may introduce false attacks, but if the system says that strong secrecy is true, then it is true, as shown by Proposition 2. These approximations are key to obtain proofs for an unbounded number of sessions.

4.4 Grouping of Destructors

When several destructors are applied one after the other with the same *else* clause, we generate the Horn clauses as if there were a single "destructor" application. Consider the following process *let* $x = g(M_1, \ldots, M_n)$ *in let* $y = g'(M'_1, \ldots, M'_{n'})$ *in* P *else* Q *else* Q, where g and g' are defined by

$g(N_1, \ldots, N_n) \rightarrow N$ and $g'(N'_1, \ldots, N'_{n'}) \rightarrow N'$. Instead of generating one clause containing testunif for each destructor application, we generate a single one for the whole sequence:

$$h \wedge \mathsf{testunif}((\rho(M_1), \ldots, \rho(M_n), \mathsf{e}, \rho'(M'_1), \ldots, \rho'(M'_{n'})), \\ \mathit{EVar}((N_1, \ldots, N_n, N, N'_1, \ldots, N'_{n'}))) \Rightarrow \mathsf{bad}$$

where $\rho' = \rho[x \mapsto \mathsf{e}]$ and $\mathsf{e} \in \mathit{EVar}$. Indeed, $(\rho(M_1), \ldots, \rho(M_n), \mathsf{e}, \rho'(M'_1), \ldots, \rho'(M'_{n'}))$ and $\mathit{EVar}((N_1, \ldots, N_n, N, N'_1, \ldots, N'_{n'}))$ unify if and only if both destructor applications succeed, and this is what the adversary can observe. This yields a more precise analysis. In the implementation, we have a construct *let pat* $=$ *M in P else Q* in which M can contain several destructors and *pat* is a pattern formed with data constructors and equality tests. All destructor applications executed for the evaluation of such a construct are considered as a single group of destructor applications. (They have the same *else* clause by construction.)

For example, the destructor applications in the process P_B of Section 2 $\mathit{let}\ (= pk_A, = pk_B, k) = \mathit{checksign}(\mathit{pdecrypt}_{\mathit{prob}}(y, sk_B), pk_A)$ generate the clause:

$$\mathsf{mess}(c[\,], y) \wedge \mathsf{testunif}(y, \mathit{pencrypt}_{\mathit{prob}}(\mathit{sign}((pk(sk_A[\,]), pk(sk_B[\,]), \mathsf{e}), sk_A[\,]), \\ pk(sk_B[\,]), \mathsf{e}')) \Rightarrow \mathsf{bad}$$

after the simplifications of testunif facts. Similarly, the process $\mathit{let}\ x = \mathit{sdecrypt}(M, k)\ \mathit{in}\ \mathit{let}\ y = 1th_n(x)\ \mathit{in}\ \mathit{if}\ y = t\ \mathit{then}\ P'$, that is, $\mathit{let}\ (=t, \ldots) = \mathit{sdecrypt}(M, k)\ \mathit{in}\ P'$ (where t is a constant) yields the clause:

$$h \wedge \mathsf{testunif}(\rho(M), \mathit{sencrypt}((t, \mathsf{e}_2, \ldots, \mathsf{e}_n), \rho(k))) \Rightarrow \mathsf{bad}$$

after simplifications. This process is used for the termination result for tagged protocols (see Section 7).

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

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

$\mathsf{sel}(R) \stackrel{\text{def}}{=} \{\mathsf{att}(x_1)\}$. (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

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

In the last case, there may be several choices for F, we simply choose one. The selection function must never select $\mathsf{testunif}$ facts because they are not defined by clauses. It should not select $\mathsf{att}(x)$ because it would lead to non-termination ($\mathsf{att}(x)$ 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*) and the elimination of duplicate hypotheses (performed by *elimdup*), 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 $\mathsf{testunif}$ 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:

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

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 $H \wedge \mathsf{testunif}(p_1, p_2) \Rightarrow C$ 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 $\mathsf{x} \in \mathit{Secr}$, $\sigma_u \mathsf{x}$ 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

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

  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 $\mathsf{testunif}$ 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:

$$\mathsf{att}(x) \wedge \mathsf{att}(y) \wedge \mathsf{att}(z) \wedge \mathsf{testunif}((z, x, \mathsf{e}'', y), (\mathsf{e}, pk(\mathsf{e}'), r[\mathsf{e}], \mathsf{e}')) \Rightarrow \mathsf{bad} \tag{2}$$

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

  Indeed, some instance of the new $\mathsf{testunif}$ 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':

$$\mathsf{att}(x) \wedge \mathsf{att}(y) \wedge \mathsf{att}(z) \wedge \mathsf{testunif}((\mathsf{e}, x, \mathsf{e}'', \mathsf{e}'), (z, pk(y), r[z], y)) \Rightarrow \mathsf{bad} \tag{3}$$

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

  Indeed, e does not occur elsewhere in the $\mathsf{testunif}$ fact (this property comes from the result of *unify* and is preserved by *swap*). Then an instance of the new $\mathsf{testunif}$ 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*:

$$\mathsf{att}(x) \wedge \mathsf{att}(y) \wedge \mathsf{att}(z) \wedge \mathsf{testunif}((x), (pk(y))) \Rightarrow \mathsf{bad} \qquad (4)$$

- Instantiation: *instantiate* transforms clauses of the form

  $$\mathsf{att}(x) \wedge \mathsf{testunif}((\dots, x, \dots), (\dots, f(p_1, \dots, p_n), \dots)) \wedge H \Rightarrow \mathsf{bad}$$

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

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

  1. When $\mathsf{att}(\mathsf{x})$ is derivable for some $\mathsf{x} \in \mathit{Secr}$, 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 $f(p'_1, \dots, p'_n)$. That is why we substitute $f(x_1, \dots, x_n)$ 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, $f(x_1, \dots, x_n)$ and $f(p_1, \dots, p_n)$ will be replaced with $x_1, \dots, x_n$ and $p_1, \dots, p_n$ in the $\mathsf{testunif}$ fact by *unify*. Repeating the simplification does not yield an infinite loop, because the size of the $\mathsf{testunif}$ fact decreases.

  This transformation is important because it replaces a non-selectable fact $\mathsf{att}(x)$ with a selectable one $\mathsf{att}(f(x_1, \dots, x_n))$. This is key for making sure that all clauses with conclusion bad and non-empty hypothesis have a selected hypothesis (see Lemma 1 below).

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

  $$\mathsf{att}(pk(x')) \wedge \mathsf{att}(y) \wedge \mathsf{att}(z) \wedge \mathsf{testunif}((pk(x')), (pk(y))) \Rightarrow \mathsf{bad} \qquad (5)$$

  This clause will then be transformed into

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

  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

  $$\mathsf{att}(x_1) \wedge \mathsf{att}(x_2) \wedge \mathsf{testunif}((\dots, x_1, \dots), (\dots, x_2, \dots)) \wedge H \Rightarrow \mathsf{bad}$$

where x_1 and x_2 appear in matching positions in the tuples in $\mathsf{testunif}$, 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 $\mathsf{testunif}$ fact will be removed by *unify*.

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

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

- Detect $\mathsf{testunif}$ true: *simptestuniftrue* transforms clauses of the form

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

  obtained after the previous simplification steps, where H contains hypotheses of the form $\mathsf{att}(x)$ and for all $i \in \{1, \dots, n\}$, $p_i \in \mathit{Secr}$. 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 $\sigma_1 \mathsf{testunif}((p_1, \dots, p_n), (p'_1, \dots, p'_n))$ is true. This fact can also be written $\mathsf{testunif}((p_1, \dots, p_n), \sigma_1(p'_1, \dots, p'_n))$. 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 $\sigma(p_1, \dots, p_n) = \sigma\sigma_1(p'_1, \dots, p'_n)$. Moreover, there exists no σ' of domain EVar such that $(p_1, \dots, p_n) = \sigma'\sigma_1(p'_1, \dots, p'_n)$, 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 $\mathsf{testunif}$ 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 $\mathit{simplify} = \mathit{elimtaut} \circ \mathit{elimattx} \circ \mathit{elimdup} \circ \mathit{simptestuniftrue} \circ \mathit{elimtestuniffalse} \circ \mathit{repeat}(\mathit{elimvar} \circ \mathit{instantiate} \circ \mathit{elimEVar} \circ \mathit{swap} \circ \mathit{unify})$. 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.

- *condense*($\mathcal{R}$) applies *simplify* to $\mathcal{R}$ and then eliminates subsumed clauses. We say that $H_1 \Rightarrow C_1$ subsumes $H_2 \Rightarrow C_2$, and we write $(H_1 \Rightarrow C_1) \sqsupseteq (H_2 \Rightarrow C_2)$, if and only if there exists a substitution σ such that $\sigma C_1 = C_2$ and $\sigma H_1 \subseteq H_2$. (Hypotheses of clauses are multisets, and the inclusion $\sigma H_1 \subseteq H_2$ is multiset inclusion.) 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 *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 1 *After simplification, if $R = H \Rightarrow \mathsf{bad}$ and H is not empty, then* $\mathsf{sel}(R) \neq \emptyset$.

Proposition 3 *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.*

The detailed proofs of these results can be found in Appendix B.

6 Further Results and Extensions

6.1 Limiting the Set of Values of Secrets

Let $fv(P_0) = \{x_1, \dots, x_n\}$. We now assume that the secrets $x_1, \dots, x_n$ are known to be in sets $S_1, \dots, S_n$ 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 $H \wedge \mathsf{testunif}((p_1, \dots, p_{n'}), (p'_1, \dots, p'_{n'})) \Rightarrow \mathsf{bad}$ 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 $\theta(p_1, \dots, p_{n'})$ and $\theta(p'_1, \dots, p'_{n'})$ 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 the observational equivalence $(\nu a_1) \dots (\nu a_n)\sigma P_0 \approx (\nu a_1) \dots (\nu a_n)\sigma' P_0$, so that $a_1, \dots, a_n$ 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 $a_1, \dots, a_n$: the clause $\mathsf{att}(a_i[\,])$ for $i \in \{1, \dots, n\}$ is removed.

6.2 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 $f(x, g(y)) = f(y, g(x))$ [4]. We encode equations with destructors, as in [15]. For instance, the previous equation can be modeled by having a destructor f defined by $f(x, g(y)) \rightarrow f'(y, g(x))$ and $f(x, y) \rightarrow f'(x, y)$. We can show that this encoding is still sound for strong secrecy. (This technique is still limited to rather simple equations; more complex ones could be handled by unification modulo the equational theory.)

6.3 Preservation by Refinement

Intuitively, a refinement P' of P is an implementation of P (where P is understood as a specification), that is, a process that executes fewer actions than P. For instance, when there is non-determinism in P, P' may execute only one of the possibilities. A formal definition of refinement may be based on observational preorder:

Definition 4 P' is a *refinement* of P if and only if $P' \sqsubseteq P$ in the observational preorder, that is, $\sqsubseteq$ is the largest relation $\mathcal{R}$ such that $P \mathrel{\mathcal{R}} Q$ 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 $P' \mathrel{\mathcal{R}} Q'$;
3. $C[P] \mathrel{\mathcal{R}} C[Q]$ for all closing evaluation contexts C.

The strong secrecy is not preserved by refinement in the sense above. For instance, assuming $\sigma P_0'$ is a refinement of σP_0 for all σ, $\sigma P_0'$ may stop for some values of σ, and execute the same actions as σP_0 otherwise, while σP_0 always executes actions independently of σ. In this case, P_0' reveals more information on the value of σ than P_0. This problem persists even if we assume that the strong secrecy for P_0 has been proved by our method (which implies stricter conditions on P_0 because of our approximations).

However, we can strengthen the definition of refinement, so that if our method proves strong secrecy for P_0, then strong secrecy is still true for all refinements of P_0. Here is the stronger definition of refinement:

Definition 5 $P \Downarrow (\nu \vec{a}) g(M_1, \ldots, M_n)$ if and only if $P\ (\rightarrow \cup \equiv)^*\ C[\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P'\ \mathit{else}\ Q']$ where C is any evaluation context, which binds the names $\vec{a}$.

P' is a *strong refinement* of P if and only if $P' \sqsubseteq P$ where $\sqsubseteq$ is the largest relation $\mathcal{R}$ such that $P \,\mathcal{R}\, Q$ implies

1. if $P \Downarrow (\nu \vec{a}) g(M_1, \ldots, M_n)$ then $Q \Downarrow (\nu \vec{a}) g(M_1, \ldots, M_n)$;
2. if $P \Downarrow m$ then $Q \Downarrow m$;
3. if $P \rightarrow P'$ then there exists Q' such that $Q \rightarrow^* Q'$ and $P' \,\mathcal{R}\, Q'$;
4. $C[P] \,\mathcal{R}\, C[Q]$ for all closing evaluation contexts C.

Intuitively, this notion of refinement considers not only actions immediately visible to the adversary (inputs and outputs), but also destructor applications, which can later become visible since different code is executed depending on their result. One could weaken a bit this definition of refinement by allowing more destructor applications in the refinement, provided that the adversary never learns which of the two branches of code, P' and Q', has been taken.

Proposition 4 *Let σ_0 be a substitution mapping all free variables of P_0 to distinct elements of Secr. If $\sigma_0 P_0'$ is a strong refinement of $\sigma_0 P_0$ and our system proves that P_0 preserves the strong secrecy of its free variables, then P_0' preserves the strong secrecy of its free variables.*

Proof P_0 satisfies the hypothesis of Proposition 1. By definition of strong refinement, so does P_0', so by Proposition 1, P_0' preserves the strong secrecy of its free variables. $\square$

We would have similar results with testing preorder and testing equivalence instead of observational preorder and observational equivalence.

6.4 Decomposition of Data Constructors

Our algorithm uses one further optimization, which we did not mention previously for simplicity and brevity, namely the decomposition of data constructors. We distinguish two categories of constructors and destructors. *Data* constructors f (such as tuples) come with *data* destructors, that is projections g_i defined by $g_i(f(x_1, \ldots, x_n)) \rightarrow x_i$ for each $i = 1, \ldots, n$. Other constructors and destructors are *cryptographic* constructors and destructors. Rules (Rf) for data constructors are collected in the set $\mathcal{R}_{\mathrm{DataConstr}}$ while rules (Rg) for data destructors are collected in $\mathcal{R}_{\mathrm{DataDestr}}$.

The decomposition of data constructors, *decomp*, transforms clauses as follows. If f is a data constructor, *decomp* replaces recursively all facts $\mathsf{att}(f(p_1, \ldots, p_n))$ with $\mathsf{att}(p_1) \wedge \ldots \wedge \mathsf{att}(p_n)$. When such a fact is in the conclusion of a clause, n clauses are created, with the same hypotheses and the conclusions $\mathsf{att}(p_1), \ldots, \mathsf{att}(p_n)$ respectively. (The soundness of this operation follows from the equivalence between $\mathsf{att}(f(p_1, \ldots, p_n))$ and $\mathsf{att}(p_1) \wedge \ldots \wedge \mathsf{att}(p_n)$ in the presence of the clauses $\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \Rightarrow \mathsf{att}(f(x_1, \ldots, x_n))$ and $\mathsf{att}(f(x_1, \ldots, x_n)) \Rightarrow \mathsf{att}(x_i)$ in $\mathcal{R}_{\mathrm{DataConstr}}$ and in $\mathcal{R}_{\mathrm{DataDestr}}$.) With decomposition, the standard clauses for data constructors and projections can be removed. (We have proved the soundness of this removal.)

This transformation makes it possible to obtain the termination result on tagged protocols (see Section 7) without having to tag the applications of data constructors and destructors.

7 Termination

We have shown the termination of the algorithm for *tagged protocols*, extending the result shown in [17] for standard secrecy. 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. Tagging retains the intended behavior of the initial protocol, and is also a good protocol design: it guarantees the absence of type flaw attacks [32], since participants can use tags to identify messages unambiguously.

This result includes two important novelties. We define tagged protocols as a class of processes, and show that the clauses generated from these processes satisfy required invariants, while [17] stated the invariants on the clauses, with only informal relations with protocols. We also extend the termination proof to handle the new clauses generated for strong secrecy.

Formally, we can define tagged protocols as follows:

Definition 6 (Tagged protocol) A tagged protocol is a process P_0 together with a signature of constructors and destructors such that:

C1. The only constructors and destructors are those of Figures 2, plus *eq*.

C2. In all occurrences of $M(x)$ and $\overline{M}\langle N\rangle$ in P_0, M is a constant name c, and $c \in fn(P_0)$.

C3. In all occurrences of the term $f(\ldots)$ with $f \in \{sencrypt, sencrypt_{prob}, pencrypt_{prob}, sign, nmrsign, H, mac\}$ in P_0, the first argument of f is a tuple $(t, M_1, \ldots, M_n)$. Different occurrences of f have different values of the tag t.

C4. In all occurrences of $let\ x = g(\ldots)\ in\ P\ else\ Q$, for $g \in \{sdecrypt, sdecrypt_{prob}, pdecrypt_{prob}, checksign, getmess\}$ in P_0, the process P is of the form $let\ y = 1th_n(x)\ in\ if\ y = t\ then\ P'$.

In all occurrences of the destructor *nmrchecksign* in P_0, its third argument is of the form $(t, M_1, \ldots, M_n)$.

C5. Destructor applications (including equality tests) have no *else* clauses. There exists an execution of P_0 (without adversary) in which all program points are executed exactly once, and all destructor applications succeed.

C6. The second argument of *pencrypt* in the execution of Condition C5 is of the form $pk(M)$ for some M.

C7. The arguments of *pk* and *host*, the second arguments of *sign* and *nmrsign* in the execution of Condition C5 are atomic constants (names created by restrictions not under inputs, destructor applications, or replications).

The first condition limits the allowed cryptographic primitives. Such a condition is necessary, because in full generality destructors can generate any clause with predicate att, which leads obviously to undecidability. The termination result would still hold if we had probabilistic signatures and/or deterministic public-key encryption. (The conditions on destructors can also be expressed by constraints on the clauses that they generate, but these constraints are complex, so it is easier to give explicitly a list of supported destructors.) Condition C2 states that all messages are sent and received on public channels. This condition is true for many protocols, even if it strongly restricts the internal computation abilities of the participants. Condition C3 requires that tags are always added when building messages, and that different messages have different tags, while Condition C4 requires that tags are always checked. Note that the sequence of tests of Condition C4 is considered as a single group of destructor applications, so generates a single clause containing testunif.

In most cases, the execution of Condition C5 is simply the intended execution of the protocol. All terms created during the execution of Condition C5 have pairwise distinct tags (since each program point is executed at most once, and tags at different program points are different by Condition C3). We can prove that it also guarantees that the terms of all clauses generated for the process P_0 have instances in the set of terms created during the execution of Condition C5 (using the fact that all program points are executed at least once). These properties are keys in the termination proof. More concretely, Condition C5 means that after removing replications of P_0, the resulting process has an execution which executes each program point (at least) once. In this execution, all destructor applications succeed and the process reduces to an inactive process $(\nu a_1) \dots (\nu a_n)0$. Since, after removing replications, the number of executions of a process is always finite, condition C5 is decidable.

Conditions C6 and C7 are technical. They are satisfied by most protocols. Condition C6 means that, in its intended execution, the protocol always applies public-key encryption with a public key, and Condition C7 that the long-term secret (symmetric and asymmetric) keys are atomic constants.

The presence of new clauses and the special simplification steps of the $\mathsf{testunif}$ predicate complicate the termination proof. However, we can prove termination as shown by the following result.

Proposition 5 *The algorithm terminates on tagged protocols.*

More details on the termination proof can be found in Appendix C. A key new idea of the proof is to consider a set of clauses transformed as follows from the clauses on which the algorithm works: the elements of $Secr \cup EVar$ are replaced with variables, and clauses $H \wedge \mathsf{testunif}(p_1, p_2) \Rightarrow \mathsf{bad}$ are replaced with $\sigma_u H \Rightarrow \mathsf{bad}$, where σ_u is the most general unifier of p_1 and p_2. (When the clause is kept by *unify*, p_1 and p_2 are unifiable after replacing elements of $Secr \cup EVar$ with variables.)

Let $\mathcal{R}'_{\mathrm{Prot}}$ be the set of protocol clauses transformed as explained above. We first show the following properties:

P1. Let S_0 be the set of patterns corresponding to messages manipulated in the "intended execution", that is, the execution of Condition C5. For each R in $\mathcal{R}'_{\mathrm{Prot}}$, there exists a substitution σ mapping the arguments of predicates of R to subterms of S_0.

P2. The arguments of predicates in clauses of $\mathcal{R}'_{\mathrm{Prot}}$ are *fully tagged*, that is essentially, all occurrences *sencrypt*, $sencrypt_{prob}$, $pencrypt_{prob}$, *sign*, *nmrsign*, H, *mac* have a first argument of the form $(t, \dots)$, where t is a constant.

The proof of P1 and P2 is rather complex, and involves inductions both on the process and on the execution trace of Condition C5. After the transformation above, the clauses for the attacker satisfy similar properties in which P2 is replaced with a slightly relaxed property P2'. In the second part of the proof, we show that all clauses generated by resolution also satisfy P1 and P2', using in particular the fact that the transformed clause of the resolution of R and R' is equal to the resolution of the transformed clauses of R and R'. We also show that the generated clauses strictly decrease in a suitable well-founded ordering, which easily yields the termination of the algorithm. The second part of the proof is an extension of the proof of [17], using similar techniques.

The following two propositions are easy to prove, and extend the scope of our termination result:

Proposition 6 *Let P_0' be the process obtained from P_0 by replacing let $x = g(M_1, \ldots, M_n)$ in P else Q with let $x = g(M_1, \ldots, M_n)$ in $P \mid Q$. If the algorithm terminates on P_0' then it terminates on P_0.*

Proof The clauses are exactly the same for P_0 and P_0'. □

This proposition enables us to weaken Condition C5, such that destructor applications can have non-empty *else* clauses. This can be useful for error handling: when receiving an erroneous message, a destructor fails; the participants of the protocol can then respond with an error message.

Proposition 7 *If $P_0 \mid Q$ is a tagged protocol, then the algorithm terminates on P_0.*

Proof The process P_0 creates less clauses than $P_0 \mid Q$, and, considering an algorithm that does not use the elimination of subsumed clauses, the resolution then also generates less clauses. If $P_0 \mid Q$ is a tagged protocol, this modified algorithm terminates for $P_0 \mid Q$ (the proof of Proposition 5 is also valid for an algorithm without elimination of subsumed clauses), so a fortiori for P_0. Then the standard algorithm also terminates for P_0. □

The last proposition makes it possible, for instance, to accommodate outputs done at the beginning of the execution of P_0, and intended to give the initial knowledge of the adversary. Such outputs have no corresponding inputs, so Condition C5 could fail because of them. We can simply add in parallel to P_0 a process Q that receives these messages. Then we can prove termination. The process Q can also be used in conjunction with Proposition 6 to receive messages sent in *else* clauses of destructors.

The tagging conditions are satisfied by many protocols after adding tags. In practice, the algorithm terminates on many examples without tags. (This is partly explained by the termination on the larger class of "weakly tagged logic programs", as mentioned in [17].)

8 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 [41], Yahalom [21], and Skeme [33]. (We comment on the treatment of Diffie-Hellman key agreements in Section 6.2.) 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 [10]. 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 connections only with honest principals). Accordingly, we show the observational equivalence of configurations in which two initiators I_1 and I_2 can 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 6.1, 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 the public keys of the principals ($pk(sk_1)$, $pk(sk_2)$, $\ldots$), 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 can 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 $sk_1, \dots, sk_4$, 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.

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

9 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.

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 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.

[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] 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.

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

[9] 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.

[10] 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.

[11] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 2, pages 19–100. North Holland, 2001.

[12] M. Backes and B. Pfitzmann. Computational probabilistic noninterference. 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] J. Heather, G. Lowe, and S. Schneider. How to Prevent Type Flaw Attacks on Security Protocols. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 255–268, July 2000.

[33] 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`.

[34] 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.

[35] P. Laud. Symmetric Encryption in Automatic Analyses for Confidentiality against Active Adversaries. In *IEEE Symposium on Security and Privacy*, pages 71–85, Oakland, California, May 2004.

[36] 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.

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

[38] C. Lynch. Oriented equational logic programming is complete. *Journal of Symbolic Computation*, 21(1):23–45, 1997.

[39] 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.

[40] 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.

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

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

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

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

[45] 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.

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

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

Appendix: Proofs

A Correctness of the Generation of Clauses

Proof (of Proposition 1) We first reformulate the hypotheses 1 and 2 using contexts. 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 $M \notin \mathit{Secr}'$.

2. If $C[\sigma'_0 P_0] \ (\rightarrow \cup \equiv)^* \ C'[\mathit{let}\ y = g(M_1, \dots, M_n)\ \mathit{in}\ Q'\ \mathit{else}\ R']$, θ is a closed substitution with $\mathit{dom}(\theta) = \mathit{Secr}'$, C' is an evaluation context that does not bind names in Secr' or in the image of θ, $g(N_1, \dots, N_n) \rightarrow N$ is in $\mathrm{def}(g)$, and $\theta(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$, then $(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$.

Let us justify it for the first case of hypothesis 1. The second case is very similar. Let $P_1 = \overline{M}\langle N\rangle.R$. We have $C[\sigma'_0 P_0] \ (\rightarrow \cup \equiv)^* \ C'[P_1]$. There exist a set of names $\vec{a}$ and a process Q such that for all P, $C[P] \equiv (\nu\vec{a})(P \mid Q)$. Moreover, we can choose Q such that $\mathit{fn}(Q) \cap \mathit{Secr}' = \emptyset$, since Secr' does not contain free or bound names of C. Hence $(\nu\vec{a})(\sigma'_0 P_0 \mid Q) \equiv C[\sigma'_0 P_0] \ (\rightarrow \cup \equiv)^* \ C'[P_1]$. Then there exists an evaluation context C'' such that $\sigma'_0 P_0 \mid Q \ (\rightarrow \cup \equiv)^* \ C''[P_1]$ with $C'[P_1] \equiv (\nu\vec{a})C''[P_1]$. (We use the following standard result: if $(\nu a)P \ (\rightarrow \cup \equiv)^* \ P'$, then there exists P'' such that $P \ (\rightarrow \cup \equiv)^* \ P''$ and $P' \equiv (\nu a)P''$. We also use the fact that P_1 does not contain restrictions at its top.) Let α' be a mapping from $\mathit{fn}(Q) \cap \mathit{Secr}$ to distinct names not in Secr, Secr', $\mathit{fn}(P_0)$, $\mathit{fn}(C)$, or the bound names of C. The mapping α' does not modify $\sigma'_0 P_0$, since $\mathit{fn}(Q) \cap \mathit{Secr} \cap \mathit{fn}(\sigma'_0 P_0) = \emptyset$, because $\mathit{Secr} \cap \mathit{fn}(\sigma'_0 P_0) \subseteq \mathit{Secr}'$. (Indeed, $\mathit{fn}(\sigma'_0 P_0) \subseteq \mathit{fn}(P_0) \cup \mathit{Im}(\sigma'_0)$, so $\mathit{fn}(\sigma'_0 P_0) \cup \mathit{Secr} \subseteq (\mathit{fn}(P_0) \cap \mathit{Secr}) \cup (\mathit{Im}(\sigma'_0) \cap \mathit{Secr}) \subseteq \emptyset \cup \mathit{Secr}'$.) So $\sigma'_0 P_0 \mid \alpha' Q \ (\rightarrow \cup \equiv)^* \ \alpha'(C''[P_1])$. Let α be the bijection from Secr' to Secr such that $\alpha\sigma'_0 = \sigma_0$. Thus $\sigma_0 P_0 \mid \alpha\alpha' Q \ (\rightarrow \cup \equiv)^* \ \alpha\alpha'(C''[P_1])$. Moreover, $\mathit{fn}(\alpha\alpha' Q) \cap \mathit{Secr} = \emptyset$, since $\mathit{fn}(\alpha' Q) \cap (\mathit{Secr} \cup \mathit{Secr}') = \emptyset$. Indeed, $\mathit{fn}(Q) \cap \mathit{Secr}' = \emptyset$, and $\mathit{fn}(\alpha' Q) \subseteq \mathit{fn}(Q) - \mathit{dom}(\alpha') \cup \mathit{Im}(\alpha')$, so $\mathit{fn}(\alpha' Q) \cap (\mathit{Secr} \cup \mathit{Secr}') \subseteq [\mathit{fn}(Q) \cap (\mathit{Secr} \cup \mathit{Secr}')] - [\mathit{fn}(Q) \cap \mathit{Secr}] \cup \emptyset \subseteq \mathit{fn}(Q) \cap \mathit{Secr}' = \emptyset$. Let α'' be a renaming of bound names of C'' to names not in $\mathit{Secr} \cup \mathit{Secr}'$. After applying the renaming α'', we can then apply hypothesis 1, which yields $\alpha\alpha'\alpha'' M \notin \mathit{Secr}$. In order to obtain a contradiction, assume that $M \in \mathit{Secr}'$. We have $\alpha'' M = M$ since C'' does not bound names in Secr', so names in Secr' are untouched by α''. We also have $\alpha' M = M$ since $\mathit{dom}(\alpha') \cap \mathit{Secr}' = \mathit{fn}(Q) \cap \mathit{Secr} \cap \mathit{Secr}' = \emptyset$. So $\alpha\alpha'\alpha'' M = \alpha M \in \mathit{Secr}$, contradiction. Then $M \notin \mathit{Secr}'$, which is the expected reformulation of hypothesis 1.

For hypothesis 2, we obtain as above $\sigma_0 P_0 \mid \alpha\alpha' Q \ (\rightarrow \cup \equiv)^* \ \alpha\alpha'(C''[P_1])$, with $P_1 = \mathit{let}\ y = g(M_1, \dots, M_n)\ \mathit{in}\ Q'\ \mathit{else}\ R'$. (We add to the previous definitions that the image of α' does not contain names in the image of θ.) Let α'' be a renaming of bound names of C'' to names not in $\mathit{Secr} \cup \mathit{Secr}'$ and not in the image of θ. After applying the renaming α'', we can then apply hypothesis 2 to $(\theta\alpha^{-1})_{|\mathit{Secr}}$: if $(\theta\alpha^{-1})_{|\mathit{Secr}}\alpha\alpha'\alpha''(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$ then $\alpha\alpha'\alpha''(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$, so $(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$ (since α, α', and α'' are renamings, and $(N_1, \dots, N_n)$ do not contain names). Moreover, $(\theta\alpha^{-1})_{|Secr}\alpha\alpha'\alpha''(M_1, \dots, M_n) = \theta\alpha'\alpha''(M_1, \dots, M_n)$, since the term $\alpha'\alpha''(M_1, \dots, M_n)$ does not contain names in $Secr'$. The bijections α' and α'' commute with θ since $dom(\alpha') \cap dom(\theta') = dom(\alpha') \cap Im(\theta') = Im(\alpha') \cap dom(\theta') = \emptyset$, and similarly for α''. So $(\theta\alpha^{-1})_{|Secr}\alpha\alpha'\alpha''(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$ if and only if $\alpha'\alpha''\theta(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$, that is, $\theta(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$ (since α' and α'' are renamings, and $(N_1, \dots, N_n)$ do not contain names). Therefore, we obtain the expected reformulation of hypothesis 2.

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 $Secr'$, 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 conditions of the definition of observational equivalence. 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, we obtain that P_0 preserves the strong secrecy of its free variables.

1. Let us show that, if $P \mathcal{R} P'$ and $P \equiv P_1$, then there exists P_1' such that $P' \equiv P_1'$ and $P_1 \mathcal{R} P_1'$.

   We have $P = \theta P'' \equiv P_1$, then there exists P_1'' such that $P_1 = \theta P_1''$ and $P'' \equiv P_1''$. This is true because $\equiv$ does not depend on terms. We have $P' = \theta' P'' \equiv \theta' P_1''$. Let $P_1' = \theta' P_1''$. Then we have $P' \equiv P_1'$. Moreover, $P_1 = \theta P_1''$, $P_1' = \theta' P_1''$, and $C[\sigma_0' P_0] \ (\rightarrow \cup \equiv)^* \ P'' \equiv P_1''$, so $P_1 \mathcal{R} P_1'$.

2. Let us show the second condition of the definition of observational equivalence: If $P \mathcal{R} P'$ and $P \rightarrow P_1$, then there exists P_1' such that $P' \rightarrow^* P_1'$ and $P_1 \mathcal{R} P_1'$.

   We modify applications of (Red $\equiv$) in the derivation of $P \rightarrow P_1$ such that $P \equiv P_2 \rightarrow P_3 \equiv P_1$ where $P_2 \rightarrow P_3$ does not use (Red $\equiv$). This can be done by moving (Red $\equiv$) before (Red Par) and (Red Res). Then, using the point above, it is enough to prove the result when $P \rightarrow P_1$ without using (Red $\equiv$).

   In this case, we have $\theta P'' \rightarrow P_1$ (without (Red $\equiv$)). We show that there exists P_1'' such that $P_1 = \theta P_1''$, $P'' \rightarrow P_1''$ (without (Red $\equiv$)), and $\theta' P'' \rightarrow \theta' P_1''$ (without (Red $\equiv$)). We proceed by cases on the derivation of $\theta P'' \rightarrow P_1$.

   - Case (Red I/O) (perhaps after (Red Par) or (Red Res)): $\theta P'' = C'[a(y).Q_1' \mid \overline{a}\langle M_1' \rangle.R_1']$, so P'' is of the form $P'' = C''[N_1(y).Q_1 \mid$

$\overline{N'_1}\langle M_1\rangle.R_1]$. Moreover, by renaming of the bound names of C'', we can choose C'' such that it does not bind elements of $Secr'$ or names in the image of θ, so that $\theta P'' = (\theta C'')[\theta N_1(y).\theta Q_1 \mid \overline{\theta N'_1}\langle \theta M_1\rangle.\theta R_1]$ and $\theta N_1 = \theta N'_1 = a$, $P_1 = \theta(C''[Q_1\{M_1/y\} \mid R_1])$. We have $C[\sigma'_0 P_0]\ (\rightarrow \cup \equiv)^*\ C''[N_1(y).Q_1 \mid \overline{N'_1}\langle M_1\rangle.R_1]$. By hypothesis 1, $N_1 \notin Secr'$ and $N'_1 \notin Secr'$. Hence $N_1 = \theta N_1 = a$ and $N'_1 = \theta N'_1 = a$. Then $P'' = C''[a(y).Q_1 \mid \overline{a}\langle M_1\rangle.R_1]$. Hence $P'' \rightarrow C''[Q_1\{M_1/y\} \mid R_1]$, and $\theta' P'' \rightarrow \theta' C''[Q_1\{M_1/y\} \mid R_1]$ (since a is not modified by θ'). So $P''_1 = C''[Q_1\{M_1/y\} \mid R_1]$ satisfies the desired properties.

- Case (Red Destr 1) (perhaps after (Red Par) or (Red Res)): As in the case (Red I/O), $P'' = C''[\mathit{let}\ y = g(M_1, \ldots, M_n)\ \mathit{in}\ Q'\ \mathit{else}\ R']$, C'' does not bind elements of $Secr'$ or names in the image of θ, $g(N_1, \ldots, N_n) \rightarrow N$ is in def(g), and there exists σ such that for all i, $\theta M_i = \sigma N_i$. We have $C[\sigma'_0 P_0]\ (\rightarrow \cup \equiv)^*\ C''[\mathit{let}\ y = g(M_1, \ldots, M_n)\ \mathit{in}\ Q'\ \mathit{else}\ R']$. By hypothesis 2, there exists σ' such that for all i, $M_i = \sigma' N_i$. Then $P_1 = \theta C''[Q'\{\sigma' N/y\}]$ and $P'' \rightarrow C''[Q'\{\sigma' N/y\}]$. Then $\theta' P'' \rightarrow \theta' C''[Q'\{\sigma' N/y\}]$, since $(\theta' M_1, \ldots, \theta' M_n) = (\theta'\sigma' N_1, \ldots, \theta'\sigma' N_n)$. So the process $P''_1 = C''[Q'\{\sigma' N/y\}]$ satisfies the desired properties.
- Case (Red Destr 2) (perhaps after (Red Par) or (Red Res)): As in the case (Red I/O), $P'' = C''[\mathit{let}\ y = g(M_1, \ldots, M_n)\ \mathit{in}\ Q'\ \mathit{else}\ R']$, C'' does not bind elements of $Secr'$ or names in the image of θ, there exists no $g(N_1, \ldots, N_n) \rightarrow N$ in def(g), and σ such that for all i, $\theta M_i = \sigma N_i$, and $P_1 = \theta C''[R']$. There exists no $g(N_1, \ldots, N_n) \rightarrow N$ in def(g), and σ' such that for all i, $M_i = \sigma' N_i$ (otherwise, there would exist one for θ), so $P'' \rightarrow C''[R']$. We have $C[\sigma'_0 P_0]\ (\rightarrow \cup \equiv)^*\ C''[\mathit{let}\ y = g(M_1, \ldots, M_n)\ \mathit{in}\ Q'\ \mathit{else}\ R']$. By hypothesis 2, if there existed σ such that for all i, $\theta' M_i = \sigma N_i$, then there would exist σ' such that for all i, $M_i = \sigma' N_i$, which is a contradiction. So there exists no σ such that $\theta' M_i = \sigma N_i$. Then $\theta' P'' = \theta' C''[\mathit{let}\ y = g(M_1, \ldots, M_n)\ \mathit{in}\ Q'\ \mathit{else}\ R'] \rightarrow \theta' C''[R']$. So $P''_1 = C''[R']$ satisfies the desired properties.

Taking $P'_1 = \theta' P''_1$, we have $P' \rightarrow P'_1$. Moreover $P_1 = \theta P''_1$, $P'_1 = \theta' P''_1$, and $C[\sigma'_0 P_0]\ (\rightarrow \cup \equiv)^*\ P'' \rightarrow P''_1$, so $P_1\, \mathcal{R}\, P'_1$.

3. Let us show the first condition of the definition of observational equivalence: If $P\, \mathcal{R}\, P'$ and $P \Downarrow m$, then $P' \Downarrow m$.

   Using the two points above, it is sufficient to show that, if $P = C_1[\overline{m}\langle N_1\rangle.R_1]$ where C_1 does not bind m, then $P' = C_2[\overline{m}\langle N_2\rangle.R_2]$ where C_2 does not bind m. We have $\theta P'' = C_1[\overline{m}\langle N_1\rangle.R_1]$, so $P'' = C_3[\overline{m'}\langle N_3\rangle.R_3]$. Therefore $C[\sigma'_0 P_0]\ (\rightarrow \cup \equiv)^*\ C_3[\overline{m'}\langle N_3\rangle.R_3]$. By hypothesis 1, $m' \notin \mathit{Secr}'$. Moreover, $\theta m'$ is a name, so $m' = \theta m' = m = \theta' m'$, hence $P' = \theta' P'' = C_2[\overline{m}\langle N_2 \rangle.R_2]$, therefore $P' \Downarrow m$.

4. Finally, let us show the third condition of the definition of observational equivalence: If $P \mathrel{\mathcal{R}} P'$, then $C'[P] \mathrel{\mathcal{R}} C'[P']$ for all closing evaluation contexts C'.

   Let Secr'' be a set of names not free or bound in C or C', and not free in P_0, containing as many elements as Secr'. Let α be a bijection from Secr' to Secr''. We have $C'[P] = C'[\theta\alpha^{-1}\alpha P''] = \theta\alpha^{-1}C'[\alpha P'']$. Similarly, $C'[P'] = \theta'\alpha^{-1}C'[\alpha P'']$. Moreover $C'[C[\alpha\sigma'_0 P_0]] = C'[\alpha C[\sigma'_0 P_0]]$ $(\rightarrow \cup \equiv)^* \; C'[\alpha P'']$. So we have $C'[P] \mathrel{\mathcal{R}} C'[P']$ by taking $C'[C]$ instead of C, Secr'' instead of Secr', $\alpha\sigma'_0$ instead of σ'_0, $\theta\alpha^{-1}$ instead of θ, $\theta'\alpha^{-1}$ instead of θ', and $C'[\alpha P'']$ instead of P''.

□

Proof (of Proposition 2) We exploit the theory developed in [3, 16] to prove the hypotheses of Proposition 1. We introduce *events* $\mathsf{end}(M)$ that can be used as witnesses that some part of a process has been executed. Such events have already been used for the specification of authenticity [16]. From the process P_0 and any adversary Q such that $\mathit{fn}(Q) \cap \mathit{Secr} = \emptyset$, 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 $\mathsf{end}(\mathsf{test}_g(M_1, \ldots, M_n))$.
- 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 $\mathsf{end}(\mathsf{com}(M))$.
- Q' is an adversary in the sense of [16], that is, it does not contain end events and $\mathit{fn}(Q') \cap \mathit{Secr} = \emptyset$.

More precisely, we transform the process $\sigma_0 P_0$ by adding the following events: Before each destructor application *let* $x = g(M_1, \ldots, M_n)$ *in* $\ldots$, we add the event $\mathsf{end}(\mathsf{test}_g(M_1, \ldots, M_n))$. Before each input $M(y)$ and output $\overline{M}\langle N \rangle$, we add the event $\mathsf{end}(\mathsf{com}(M))$. We add in parallel with this process the process:

$$\begin{aligned} R = {} & !c_{\mathsf{com}}(x).\mathsf{end}(\mathsf{com}(x)) \\ & \mid \prod_{\text{destructor } g \text{ of arity } n} !c_g((x_1, \ldots, x_n)).\mathsf{end}(\mathsf{test}_g(x_1, \ldots, x_n)) \end{aligned}$$

where c_{com} and c_g are names that do not occur elsewhere. Let P'_0 be the obtained process.

From the process Q such that $\mathit{fn}(Q) \cap \mathit{Secr} = \emptyset$ and Q does not contain end events, we build the process Q' such that $\mathit{fn}(Q') \cap \mathit{Secr} = \emptyset$ and Q' does not contain end events. Q' is obtained from Q by adding $\overline{c_g}\langle(M_1, \dots, M_n)\rangle$ before each destructor application *let* $y = g(M_1, \dots, M_n)$ *in* ... in Q and $\overline{c_{\mathsf{com}}}\langle M\rangle$ before each input $M(y)$ and output $\overline{M}\langle N\rangle$ in Q. Since Q' cannot execute end events directly, it sends a message on c_{com} or c_g instead of executing the end event, and the process R included in P'_0 executes the required end event when receiving this 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 $\mathsf{end}(M)$ then a fact $\mathsf{end}(p)$ is derivable from the clauses, where p is the pattern corresponding to the term M after encoding names by functions. More precisely, we can show that, if for some adversary Q', $P'_0 \mid Q' \ (\rightarrow \cup \equiv)^* \ C'[\mathsf{end}(M)]$, then there exists a function E from terms to patterns such that

E1. E substitutes names with patterns $a[\dots]$; names not bound in C' are substituted with patterns $a[\,]$ where a does not correspond to a restriction in P'_0;

E2. E is an injection;

E3. if $E(M')$ is an instance of a term N not containing names, then M' is an instance of N;

E4. $\mathsf{end}(p)$ is derivable from $\mathcal{R}'$ where $p = E(M)$.

In fact, E simply maps each term to its corresponding pattern. The last property of E is the soundness of the Horn clause representation: if some end event may happen, then the corresponding end fact is derivable from the clauses. (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 clause $h \Rightarrow \mathsf{end}(\mathsf{test}_g(p_1, \dots, p_n))$ in $\mathcal{R}'$, for each rewrite rule $g(N_1, \dots, N_n) \rightarrow N$ in $\mathrm{def}(g)$, the clause $h \wedge \mathsf{testunif}((p_1, \dots, p_n), \mathit{EVar}((N_1, \dots, N_n))) \Rightarrow \mathsf{bad}$ is in $\mathcal{R}_{P_0}$.
- For each clause $h \Rightarrow \mathsf{end}(\mathsf{com}(p))$ in $\mathcal{R}'$, the clause $h \Rightarrow \mathsf{com}(p)$ is in $\mathcal{R}_{P_0}$.
- The clauses that define the predicates of h (att and mess) are the same in $\mathcal{R}_{P_0}$ and $\mathcal{R}'$.

Let us prove the second hypothesis of Proposition 1. Assume that $\sigma_0 P_0 \mid Q \ (\rightarrow \cup \equiv)^* \ C[\mathit{let}\ y = g(M_1, \dots, M_n)\ \mathit{in}\ \dots]$, θ is a closed substitution with $\mathit{dom}(\theta) = \mathit{Secr}$, C is an evaluation context that does not bind names in Secr or in the image of θ, $g(N_1, \dots, N_n) \rightarrow N$ is in $\mathrm{def}(g)$, and $\theta(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$.

Since $\sigma_0 P_0 \mid Q\ (\rightarrow \cup \equiv)^* C[let\ y = g(M_1, \dots, M_n)\ in\ \dots]$, we have $P'_0 \mid Q'\ (\rightarrow \cup \equiv)^* C'[\mathsf{end}(\mathsf{test}_g(M_1, \dots, M_n))]$ for some evaluation context C' that does not bind names in *Secr* or in the image of θ. (Either the end event is executed just before the destructor application, or an output on c_g is executed just before the destructor application, and this output enables the execution of $c_g((x_1, \dots, x_n)).\mathsf{end}(\mathsf{test}_g(x_1, \dots, x_n))$.) Hence, by Property E4, $\mathsf{end}(\mathsf{test}_g(p_1, \dots, p_n))$ is derivable from $\mathcal{R}'$ where $p_i = E(M_i)$ for $i = 1, \dots, n$.

Let σ such that for all i, $\theta M_i = \sigma N_i$. Then $E(\theta M_i) = E(\sigma N_i)$, that is, $E(\theta) p_i = E(\sigma) N_i$ since $p_i = E(M_i)$ and N_i does not contain names. By Property E1, since the image of θ does not contain names bound in C', the image of $E(\theta)$ does not contain bound names. By construction of $\mathcal{R}_{P_0}$ and $\mathcal{R}'$, if $\mathsf{end}(\mathsf{test}_g(p_1, \dots, p_n))$ is derivable from $\mathcal{R}'$ and there exists $g(N_1, \dots, N_n) \rightarrow N$ in $\mathrm{def}(g)$ such that $\mathsf{testunif}((p_1, \dots, p_n), EVar((N_1, \dots, N_n)))$ is true, then bad is derivable from $\mathcal{R}_{P_0}$. Since bad is not derivable from $\mathcal{R}_{P_0}$ and $\mathsf{end}(\mathsf{test}_g(p_1, \dots, p_n))$ is derivable from $\mathcal{R}_{P_0}$, for all $g(N_1, \dots, N_n) \rightarrow N$ in $\mathrm{def}(g)$, $\mathsf{testunif}((p_1, \dots, p_n), EVar((N_1, \dots, N_n)))$ is false. Moreover, let σ'' be the substitution that maps $\mathsf{x} \in Secr$ to $E(\theta)\mathsf{x}$ and $\mathsf{e} \in EVar$ to $E(\sigma)x$ where x is the variable such that applying $EVar$ to $(N_1, \dots, N_n)$ maps x to e. Then σ'' is a closed substitution of domain $Secr \cup EVar$ such that $\sigma'' Secr$ does not contain bound names and $\sigma''(p_1, \dots, p_n) = \sigma'' EVar((N_1, \dots, N_n))$. Hence, there exists σ' of domain $EVar$ such that $\sigma'(p_1, \dots, p_n) = \sigma' EVar((N_1, \dots, N_n))$. Since the elements of $EVar$ do not occur in $(p_1, \dots, p_n)$, $(p_1, \dots, p_n)$ is an instance of $(N_1, \dots, N_n)$. So $E((M_1, \dots, M_n))$ is an instance of $(N_1, \dots, N_n)$. So, by Property E3, $(M_1, \dots, M_n)$ is an instance of $(N_1, \dots, N_n)$.

We now prove the first hypothesis of Proposition 1. Similarly to the proof of the second hypothesis, if $\sigma_0 P_0 \mid Q\ (\rightarrow \cup \equiv)^* C[N_1(y).Q_1]$ or $\sigma_0 P_0 \mid Q\ (\rightarrow \cup \equiv)^* C[\overline{N_1}\langle M_1 \rangle.R_1]$, and C is an evaluation context that does not bind names in *Secr*, then $P'_0 \mid Q'\ (\rightarrow \cup \equiv)^* C'[\mathsf{end}(\mathsf{com}(N_1))]$ for some evaluation context C' that does not bind names in *Secr*. Then, by Property E4, $\mathsf{end}(\mathsf{com}(p_1))$ is derivable from $\mathcal{R}'$, where $p_1 = E(N_1)$. So $\mathsf{com}(p_1)$ is derivable from $\mathcal{R}_{P_0}$, by construction of $\mathcal{R}'$ and $\mathcal{R}_{P_0}$. Since bad is not derivable from $\mathcal{R}_{P_0}$, for all $\mathsf{x} \in Secr$, $\mathsf{com}(\mathsf{x})$ is not derivable from $\mathcal{R}_{P_0}$. (Otherwise, bad would be derivable by (Rcom).) Hence, $p_1 \notin Secr$, so $N_1 \notin Secr$.

We can then conclude by Proposition 1. $\square$

B Correctness of the Solving Algorithm

This proof extends the proof of soundness and completeness of resolution with free selection (see for example [11, 38], although their proof technique is different from ours). The main novelties are the proof of the simplification steps for $\mathsf{testunif}$, in Lemma 7, and the proof of Lemma 1.

Lemma 2 *If a variable occurs in the conclusion of a clause, or in a* testunif *fact, then it also occurs in non-*testunif *facts in its hypothesis.*

Proof During the building of clauses by $[\![P]\!]\rho sh$, the variables in the image of ρ also occur in h, and the variables in the conclusion of generated clauses or in testunif facts of generated clauses are in the image of ρ. So if a variable occurs in the conclusion of a clause or in a testunif fact, then it also occurs in non-testunif facts in its hypothesis. So we have the desired property for the clauses of $[\![P]\!]\rho sh$. The desired property is also easy to check for the other clauses of $\mathcal{R}_{P_0}$. The invariant is preserved by the solving algorithm. $\square$

Lemma 3 *After simplification, if* $\mathsf{sel}(R) = \emptyset$*, then the conclusion of* R *is not of the form* $\mathsf{att}(x)$ *for a variable* x*.*

Proof If $\mathsf{sel}(R) = \emptyset$ and its conclusion was $\mathsf{att}(x)$, $R = \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \Rightarrow \mathsf{att}(x)$. By Lemma 2, the variables in the conclusion of R also occur in the hypothesis of R, R would be a tautology, so it would have been eliminated by *elimtaut*. $\square$

By the above lemma, the only case in which we resolve on facts of the form $\mathsf{att}(x)$ is for the clause (Ratteq). This is important because it strongly limits the number of possible resolutions.

Let $\mathcal{R}_0 = \mathcal{R}_{P_0}$ be the initial set of rules, $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$ be the final set of rules, and $\mathcal{R}$ be the set of clauses during the saturation process. At the end of the saturation algorithm, we have $\mathcal{R}_1 = \{R \in \mathcal{R} \mid \mathsf{sel}(R) = \emptyset\}$.

Lemma 4 *At the end of the saturation,* $\mathcal{R}$ *satisfies the following properties:*

1. *For all* $R \in$ *simplify*$(\mathcal{R}_0)$*, there exists* $R' \in \mathcal{R}$ *such that* $R' \sqsupseteq R$*.*
2. *Let* $R \in \mathcal{R}$ *and* $R' \in \mathcal{R}$*. Assume that* $\mathsf{sel}(R) = \emptyset$ *and there exists* $F_0 \in \mathsf{sel}(R')$ *such that* $R \circ_{F_0} R'$ *is defined. In this case, for all* $R'' \in$ *simplify*$(R \circ_{F_0} R')$*, there exists* $R''' \in \mathcal{R}$ *such that* $R''' \sqsupseteq R''$*.*

Proof To prove the first property, let $R \in$ *simplify*$(\mathcal{R}_0)$. We show that during the whole execution of the saturation, there exists $R' \in \mathcal{R}$ such that $R' \sqsupseteq R$.

At the beginning, the algorithm first builds *simplify*$(\mathcal{R}_0)$ (which obviously satisfies the required property), then removes subsumed clauses by *condense*. The property is preserved by elimination of subsumed clauses. So $\mathcal{R} =$ *condense*$(\mathcal{R}_0)$ satisfies the property. Further additions of clauses and eliminations of subsumed clauses preserve the property, so we have the result.

The second property simply means that the fixpoint is reached at the end of saturation. $\square$

Let $\mathcal{T}_{\mathsf{testunif}}$ be the set of true testunif facts.

Definition 7 (Derivation) Let F be a closed fact. Let $\mathcal{R}$ be a set of rules. A derivation of F from $\mathcal{R}$ is a finite tree defined as follows:

1. Its nodes (except the root) are labelled by rules $R \in \mathcal{R}$ or true testunif facts, in $\mathcal{T}_{\mathsf{testunif}}$;

2. Its edges are labelled by closed facts;

3. If the tree contains a node labelled by R with one incoming edge labelled by F_0 and n outgoing edges labelled by $F_1, \ldots, F_n$, then $R \sqsupseteq \{F_1, \ldots, F_n\} \Rightarrow F_0$. If the tree contains a node labelled by a true testunif fact, this node has one incoming edge labelled by the same testunif fact and no outgoing edge.

4. The root has one outgoing edge, labelled by F.

In a derivation, if there is a node labelled by R with one incoming edge labelled by F_0 and n outgoing edges labelled by $F_1, \ldots, F_n$, then the rule R can be used to infer F_0 from $F_1, \ldots, F_n$. Therefore, there exists a derivation of F from B if and only if F can be inferred from rules in B (in classical logic).

The key idea of the proof of the algorithm is the following. Assume that F is derivable from $\mathcal{R}_0$ and consider a derivation of F from $\mathcal{R}_0$. Assume that the rules R and R' are applied one after the other in the derivation of F. Also assume that these rules have been combined by $R \circ_{F_0} R'$, yielding rule R''. In this case, we replace R' by R'' in the derivation of F. When no more replacement can be done, we show that all remaining rules have no selected hypothesis. Then all these rules are in $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$, and we have built a derivation of F from $\mathcal{R}_1$. Moreover, this replacement process terminates because the number of nodes of the derivation strictly decreases.

Lemma 5 *Consider a derivation containing a node n', labelled R'. Let F_0 be an hypothesis of R'. Then there exists a son n of n', labelled R, such that the edge $n \to n'$ is labelled by an instance of F_0, $R \circ_{F_0} R'$ is defined, and we still have a derivation if we replace the nodes n and n' by a node n'' labelled $R'' = R \circ_{F_0} R'$.*

Proof This proof is illustrated in Figure 4. Let $R' = H' \Rightarrow C'$, H'_1 be the multiset of the labels of the outgoing edges of n', and C'_1 the label of its incoming edge. We have $R' \sqsupseteq (H'_1 \Rightarrow C'_1)$, then there exists σ such that $\sigma H' \subseteq H'_1$ and $\sigma C' = C'_1$. Then there is an outgoing edge of n' labelled σF_0, since $\sigma F_0 \in H'_1$. Let n be the node at the end of this edge, let $R = H \Rightarrow C$ be the label of n. We rename the variables of R such that they are distinct

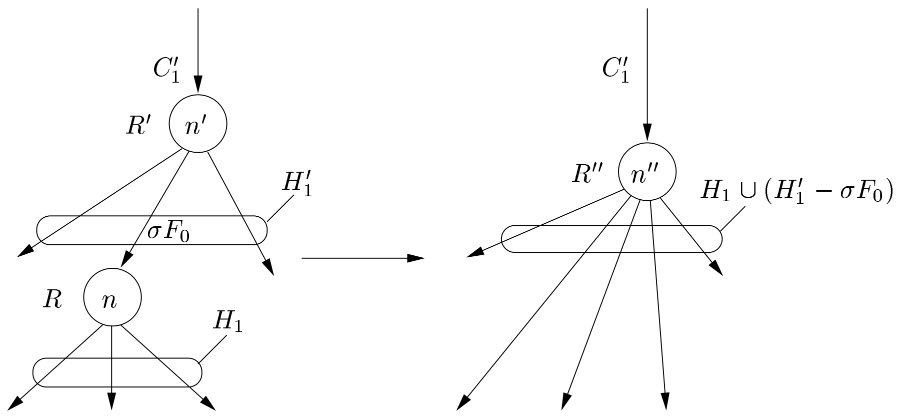

Figure 4: Merging of nodes of Lemma 5

from the variables of R'. Let H_1 be the multiset of the labels of the outgoing edges of n. Then $R \sqsupseteq (H_1 \Rightarrow \sigma F_0)$. By the above choice of distinct variables, we can then extend σ such that $\sigma H \subseteq H_1$ and $\sigma C = \sigma F_0$.

The edge $n' \rightarrow n$ is labelled σF_0, instance of F_0. We have $\sigma C = \sigma F_0$, then C and F_0 are unifiable, then $R \circ_{F_0} R'$ is defined. Let σ' be the most general unifier of C and F_0, and σ'' such that $\sigma = \sigma''\sigma'$. We have $R \circ_{F_0} R' = \sigma'(H \cup (H' - F_0)) \Rightarrow \sigma' C'$. Moreover, $\sigma''\sigma'(H \cup (H' - F_0)) \subseteq H_1 \cup (H'_1 - \sigma F_0)$ and $\sigma''\sigma' C' = \sigma C' = C'_1$. Then $R'' = R \circ_{F_0} R' \sqsupseteq (H_1 \cup (H'_1 - \sigma F_0)) \Rightarrow C'_1$. The multiset of labels of outgoing edges of n'' is precisely $H_1 \cup (H'_1 - \sigma F_0)$ and the label of its incoming edge is C'_1, therefore we have obtained a correct derivation by replacing n and n' by n''. □

Lemma 6 *If D is a derivation whose node n is labelled R, one obtains a derivation D' by replacing R with a clause R' such that $R' \sqsupseteq R$ at node n.*

Proof Let H be the multiset of labels of outgoing edges of the considered node n, and C be the label of its incoming edge. We have $R \sqsupseteq H \Rightarrow C$. By transitivity of $\sqsupseteq$, $R' \sqsupseteq H \Rightarrow C$. So we can replace R with R'. □

For each simplification function f, among *elimattx*, *elimtaut*, *elimdup*, *unify*, *swap*, *elimEVar*, *instantiate*, *elimvar*, *elimtestuniffalse*, and *simptestuniftrue*, we show the following lemma:

Lemma 7 *If D is a derivation whose node n is labelled R, one obtains a derivation D' of the same fact as D by replacing R with some $R' \in f(\{R\}) \cup$ NeededClauses at node n or removing n, and perhaps replacing sons of n labelled by a true* testunif *fact. (To prove this, we may assume that R is in the result of simplifications steps that are done before f in the definition of simplify.)*

If D' contains a node n labelled $R' \in f(\{R\})$, then there exists a derivation D using the clauses of D' except R', the clauses of $\mathcal{R}_0$, true $\mathsf{testunif}$ *facts, and R, that derives the same fact as D'.*

where *NeededClauses* is a set of clauses that must be in the clause set for the transformation to be correct. The set *NeededClauses* is empty, except for $f = \mathit{instantiate}$, $\mathit{NeededClauses} = \{\mathsf{att}(\mathsf{x}) \Rightarrow \mathsf{bad} \,|\, \mathsf{x} \in \mathit{Secr}\}$, and for $f = \mathit{elimvar}$, $\mathit{NeededClauses} = \{(\mathrm{Ratteq})\}$. The proof of these results follows closely the intuition for correctness given in Section 5.

By combining these lemmas, we get the same result for $f = \mathit{simplify}$, with $\mathit{NeededClauses} = \{(\mathrm{Ratteq})\} \cup \{\mathsf{att}(\mathsf{x}) \Rightarrow \mathsf{bad} \mid \mathsf{x} \in \mathit{Secr}\}$.

Proof (for *elimattx*) Direct point obvious: R' has less hypotheses than R, so $R' \sqsupseteq R$.

Conversely, take some closed pattern p such that $\mathsf{att}(p)$ is derivable from $\mathcal{R}_0$ (there exists an infinite number of such p). Build a derivation D by replacing R' with R at node n and adding a derivation of $\mathsf{att}(p)$ as a subtree of n. This derivation is also well-formed. □

Proof (for *elimtaut*) Obvious: we remove n and replace it by one of its subtrees. Converse obvious since $f(\{R\}) = \emptyset$ or $f(\{R\}) = \{R\}$. □

Proof (for *elimdup*) Direct point obvious: R' has less hypotheses than R, so $R' \sqsupseteq R$.

Conversely, we can form a derivation using R instead of R' by just duplicating the subtrees that derive the duplicate hypotheses of R. □

Proof (for *unify*) Let $F = \mathsf{testunif}(p_1, p_2)$ be the fact modified by *unify*. We are going to show that if $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$, then *unify* keeps the clause and replaces F with some F' in it, and $\sigma F' \in \mathcal{T}_{\mathsf{testunif}}$. Conversely, if *unify* transforms F into F' and $\sigma F' \in \mathcal{T}_{\mathsf{testunif}}$, then $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$.

It is easy to infer the lemma from this property. Indeed, $R = H \wedge F \Rightarrow C$. Let H' be the multiset of labels of outgoing edges of n and C' the label of its incoming edge. Since D is a derivation, there exists σ such that $\sigma H \wedge \sigma F \subseteq H'$, and $\sigma C = C'$. Then σF is derived by a son of n, so $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$. Then by the above property $\sigma F' \in \mathcal{T}_{\mathsf{testunif}}$, and D' obtained from D by replacing R with $R' = H \wedge F' \Rightarrow C$ and σF with $\sigma F'$ as son of n' is also a derivation. Conversely, if D' is a derivation, there exists σ such that $\sigma H \wedge \sigma F' \subseteq H'$, and $\sigma C = C'$. Then $\sigma F' \in \mathcal{T}_{\mathsf{testunif}}$. By the above property, $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$ and D obtained by replacing R' with R at node n, and $\sigma F'$ with σF as son of n is a derivation of the same fact as D'.

Assume now $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$. Then, there exists σ'' closed substitution of domain $\mathit{Secr} \cup \mathit{EVar}$ such that $\sigma''\mathit{Secr}$ does not contain bound names and $\sigma''\sigma p_1 = \sigma''\sigma p_2$, and there exists no σ' closed substitution of domain EVar such that $\sigma'\sigma p_1 = \sigma'\sigma p_2$.

Then p_1 and p_2 unify, when considering elements of $\mathit{Secr} \cup \mathit{EVar}$ as variables. Let σ_u be their most general unifier. (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.)

For all $\mathsf{x} \in \mathit{Secr}$, $\sigma_u \mathsf{x}$ is more general than $\sigma'' \mathsf{x}$ (σ modifies variables, not elements of Secr, so $\sigma''\sigma\mathsf{x} = \sigma''\mathsf{x}$), so $\sigma_u\mathsf{x}$ does not contain bound names.

Then $\mathit{unify}(\{R\}) = \{H \wedge F' \Rightarrow C\}$ with

$$F' = \mathsf{testunif}((x_1, \dots, x_n), (\sigma_u x_1, \dots, \sigma_u x_n))$$

where $x_1, \dots, x_n$ are all variables modified by σ_u.

For all σ substituting variables and elements of $\mathit{Secr} \cup \mathit{EVar}$, we have: $\sigma p_1 = \sigma p_2$ if and only if $\sigma(x_1, \dots, x_n) = \sigma\sigma_u(x_1, \dots, x_n)$. Indeed, if $\sigma p_1 = \sigma p_2$, then there exists σ'' such that $\sigma = \sigma''\sigma_u$. So we have $\sigma(x_1, \dots, x_n) = \sigma''\sigma_u(x_1, \dots, x_n) = \sigma''\sigma_u\sigma_u(x_1, \dots, x_n) = \sigma\sigma_u(x_1, \dots, x_n)$ (since $\sigma_u\sigma_u = \sigma_u$, the variables of the image of σ_u being unchanged by σ_u). Conversely, if $\sigma(x_1, \dots, x_n) = \sigma\sigma_u(x_1, \dots, x_n)$, then for all variables x of p_1 and p_2, $\sigma x = \sigma\sigma_u x$ (σ_u does not change variables other than $x_1, \dots, x_n$), so we have $\sigma p_1 = \sigma\sigma_u p_1 = \sigma\sigma_u p_2 = \sigma p_2$.

Since $\sigma''\sigma p_1 = \sigma''\sigma p_2$, we have $\sigma''\sigma(x_1, \dots, x_n) = \sigma''\sigma\sigma_u(x_1, \dots, x_n)$. Since for all σ' of domain EVar, $\sigma'\sigma p_1 \neq \sigma'\sigma p_2$, we have $\sigma'\sigma(x_1, \dots, x_n) \neq \sigma'\sigma\sigma_u(x_1, \dots, x_n)$. So $\sigma F' \in \mathcal{T}_{\mathsf{testunif}}$, and D' is also a derivation.

Conversely, assume $\sigma F' \in \mathcal{T}_{\mathsf{testunif}}$. Then there exists σ'' closed substitution of domain $\mathit{Secr} \cup \mathit{EVar}$ such that $\sigma''\mathit{Secr}$ does not contain bound names and $\sigma''\sigma(x_1, \dots, x_n) = \sigma''\sigma\sigma_u(x_1, \dots, x_n)$, and there exists no σ' closed substitution of domain EVar such that $\sigma'\sigma(x_1, \dots, x_n) = \sigma'\sigma\sigma_u(x_1, \dots, x_n)$. Then by the property above, $\sigma''\sigma p_1 = \sigma''\sigma\sigma_u p_2$ and there exists no σ' of domain EVar such that $\sigma'\sigma p_1 \neq \sigma'\sigma p_2$, so $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$. $\square$

Proof (for *swap*) Let $R = H \wedge \mathsf{testunif}(p_1, p_2) \Rightarrow C$, and $R' = H \wedge \mathsf{testunif}(p'_1, p'_2) \Rightarrow C$. Since $\{R'\} = \mathit{swap}(\{R\})$, we can easily show that for all σ, $\sigma p_1 = \sigma p_2$ if and only if $\sigma p'_1 = \sigma p'_2$. Then we conclude as in the case of *unify*. $\square$

Proof (for *elimEVar*) Let $F = \mathsf{testunif}((\mathsf{e}, p_1, \dots, p_n), (p'_0, \dots, p'_n))$ with $\mathsf{e} \in \mathit{EVar}$ be transformed into $F' = \mathsf{testunif}((p_1, \dots, p_n), (p'_1, \dots, p'_n))$ by *elimEVar*.

Assume $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$. Then there exists σ' of domain $\mathit{Secr} \cup \mathit{EVar}$ such that $\sigma'\mathit{Secr}$ does not contain bound names and $\sigma'\sigma(\mathsf{e}, p_1, \dots, p_n) = \sigma'\sigma(p'_0, \dots, p'_n)$. So $\sigma'\sigma(p_1, \dots, p_n) = \sigma'\sigma(p'_1, \dots, p'_n)$. Moreover, there exists no σ'' of domain EVar such that $\sigma''\sigma(\mathsf{e}, p_1, \dots, p_n) = \sigma''\sigma(p'_0, \dots, p'_n)$. Then there exists no σ''_1 of domain EVar such that $\sigma''_1\sigma(p_1, \dots, p_n) = \sigma''_1\sigma(p'_1, \dots, p'_n)$. Indeed, if σ''_1 existed, $\sigma'' = \sigma''_1[\mathsf{e} \mapsto \sigma''_1\sigma p'_0]$ would contradict the non-existence of σ''. Then $\sigma F' \in \mathcal{T}_{\mathsf{testunif}}$.

Assume $\sigma F' \in \mathcal{T}_{\mathsf{testunif}}$. Then there exists σ' of domain $\mathit{Secr} \cup \mathit{EVar}$ such that $\sigma'\mathit{Secr}$ does not contain bound names and $\sigma'\sigma(p_1, \dots, p_n) = \sigma'\sigma(p'_1, \dots, p'_n)$. Let $\sigma'_1 = \sigma'[\mathsf{e} \mapsto \sigma'\sigma p'_0]$. Then $\sigma'_1\sigma(\mathsf{e}, p_1, \dots, p_n) = \sigma'_1\sigma(p'_0, \dots, p'_n)$. Moreover, there exists no σ'' of domain EVar such that $\sigma''\sigma(p_1, \dots, p_n) = \sigma''\sigma(p'_1, \dots, p'_n)$. So there exists no σ'' of domain EVar such that $\sigma''\sigma(\mathsf{e}, p_1, \dots, p_n) = \sigma''\sigma(p'_0, \dots, p'_n)$. Then $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$.

Then we conclude as in the case of *unify*. □

Proof (for *instantiate*) Let $R = H \Rightarrow \mathsf{bad}$, where $F = \mathsf{testunif}(p_1, p_2) \in H$ is the fact concerned by *instantiate*. Let x and $f(p'_1, \dots, p'_n)$ be the matching patterns in F, $\mathsf{att}(x) \in H$ and $R' = R\{f(x_1, \dots, x_n)/x\}$.

Let H' be the multiset of labels of outgoing edges of n and C' the label of its incoming edge. Since D is a derivation, there exists σ such that $\sigma H \subseteq H'$, and $\sigma \mathsf{bad} = \mathsf{bad} = C'$. Then $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$. Then there exists σ'' of domain $\mathit{Secr} \cup \mathit{EVar}$ such that $\sigma''\sigma p_1 = \sigma''\sigma p_2$. In particular, $\sigma''\sigma x = \sigma''\sigma f(p'_1, \dots, p'_n)$. The term σx is closed, so we can have the following cases.

- First case: σx is an instance of $f(x_1, \dots, x_n)$. Then σR is an instance of R', so D' obtained by replacing R with R' in D is also a derivation.

- Second case: $\sigma x = \mathsf{x} \in \mathit{Secr}$. Since $\mathsf{att}(x) \in H$, $\mathsf{att}(\mathsf{x}) \in \sigma H \subseteq H'$. Then D' obtained by replacing R with $\mathsf{att}(\mathsf{x}) \Rightarrow \mathsf{bad}$ in D is also a derivation.

- Third case: $\sigma x \in \mathit{EVar}$. Since $\mathsf{att}(x) \in H$, we would have $\mathsf{att}(\sigma x) \in \sigma H \subseteq H'$, and this is impossible since $\mathsf{att}(\mathsf{e})$ with $\mathsf{e} \in \mathit{EVar}$ is never derivable. (Elements of EVar occur only in $\mathsf{testunif}$ facts.)

Conversely, assume that D' is a derivation. Let $R' = H\{f(x_1, \dots, x_n)/x\} \Rightarrow \mathsf{bad}$. Then there exists σ such that $\sigma(H\{f(x_1, \dots, x_n)/x\}) \subseteq H''$, and $\sigma \mathsf{bad} = C'$. Taking σ' defined by $\sigma' = \sigma\{x \mapsto f(x_1, \dots, x_n)\}$, we have $\sigma' H = \sigma(H\{f(x_1, \dots, x_n)/x\})$, so $\sigma' H \subseteq H'$, and $\sigma' \mathsf{bad} = \mathsf{bad} = C'$. Then D obtained by replacing R' with R is also a derivation. □

Proof (for *elimvar*) Let $R = H \Rightarrow C$, where $F = \mathsf{testunif}(p_1, p_2) \in H$ is the fact concerned by *elimvar*. Let x_1 and x_2 be the variables such that $R' = R\{x_1/x_2\}$.

Let H' be the multiset of labels of outgoing edges of n and C' the label of its incoming edge. Since D is a derivation, there exists σ such that $\sigma H \subseteq H'$, and $\sigma C = C'$. Then $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$.

- First case: $\sigma x_1 = \sigma x_2$. Then $\sigma R' = \sigma R$, so D' obtained by replacing R with R' in D is a derivation.

- Second case: $\sigma x_1 \neq \sigma x_2$. Since $\sigma F \in \mathcal{T}_{\mathsf{testunif}}$, there exists σ'' of domain $\mathit{Secr} \cup \mathit{EVar}$ such that $\sigma''\mathit{Secr}$ contains no bound names and $\sigma''\sigma p_1 =$

$\sigma''\sigma p_2$ so $\sigma''\sigma x_1 = \sigma''\sigma x_2$. Then $\sigma\mathsf{testunif}((x_1),(x_2)) \in \mathcal{T}_{\mathsf{testunif}}$. Moreover $\mathsf{att}(x_1) \in H$, so $\sigma\mathsf{att}(x_1) \in \sigma H \subseteq H'$, and similarly for $\mathsf{att}(x_2)$. Then D' obtained by replacing R with (Ratteq) $\mathsf{att}(x_1) \wedge \mathsf{att}(x_2) \wedge \mathsf{testunif}((x_1),(x_2)) \Rightarrow \mathsf{bad}$ and σF with $\sigma\mathsf{testunif}((x_1),(x_2))$ in D is a derivation.

Conversely, assume that D' is a derivation. Let $R' = H\{x_1/x_2\} \Rightarrow C\{x_1/x_2\}$. Then there exists σ such that we have $\sigma(H\{x_1/x_2\}) \subseteq H'$, and $\sigma(C\{x_1/x_2\}) = C'$. Extending σ to x_2 by $\sigma x_2 = \sigma x_1$, we have $\sigma H = \sigma(H\{x_1/x_2\})$ and $\sigma C = \sigma(C\{x_1/x_2\})$, so $\sigma H \subseteq H'$, and $\sigma C = C'$. Then D obtained by replacing R' with R is also a derivation. $\square$

Proof (for *elimtestuniffalse***)** We have $\mathsf{testunif}((),()) \notin \mathcal{T}_{\mathsf{testunif}}$, so if R appears in a derivation, it does not contain $\mathsf{testunif}((),())$ as hypothesis, so we obtain $\mathit{elimtestuniffalse}(\{R\}) = \{R\}$, hence the result.

The converse part is obvious. $\square$

Proof (for *simptestuniftrue***)** Assume that $\mathit{simptestuniftrue}(\{R\}) = \{\mathsf{bad}\}$ (otherwise $\mathit{simptestuniftrue}(\{R\}) = \{R\}$ and the result is obvious). The direct part is obvious (since R concludes bad). Let us prove the converse.

We have

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

H containing only hypotheses of the form $\mathsf{att}(x)$, and for all $i \in \{1,\ldots,n\}$, $p_i \in \mathit{Secr}$. R is obtained after the previous simplifications.

Let p be any pattern such that $\mathsf{att}(p)$ is derivable from $\mathcal{R}_0$. Let σ_1 be the substitution that maps all variables of H to p. Let D be obtained by replacing nodes labelled bad in D' with a subtree whose root n is labelled by R and the sons of n consist of derivations of $\mathsf{att}(p)$ and of the fact $\sigma_1\mathsf{testunif}((p_1,\ldots,p_n),(p'_1,\ldots,p'_n))$. Then we show that D is a derivation. The only point to check is that all hypotheses of $\sigma_1 R$ have been derived by sons of n in D. The fact $\sigma_1 H = \{\mathsf{att}(p)\}$ is derived in D by sons of n. Moreover $\sigma_1\mathsf{testunif}((p_1,\ldots,p_n),(p'_1,\ldots,p'_n)) \in \mathcal{T}_{\mathsf{testunif}}$. To show this point, 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 $\sigma'\sigma_1 p'_i$. Hence the $\mathsf{testunif}$ fact is true as announced. $\square$

Proposition 8 *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 is to replace rules as allowed by the previous lemmas. 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 the number of nodes of the derivation.

Proof Let us first prove that if there exists a derivation of F from $\mathcal{R}_0$, then there is a derivation of F from $\mathsf{saturate}(\mathcal{R}_0)$. Let us consider a derivation of F from $\mathcal{R}_0$.

For each rule R in $\mathcal{R}_0$, for each $R'' \in \mathit{simplify}(R)$, there exists a rule R' in $\mathcal{R}$ such that $R' \sqsupseteq R''$ (Lemma 4, Property 1). Assume that there exists a node labelled R in this derivation. By Lemma 7, we can replace R with some $R'' \in \mathit{simplify}(R)$. By Lemma 6, we can replace R'' with R'. Therefore, we can replace nodes labelled by R with nodes labelled by R'. This way, we obtain a derivation of F from $\mathcal{R}$.

We build a derivation of F whose rules are in $\mathcal{R}_1 \cup \mathcal{T}_{\mathsf{testunif}}$, where $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$.

- The rules R at the leafs of the derivation are in $\mathcal{R}_1 \cup \mathcal{T}_{\mathsf{testunif}}$ since they are in $\mathcal{R} \cup \mathcal{T}_{\mathsf{testunif}}$ and they have no hypotheses, so $\mathsf{sel}(R) = \emptyset$.

- Assume that we have a derivation D such that all sons of a node n' are labelled by rules in $\mathcal{R}_1 \cup \mathcal{T}_{\mathsf{testunif}}$. Let R' be the rule labelling n'. If $R' \notin \mathcal{R}_1 \cup \mathcal{T}_{\mathsf{testunif}}$, $\mathsf{sel}(R') \neq \emptyset$. Take $F_0 \in \mathsf{sel}(R')$. By Lemma 5, there exists a son n of n' labelled R, such that $R \circ_{F_0} R'$ is defined. By induction hypothesis, $\mathsf{sel}(R) = \emptyset$. Moreover, by definition of the selection function, F_0 is not a $\mathsf{testunif}$ fact, so $R \notin \mathcal{T}_{\mathsf{testunif}}$, so $R \in \mathcal{R}$. Then by Lemma 4, Property 2, for each $R''' \in \mathit{simplify}(R \circ_{F_0} R')$, there exists $R'' \in \mathcal{R}$ such that $R'' \sqsupseteq R'''$. By Lemma 5, we can replace n and n' with n'' labelled by $R \circ_{F_0} R'$. By Lemma 7, we can replace $R \circ_{F_0} R'$ with some $R''' \in \mathit{simplify}(R \circ_{F_0} R') \cup \{(\text{Ratteq})\} \cup \{\mathsf{att}(\mathsf{x}) \Rightarrow \mathsf{bad} \mid \mathsf{x} \in \mathit{Secr}\}$. The rule (Ratteq) is subsumed by some rule in $\mathcal{R}$, since it is obtained by simplification from (Rt) for $g = \mathit{eq}$. Similarly, for all $\mathsf{x} \in \mathit{Secr}$, $\mathsf{att}(\mathsf{x}) \Rightarrow \mathsf{bad}$ is subsumed by some rule in $\mathcal{R}$, since it is inferred from (Rc) and (Rcom). So, in all cases, there exists $R'' \in \mathcal{R}$ such that $R'' \sqsupseteq R'''$. By Lemma 6, we can replace R''' with R'', and we obtain a derivation D' of F from $\mathcal{R}$, containing fewer nodes as D (since the resolution of two clauses removes one node, and simplifications do not add nodes).

  Since the number of nodes strictly decreases, this replacement process terminates.

  When we cannot perform this replacement any more, $\mathsf{sel}(R') = \emptyset$, so $R' \in \mathcal{R}_1$.

By induction, this proof shows that we can build a derivation of F from $\mathcal{R}_1$, which is the expected result.

For the converse implication, notice that if a fact is derivable from $\mathcal{R}_1$ then it is derivable from $\mathcal{R}$, and that all rules added to $\mathcal{R}$ do not create derivations of new facts: when composing two rules R and R', the created rule can derive facts that could also by derived by R and R'; by Lemma 7, the simplification does not create derivations of new facts. $\square$

Proof of Lemma 1 *After simplification, if* $R = H \Rightarrow \mathsf{bad}$ *and* H *is not empty, then* $\mathsf{sel}(R) \neq \emptyset$.

Proof If $\mathsf{sel}(R)$ was empty, H contains only $\mathsf{att}(x_i)$ and $\mathsf{testunif}$ facts. If H contains only $\mathsf{att}(x_i)$ facts, these are removed by *elimattx*. Otherwise, $H = H' \wedge \mathsf{testunif}((p_1, \dots, p_n), (p'_1, \dots, p'_n))$ where H' contains facts $\mathsf{att}(x_i)$ and, by the above simplification process, there exists $i \in \{1, \dots, n\}$ such that p_i is a variable (which then occurs in some $\mathsf{att}(x_i)$ in H', since variables in $\mathsf{testunif}$ facts always occur in att hypotheses of the clause, by Lemma 2) and p'_i is not a variable and not in $Secr \cup EVar$.

Indeed, after *unify*, all pairs p_i, p'_i can be:

1. $p_i \in EVar$, eliminated by *elimEVar*,
2. p_i variable, then $p'_i \in Secr \cup EVar$ is eliminated by *swap*, and p'_i variable is eliminated by *elimvar*. The only remaining possibility is p'_i not a variable and not in $Secr \cup EVar$.
3. $p_i \in Secr$.

If there exists no pair such that p_i is a variable and p'_i not a variable and not in $Secr \cup EVar$, then all pairs are in case 3, which is eliminated by *simptestuniftrue*.

Since p'_i is not a variable and not in $Secr \cup EVar$, it is $f(p_1, \dots, p_n)$. Then *instantiate* can be applied, thus making this situation disappear. $\square$

Proof of Proposition 3 $\mathsf{bad} \in \mathsf{saturate}(\mathcal{R}_{P_0})$ *if and only if* bad *is derivable from* $\mathcal{R}_{P_0}$.

Proof By Proposition 8, bad is derivable from $\mathcal{R}_{P_0}$ if and only if bad is derivable from $\mathsf{saturate}(\mathcal{R}_{P_0})$. Since all rules $H \Rightarrow \mathsf{bad}$ with H not empty have non-empty selection by Lemma 1, they are not in $\mathsf{saturate}(\mathcal{R}_{P_0})$, so if bad is derivable from $\mathsf{saturate}(\mathcal{R}_{P_0})$ then $\mathsf{bad} \in \mathsf{saturate}(\mathcal{R}_{P_0})$. The converse is obvious, hence the result. $\square$

C Termination

By Condition C2, the input and output constructs in the protocol always work on a public channel c. Then the facts $\mathsf{mess}(c, p)$ are replaced by $\mathsf{att}(p)$ in all clauses. The only remaining clauses containing mess are (Rl) and (Rs). Since $\mathsf{mess}(x, y)$ is selected in these clauses, the only inference with these clauses is to combine (Rs) with (Rl), and it yields a tautology which is immediately removed. Therefore, we can ignore these clauses in our termination proof.

The patterns of a fact $pred(p_1, \ldots, p_n)$ are $p_1, \ldots, p_n$. The *patterns of* a clause R are the patterns of all facts in R, and we denote the set of patterns of R by $patterns(R)$. A pattern is said to be *non-data* when it is not of the form $f(\ldots)$ with f a data constructor. The set $sub(S)$ contains the subterms of patterns in the set S.

Definition 8 (Reference clause) We define a function VarSubst, which substitutes all elements of *Secr* with distinct new variables, in a term, fact, sequence of facts, or clause.

Let R be a clause. If R is

$$\mathsf{att}(p_1) \wedge \ldots \wedge \mathsf{att}(p_n) \wedge \mathsf{testunif}(p, p') \Rightarrow \mathsf{bad} \tag{7}$$

and p and p' unify, considering elements of $EVar$ as variables, then the clause TestUnifRef(R) is $\mathsf{att}(\sigma_u p_1) \wedge \ldots \wedge \mathsf{att}(\sigma_u p_n) \Rightarrow \mathsf{bad}$ where σ_u is the most general unifier of p and p'. If R is of the form (7) but p and p' do not unify when considering elements of $EVar$ as variables, TestUnifRef(R) is undefined. If R does not contain a $\mathsf{testunif}$ fact, TestUnifRef$(R) = R$.

Let *Keys* be the set containing the arguments of *host* and *pk* and the second arguments of *sign* and *nmrsign* in the terms manipulated in the execution of Condition C5. We define a function OneKey from patterns to patterns which replaces all elements of *Keys* with a single atomic constant k_0. We extend this function naturally to facts, clauses, and sets of patterns.

We define $\mathrm{Rref}_1(R) = \mathrm{TestUnifRef}(\mathrm{VarSubst}(R))$. The reference clause of R, $\mathrm{Rref}(R)$, is then $\mathrm{Rref}(R) = \mathrm{OneKey}(\mathrm{Rref}_1(R))$.

Note that $\mathrm{Rref}(R)$ may contain several times the same hypothesis. These duplicate hypotheses are kept; this is useful in the following of the proof.

Definition 9 (Weakly tagged program) A program $\mathcal{R}_0$ is *weakly tagged* if there exists a finite set of closed patterns S_0 and a set of patterns *tagGen* such that:

W1. The elements of *Secr* do not occur in strict subterms of patterns of *tagGen*.

W2. If two patterns p_1 and p_2 in *tagGen* unify, p'_1 is an instance of p_1 in $sub(S_0)$, and p'_2 is an instance of p_2 in $sub(S_0)$, then $p'_1 = p'_2$.

W3. $\mathcal{R}_0$ is included in $\mathcal{R}_{\mathrm{Prot0}} \cup \mathcal{R}_{\mathrm{Prot1}}$.

with the following definitions of $\mathcal{R}_{\mathrm{Prot0}}$ and $\mathcal{R}_{\mathrm{Prot1}}$:

- A pattern is *top-tagged* when it is an instance of a pattern in *tagGen*.

- A pattern is *fully tagged* when all its non-variable, non-element of $EVar$, non-data subterms are top-tagged.
- $\mathcal{R}_{\text{Prot0}}$ is the set of clauses R such that $\text{Rref}(R)$ is of the form

  $$\mathsf{att}(p_1) \wedge \ldots \wedge \mathsf{att}(p_n) \Rightarrow C$$

  where C can be $\mathsf{att}(p)$, $\mathsf{com}(p)$, or bad, there exists a substitution σ such that $\mathit{patterns}(\sigma\text{Rref}(R)) \subseteq \mathit{sub}(S_0)$, the arguments of com are fully tagged, and for each argument p of att either

  - p is fully tagged, or
  - the strict subterms of p are fully tagged, p is more general than every pattern in $\mathit{sub}(S_0)$ that has the same root function symbol as p. For each variable x of p, either x has the same image by all substitutions that map p to an element of $\mathit{sub}(S_0)$, or x does not appear in strict subterms of patterns of $\text{Rref}(R)$ other than p (x may be equal to some patterns of $\text{Rref}(R)$).
- $\mathcal{R}_{\text{Prot1}}$ is the set of clauses R of the form $\mathsf{com}(\mathsf{x}) \Rightarrow \mathsf{bad}$ where $\mathsf{x} \in \mathit{Secr}$.

The following three lemmas are easy to prove.

Lemma 8 *If p is fully tagged and for all x, σx is fully tagged, then σp is fully tagged.*

Lemma 9 *If for all x, σx is fully tagged, and for all x, $\sigma' x$ is fully tagged, then for all x, $(\sigma \circ \sigma')x$ is fully tagged.*

Lemma 10 *If σ is the most general unifier of fully tagged patterns, then for all x, σx is fully tagged.*

C.1 Tagged Implies Weakly Tagged (Detailed Proof)

Proposition 9 *If P_0 is a tagged protocol, then $\mathcal{R}_{P_0}$ is a weakly tagged program.*

The proof of this proposition is very long, so we split it in several lemmas. We also include a shorter proof sketch for people who are not interested in all the technical details in Section C.2.

We assume that different occurrences of restrictions and variables have different identifiers, and identifiers different from free names and variables. We first instrument processes as in [16]: each replication is instrumented with a session identifier i: $!P$ becomes $!^i P$, and each restriction is instrumented with a pattern $a[\ldots]$, containing as arguments all inputs, results of destructor applications, and session identifiers of replications above the

$$
\begin{array}{ll}
(\nu a_1 : p_1)(\nu a_2 : p_2)P \equiv (\nu a_2 : p_2)(\nu a_1 : p_1)P & P \mid 0 \equiv P \\
\qquad \text{if } a_1 \neq a_2,\ a_1 \notin fn(p_2),\ a_2 \notin fn(p_1) & P \mid Q \equiv Q \mid P \\
(\nu a : p)(P \mid Q) \equiv P \mid (\nu a : p)Q \ \text{ if } a \notin fn(P) & (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
P \equiv Q \ \Rightarrow\ P \mid R \equiv Q \mid R & P \equiv P \\
P \equiv Q \ \Rightarrow\ (\nu a : p)P \equiv (\nu a : p)Q & Q \equiv P \ \Rightarrow\ P \equiv Q \\
 & P \equiv Q,\, Q \equiv R \ \Rightarrow\ P \equiv R
\end{array}
$$

$$
\begin{array}{ll}
E, \overline{c}\langle M\rangle.Q \mid c(x).P, \mathcal{M} & \\
\qquad \rightarrow\ E[x \mapsto M], Q \mid P\{M/x\}, \mathcal{M} \cup \{M, c\} & \text{(Red I/O')} \\
E, \textit{let } x = g(M_1, \ldots, M_n) \textit{ in } P \textit{ else } 0, \mathcal{M} \ \rightarrow & \text{(Red Destr')} \\
\quad E[x \mapsto M'], P\{M'/x\}, \mathcal{M} \cup \{M_1, \ldots, M_n, M'\} \text{ if } g(M_1, \ldots, M_n) \rightarrow M' & \\
E, !^i P, \mathcal{M} \ \rightarrow\ E[i \mapsto \mathrm{Id}_0], P\{\mathrm{Id}_0/i\}, \mathcal{M} & \text{(Red Repl')} \\
E, P, \mathcal{M} \ \rightarrow\ E', Q, \mathcal{M}' \ \Rightarrow\ E, P \mid R, \mathcal{M} \ \rightarrow\ E', Q \mid R, \mathcal{M}' & \text{(Red Par')} \\
E, P, \mathcal{M} \ \rightarrow\ E', Q, \mathcal{M}' & \\
\qquad \Rightarrow\ E, (\nu a : p)P, \mathcal{M} \ \rightarrow\ E', (\nu a : p)Q, \mathcal{M}' & \text{(Red Res')} \\
P' \equiv P,\ E, P, \mathcal{M} \ \rightarrow\ E', Q, \mathcal{M}',\ Q \equiv Q' & \\
\qquad \Rightarrow\ E, P', \mathcal{M} \ \rightarrow\ E', Q', \mathcal{M}' & \text{(Red } \equiv\text{')}
\end{array}
$$

Figure 5: Structural equivalence and reduction for instrumented processes

restriction: (νa) becomes $(\nu a : a[\ldots])$. The clauses $[\![P]\!]\rho s h$ can be computed from the instrumented process P' corresponding to P, by removing the sequence s in the arguments of $[\![P']\!]$, and replacing the definitions for replication and restriction with:

$$
\begin{aligned}
[\![!^i P]\!]\rho h &= [\![P]\!](\rho[i \rightarrow i'])(h \wedge \mathsf{att}(i')) \\
[\![(\nu a : p)P]\!]\rho h &= [\![P]\!](\rho[a \mapsto \rho(p)])h
\end{aligned}
$$

In Figure 5, we define a semantics for instrumented processes, which is only used as a tool in the proof. A semantic configuration consists of three components: an environment E mapping variables to terms, an instrumented process P, and a set of terms $\mathcal{M}$. The semantics is defined as a reduction relation on semantic configurations and a structural equivalence relation on processes. Note that, in this semantics, we do not allow renaming of bound names. Indeed, renaming of bound names is useless here, since the replication does not copy processes in this semantics, and the names are initially pairwise distinct.

By hypothesis, P_0 is a tagged protocol. Let P_0' be the instrumented process corresponding to P_0.

Lemma 11 *We have $\{\}, P_0', fn(P_0) \rightarrow^* E', (\nu a_1 : p_1) \ldots (\nu a_n : p_n)0, \mathcal{M}'$, for some E' and $\mathcal{M}'$, such that the second argument of pencrypt in $\mathcal{M}'$ is of* *the form $pk(M)$, the arguments of host and pk and the second arguments of sign and nmrsign in $\mathcal{M}'$ are atomic constants in Keys.*

Proof Let P_1 be the process obtained from P_0 by removing replications. An execution of P_0 in which each program point is executed at most once (as required by Condition C5) is also an execution of P_1. By Condition C2, all inputs and outputs are on a channel c, free name of the processes. By Condition C5, all destructors have a 0 "else" clause. Moreover, since all program points are executed, there remains no destructor, input, or output to execute, so $P_1 \rightarrow^* (\nu a_1) \dots (\nu a_n)0$.

Then we can immediately see that $\{\}, P'_0, fn(P_0)$ reduces to $E', (\nu a_1 : p_1) \dots (\nu a_n : p_n)0, \mathcal{M}'$, by applying the same reduction rules as for reducing P_1 (plus reductions for replications, and up to reorderings of restrictions). Moreover, the second argument of *pencrypt* in $\mathcal{M}'$ is of the form $pk(M)$ by Condition C6. The arguments of *host* and *pk* and the second arguments of *sign* and *nmrsign* in $\mathcal{M}'$ are atomic constants in *Keys*, by Condition C7 and definition of *Keys*. □

By Lemma 11, $\{\}, P'_0, fn(P_0) \rightarrow^* E', (\nu a_1 : p_1) \dots (\nu a_n : p_n)0, \mathcal{M}'$, for some E' and $\mathcal{M}'$, such that the second argument of *pencrypt* in $\mathcal{M}'$ is of the form $pk(M)$, the arguments of *host* and *pk* and the second arguments of *sign* and *nmrsign* in $\mathcal{M}'$ are atomic constants in *Keys*.

We define the set S'_0 of all patterns of terms manipulated by the execution of Condition C5. Formally, S'_0 is defined as follows. Let $E_0 = \{a \mapsto a[\,] \mid a \in fn(P_0)\}$, $E_i = E_{i-1}[a_i \mapsto E_{i-1}(p_i)]$. Let $S'_0 = E_n(\mathcal{M}') \cup \{b[\mathrm{Id}_0]\} \cup \{a[\,] \mid a \in fn(P_0)\}$. Let c, c', c'' be constants. We add the following patterns to the set S'_0: $sencrypt((c, c'), c'')$, $sencrypt_{prob}((c, c'), c'', c')$, $pencrypt_{prob}((c, c'), pk(k_0), c')$, $sign((c, c'), k_0)$, $nmrsign((c, c'), k_0)$, $host(k_0)$ if S'_0 contains no instance of the terms $sencrypt(x, y)$, $sencrypt_{prob}(x, y, z)$, $pencrypt_{prob}(x, y, z)$, $sign(x, y)$, $nmrsign(x, y)$, $host(x)$, respectively. Let $S_0 = \mathrm{OneKey}(S'_0)$. Then S_0 is a finite set of closed patterns. Let $E'' = E_n \cup E_n \circ E'$.

Lemma 12 *Let P be an instrumented process, subprocess of P'_0. Assume that $E, C[E(P)], \mathcal{M} \rightarrow^* E', (\nu a_1 : p_1) \dots (\nu a_n : p_n)0, \mathcal{M}'$, where C is an evaluation context built from $[\,] \mid Q$, $Q \mid [\,]$, and $(\nu a : p)[\,]$, and that there exists σ' such that $E''_{|dom(\rho)} = \sigma' \circ \rho$ and $patterns(\sigma' h) \subseteq sub(S'_0)$. Then for all $R \in [\![P]\!]\rho h$, $\mathrm{TestUnifRef}(R)$ is defined and there exists σ'' such that $patterns(\sigma'' \mathrm{TestUnifRef}(R)) \subseteq sub(S'_0)$.*

Proof Let us denote $\mathrm{PatProp}(R)$ the property to prove for all $R \in [\![P]\!]\rho h$. The proof proceeds by induction on P.

- Case 0: Obvious.

- Case $P \mid Q$: We have $E, C[E(P \mid Q)], \mathcal{M} \to^* E', (\nu a_1 : p_1) \dots (\nu a_n : p_n)0, \mathcal{M}'$. We have $E(P \mid Q) = E(P) \mid E(Q)$. Taking $C' = C[[\,] \mid E(Q)]$,

$$E, C'[E(P)], \mathcal{M} \to^* E', (\nu a_1 : p_1) \dots (\nu a_n : p_n)0, \mathcal{M}'$$

By induction hypothesis, for all $R \in [\![P]\!]\rho h$, $\mathrm{PatProp}(R)$. Similarly, for all $R \in [\![Q]\!]\rho h$, $\mathrm{PatProp}(R)$. Since $[\![P \mid Q]\!]\rho h = [\![P]\!]\rho h \cup [\![Q]\!]\rho h$, for all $R \in [\![P \mid Q]\!]\rho h$, $\mathrm{PatProp}(R)$.

- Case $!^i P$: We have $E, C[E(!^i P)], \mathcal{M} \to^* E', (\nu a_1{:}p_1) \dots (\nu a_n{:}p_n)0, \mathcal{M}'$. We have $E(!^i P) = !^i E(P)$. At some point, $E(!^i P)$ must be reduced in the reduction $E, C[E(!^i P)], \mathcal{M} \to^* E', (\nu a_1 : p_1) \dots (\nu a_n : p_n)0, \mathcal{M}'$, since otherwise $!^i E(P)$ would be present in the final process. So

$$\begin{aligned}
&E, C[E(!^i P)], \mathcal{M} \to^* E'_1, C'[E(!^i P)], \mathcal{M}_1 \\
&\quad \to E'_1[i \mapsto \mathrm{Id}_0], C'[E(P)\{\mathrm{Id}_0/i\}], \mathcal{M}_1 \\
&\quad \to^* E', (\nu a_1 : p_1) \dots (\nu a_n : p_n)0, \mathcal{M}'
\end{aligned}$$

with $(E'_1[i \mapsto \mathrm{Id}_0])(P) = E(P)\{\mathrm{Id}_0/i\}$, since the environment, first component of a semantic configuration, is always extended when the configuration is reduced. Let $\rho' = \rho[i \mapsto i']$ and $h' = h \wedge \mathsf{att}(i')$. We extend σ' by $\sigma'(i') = \mathrm{Id}_0$. Then $E''(i) = \mathrm{Id}_0 = (\sigma' \circ \rho')(i)$, so $E''_{|dom(\rho')} = \sigma' \circ \rho'$. Moreover, $\mathit{patterns}(\sigma' h') = \mathit{patterns}(\sigma' h) \cup \{\mathrm{Id}_0\} \subseteq \mathit{sub}(S'_0)$. By induction hypothesis, for all $R \in [\![P]\!]\rho' h'$, $\mathrm{PatProp}(R)$. Since $[\![!^i P]\!]\rho h = [\![P]\!]\rho' h'$, we have for all $R \in [\![!^i P]\!]\rho h$, $\mathrm{PatProp}(R)$.

- Case $(\nu a{:}p)P$: We have $E, C[E((\nu a{:}p)P)], \mathcal{M} \to^* E', (\nu a_1{:}p_1) \dots (\nu a_n{:} p_n)0, \mathcal{M}'$. We have $E((\nu a : p)P) = (\nu a : E(p))E(P)$. Taking $C' = C[(\nu a : E(p))[\,]]$, we have

$$E, C'[E(P)], \mathcal{M} \to^* E', (\nu a_1 : p_1) \dots (\nu a_n : p_n)0, \mathcal{M}'$$

Let $\rho' = \rho[a \mapsto \rho(p)]$. Since the reduction never removes restrictions, a must be some a_i and $E(p) = p_i$. Then $E''(a) = E_i(a_i) = E_{i-1}(p_i) = E_{i-1}(E(p)) = E''(p) = (\sigma' \circ \rho)(p)$ and $\rho'(a) = \rho(p)$, so $E''_{|dom(\rho')} = \sigma' \circ \rho'$. By induction hypothesis, for all $R \in [\![P]\!]\rho' h$, $\mathrm{PatProp}(R)$. Since $[\![(\nu a{:}p)P]\!]\rho h = [\![P]\!]\rho' h$, we have for all $R \in [\![(\nu a{:}p)P]\!]\rho h$, $\mathrm{PatProp}(R)$.

- Case $\overline{c}\langle M \rangle.P$: We have $E, C[E(\overline{c}\langle M \rangle.P)], \mathcal{M} \to^* E', (\nu a_1{:}p_1) \dots (\nu a_n{:} p_n)0, \mathcal{M}'$. We have $E(\overline{c}\langle M \rangle.P) = \overline{c}\langle E(M) \rangle.E(P)$. At some point, $E(\overline{c}\langle M \rangle.P)$ must be reduced in the reduction $E, C[E(\overline{c}\langle M \rangle.P)], \mathcal{M} \to^* E', (\nu a_1 : p_1) \dots (\nu a_n : p_n)0, \mathcal{M}'$, since otherwise $\overline{c}\langle E(M) \rangle.E(P)$ would be present in the final process. So

$$\begin{aligned}
&E, C[E(\overline{c}\langle M\rangle.P)], \mathcal{M} \rightarrow^* E_1', C'[E(\overline{c}\langle M\rangle.P) \mid c(x).Q], \mathcal{M}_1\\
&\quad \rightarrow E_1'[x \mapsto E(M)], C'[E(P) \mid Q\{E(M)/x\}], \mathcal{M}_1 \cup \{E(M), c\}\\
&\quad = E_2', C''[E(P)], \mathcal{M}_1 \cup \{E(M), c\}\\
&\quad \rightarrow^* E', (\nu a_1 : p_1) \ldots (\nu a_n : p_n)0, \mathcal{M}'
\end{aligned}$$

with $E_2'(P) = E(P)$. By induction hypothesis, for all $R \in [\![P]\!]\rho h$, $\mathrm{PatProp}(R)$.

Moreover, $\mathit{patterns}(\sigma'(h \Rightarrow \mathsf{att}(\rho(M)))) = \mathit{patterns}(\sigma' h) \cup \{\sigma'\rho(M)\} = \mathit{patterns}(\sigma' h) \cup \{E''(M)\}$. Since the set of terms, third component of a semantic configuration, increases when the configuration is reduced, $E(M) \in \mathcal{M}'$, $E''(M) \in S_0'$, so $\mathit{patterns}(\sigma'(h \Rightarrow \mathsf{att}(\rho(M)))) \subseteq \mathit{sub}(S_0')$.

Similarly, $\mathit{patterns}(\sigma'(h \Rightarrow \mathsf{com}(c[\,]))) = \mathit{patterns}(\sigma' h) \cup \{c[\,]\}$. Since $c \in \mathcal{M}'$, $c[\,] \in S_0'$, so $\mathit{patterns}(\sigma'(h \Rightarrow \mathsf{com}(c[\,]))) \subseteq \mathit{sub}(S_0')$.

Then we have for all $R \in [\![\overline{c}\langle M\rangle.P]\!]\rho h$, $\mathrm{PatProp}(R)$.

- Case $c(x).P$: We have $E, C[E(c(x).P)], \mathcal{M} \rightarrow^* E', (\nu a_1 : p_1) \ldots (\nu a_n : p_n)0, \mathcal{M}'$. We have $E(c(x).P) = c(x).E(P)$. At some point, $E(c(x).P)$ must be reduced in the reduction $E, C[E(c(x).P)], \mathcal{M} \rightarrow^* E', (\nu a_1 : p_1) \ldots (\nu a_n : p_n)0, \mathcal{M}'$, since otherwise $E(c(x).P)$ would be present in the final process. So

$$\begin{aligned}
&E, C[E(c(x).P)], \mathcal{M} \rightarrow^* E_1', C'[c(x).E(P) \mid \overline{c}\langle M\rangle.Q], \mathcal{M}_1\\
&\quad \rightarrow E_1'[x \mapsto M], C'[E(P)\{M/x\} \mid Q], \mathcal{M}_1 \cup \{M, c\}\\
&\quad = E_1'[x \mapsto M], C''[E(P)\{M/x\}], \mathcal{M}_1 \cup \{M, c\}\\
&\quad \rightarrow^* E', (\nu a_1 : p_1) \ldots (\nu a_n : p_n)0, \mathcal{M}'
\end{aligned}$$

with $(E_1'[x \mapsto M])(P) = E(P)\{M/x\}$. Let $\rho' = \rho[x \mapsto x']$ and $h' = h \wedge \mathsf{att}(x')$. By the reduction trace above, $E'(x) = M \in \mathcal{M}'$, so $E''(x) = E_n(M) \in S_0'$. Let us define $\sigma'(x') = E_n(M)$. Then $E''_{|\mathit{dom}(\rho')} = \sigma' \circ \rho'$ and $\mathit{patterns}(\sigma' h') = \mathit{patterns}(\sigma' h) \cup \{E_n(M)\} \subseteq \mathit{sub}(S_0')$. By induction hypothesis, for all $R \in [\![P]\!]\rho' h'$, $\mathrm{PatProp}(R)$.

Moreover, $\mathit{patterns}(\sigma'(h \Rightarrow \mathsf{com}(c[\,]))) = \mathit{patterns}(\sigma' h) \cup \{c[\,]\}$. Since $c \in \mathcal{M}'$, $c[\,] \in S_0'$, so $\mathit{patterns}(\sigma'(h \Rightarrow \mathsf{com}(c[\,]))) \subseteq \mathit{sub}(S_0')$.

Then for all $R \in [\![c(x).P]\!]\rho h$, $\mathrm{PatProp}(R)$.

- Case *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* 0: We have $E, C[P_1], \mathcal{M} \rightarrow^* E', (\nu a_1{:}p_1) \ldots (\nu a_n{:}p_n)0, \mathcal{M}'$ with $P_1 = E(\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ 0) = \mathit{let}\ x = g(E(M_1), \ldots, E(M_n))\ \mathit{in}\ E(P)\ \mathit{else}\ 0$. At some point, P_1 must be reduced in the reduction $E, C[P_1], \mathcal{M} \rightarrow^* E', (\nu a_1 :$

$p_1)\dots(\nu a_n : p_n)0, \mathcal{M}'$, since otherwise P_1 would be present in the final process. So

$$\begin{aligned}&E, C[P_1], \mathcal{M} \to^* E'_1, C'[P_1], \mathcal{M}_1\\&\quad\to E'_1[x \mapsto M'], C'[E(P)\{M'/x\}], \mathcal{M}_1 \cup \{E(M_1), \dots, E(M_n), M'\}\\&\quad\to^* E', (\nu a_1 : p_1)\dots(\nu a_n : p_n)0, \mathcal{M}'\end{aligned}$$

with $(E'_1[x \mapsto M'])(P) = E(P)\{M'/x\}$. All destructors considered in tagged protocols are defined by a single rule. Let $g(N_1, \dots, N_n) \to N$ be this rule for g. Let σ_u be the most general unifier of the pairs $\rho(M_i), N_i$ for i in $1, \dots, n$. Since $g(E(M_1), \dots, E(M_n))$ reduces, $E(M_1), \dots, E(M_n)$ is an instance of $N_1, \dots, N_n$. Then $\sigma'\rho(M_i) = E''(M_i) = E_n(E(M_i))$ is an instance of N_i. We can extend σ', such that it becomes a unifier of $\rho(M_i)$ and N_i, since the variables of N_i are not in $dom(\sigma')$. Then there exists σ'' such that $\sigma' = \sigma''\sigma_u$.

Let $\rho' = \sigma_u\rho[x \mapsto \sigma_u N]$ and $h' = \sigma_u h$. We have $E''_{|dom(\rho)} = \sigma' \circ \rho = \sigma'' \circ \rho'_{|dom(\rho)}$ and $E''(x) = E_n(E'_1(x)) = \sigma' N = \sigma''\sigma_u N$, so $E''_{|dom(\rho')} = \sigma'' \circ \rho'$ and $patterns(\sigma'' h') = patterns(\sigma' h) \subseteq sub(S'_0)$. By induction hypothesis, for all $R \in [\![P]\!]\rho' h'$, $\mathrm{PatProp}(R)$.

Moreover, let $R = h \wedge \mathsf{testunif}((\rho(M_1), \dots, \rho(M_n)), (N'_1, \dots, N'_n)) \Rightarrow \mathsf{bad}$, where $N'_1, \dots, N'_n$ are obtained from $N_1, \dots, N_n$ by replacing variables with elements of $EVar$. As we have seen above, for all i in $1, \dots, n$, $\sigma'\rho(M_i) = \sigma' N_i$, so $\rho(M_i)$ and N_i unify. Their most general unifier is σ_u as defined above. Then $\mathrm{TestUnifRef}(R) = \sigma_u h \Rightarrow \mathsf{bad}$, so $patterns(\sigma''\mathrm{TestUnifRef}(R)) = patterns(\sigma' h) \subseteq sub(S'_0)$.

Then we have for all $R \in [\![let\ x = g(M_1, \dots, M_n)\ in\ P\ else\ 0]\!]\rho h$, $\mathrm{PatProp}(R)$.

□

Lemma 13 *Let $\rho = \{a \mapsto a[\,] \mid a \in fn(P_0)\}$. For all clauses R in $[\![P_0]\!]\rho\emptyset\emptyset$, the clause $\mathrm{Rref}(R)$ is defined and there exists a substitution σ such that $patterns(\sigma\mathrm{Rref}(R)) \subseteq sub(S_0)$.*

Proof By Lemma 12 applied to the instrumented process P'_0 corresponding to P_0, there exists σ'' such that $patterns(\sigma''\mathrm{TestUnifRef}(R)) \subseteq sub(S'_0)$.

For clauses not containing $\mathsf{testunif}$, $\mathrm{Rref}_1(R) = \mathrm{VarSubst}(R)$ is more general than $\mathrm{TestUnifRef}(R) = R$, so a fortiori there exists σ such that $patterns(\sigma\mathrm{Rref}_1(R)) \subseteq sub(S'_0)$.

For clauses containing $\mathsf{testunif}$, such that $\mathrm{VarSubst}(R) = H \wedge \mathsf{testunif}(p_1, p_2) \Rightarrow \mathsf{bad}$, $\mathrm{Rref}_1(R)$ is also more general than $\mathrm{TestUnifRef}(R)$. Indeed, let σ' be a substitution of variables to elements of $Secr \cup EVar$, such that $\sigma'\mathrm{VarSubst}(R) = R$. Then $\mathrm{Rref}_1(R) = \sigma'_u H \Rightarrow \mathsf{bad}$ where σ'_u is the most general unifier of p_1 and p_2, and $\mathrm{TestUnifRef}(R) = \sigma_u\sigma' H \Rightarrow \mathsf{bad}$ where σ_u is the most general unifier of $\sigma' p_1$ and $\sigma' p_2$. We have $\sigma_u\sigma' p_1 = \sigma_u\sigma' p_2$, so σ'_u, most general unifier of p_1 and p_2, is more general than $\sigma_u\sigma'$, so $\mathrm{Rref}_1(R)$ is more general than $\mathrm{TestUnifRef}(R)$. Hence there exists σ such that $\mathit{patterns}(\sigma\mathrm{Rref}_1(R)) \subseteq \mathit{sub}(S'_0)$.

Let $\sigma'' = \mathrm{OneKey}(\sigma)$ be the substitution that maps variables x to $\mathrm{OneKey}(\sigma x)$. Then we obtain

$$\begin{aligned}\mathit{patterns}(\sigma''\mathrm{Rref}(R)) &= \mathit{patterns}(\sigma''\mathrm{OneKey}(\mathrm{Rref}_1(R)) \\ &= \mathrm{OneKey}(\mathit{patterns}(\sigma\mathrm{Rref}_1(R))) \\ &\subseteq \mathit{sub}(\mathrm{OneKey}(S'_0)) = \mathit{sub}(S_0).\end{aligned}$$

which completes the proof. □

Let

$$\begin{aligned}\mathit{tagGen} = \{&f((t_i, x_1, \dots, x_n), x'_2, \dots, x'_{n'}) \mid f \in \{\mathit{sencrypt}, \mathit{sencrypt}_{\mathit{prob}}, \\ &\quad \mathit{pencrypt}_{\mathit{prob}}, \mathit{sign}, \mathit{nmrsign}, H, \mathit{mac}\}\} \\ &\cup \{a[x_1, \dots, x_n] \mid a \text{ name function symbol}\} \\ &\cup \{\mathit{pk}(x), \mathit{host}(x)\} \cup \{c \mid c \text{ atomic constant}\}\end{aligned}$$

Elements of *Secr* do not occur in strict subterms of patterns in *tagGen* (but they are included in *tagGen* as atomic constants).

Lemma 14 *If $\mathit{patterns}(R)$ are fully tagged, then $\mathit{patterns}(\mathrm{VarSubst}(R))$ are fully tagged.*

Proof All non-variable, non-data, non-element of *EVar* subterms of patterns of R are instances of elements of *tagGen*. Either such a subterm is in *Secr*, in which case the corresponding subterm in $\mathrm{VarSubst}(R)$ is a variable, or it is an instance of an element of *tagGen* not containing elements of *Secr* (by Condition W1). Then after replacing elements of *Secr* with variables, it is still an instance of the same element of *tagGen*. So the patterns of $\mathrm{VarSubst}(R)$ are fully tagged. □

Lemma 15 *Assume that the patterns of the image of ρ and of h are fully tagged. Assume that P is an instrumented process, subprocess of P'_0. For all $R \in [\![P]\!]\rho h$, $\mathit{patterns}(\mathrm{Rref}_1(R))$ are fully tagged.*

Proof First, by Condition C3, the terms of P in which names are replaced with variables (then these terms are patterns) are fully tagged. We use the form of destructor applications coming from Condition C4.

We reason by induction on P.

- Case 0: Obvious.

- Case $P \mid Q$: Obvious by induction hypothesis.

- Case $!^{i}P$: Obvious by induction hypothesis (the variable i' is fully tagged).

- Case $(\nu a : p)P$: Obvious by induction hypothesis ($a[x_1, \dots, x_n]$ is fully tagged).

- Case $\overline{c}\langle M\rangle.P$: The generated clauses are

$$[\![\overline{c}\langle M\rangle.P]\!]\rho h = [\![P]\!]\rho h \cup \{h \Rightarrow \mathsf{att}(\rho(M)), h \Rightarrow \mathsf{com}(c[\,])\}$$

  By induction hypothesis, for all $R \in [\![P]\!]\rho h$, $\mathit{patterns}(R)$ are fully tagged. Moreover, $\mathit{patterns}(h \Rightarrow \mathsf{att}(\rho(M))) = \mathit{patterns}(h) \cup \{\rho(M)\}$. By Lemma 8, $\rho(M)$ is fully tagged, since M with names replaced with variables is fully tagged and the image of ρ consists of fully tagged patterns. The patterns of $h \Rightarrow \mathsf{com}(c[\,])$ are also fully tagged since $c[\,]$ is fully tagged. By Lemma 14, for these clauses, $\mathit{patterns}(\mathrm{Rref}_1(R)) = \mathit{patterns}(\mathrm{VarSubst}(R))$ are fully tagged.

- Case $c(x).P$: Let $\rho' = \rho[x \mapsto x']$ and $h' = h \wedge \mathsf{att}(x')$. The generated clauses are:

$$[\![c(x).P]\!]\rho h = [\![P]\!]\rho' h' \cup \{h \Rightarrow \mathsf{com}(c[\,])\}$$

  By induction hypothesis, for all $R \in [\![P]\!]\rho' h'$, $\mathit{patterns}(\mathrm{Rref}_1(R))$ are fully tagged. The patterns of $h \Rightarrow \mathsf{com}(c[\,])$ are also fully tagged since $c[\,]$ is fully tagged. By Lemma 14, for this clause, $\mathit{patterns}(\mathrm{Rref}_1(R)) = \mathit{patterns}(\mathrm{VarSubst}(R))$ are fully tagged.

- Case $P' = \mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\,\mathit{let}\ y = 1\mathit{th}_m(x)\ \mathit{in}\,\mathit{if}\ y = t\ \mathit{then}\ P$, with $g \in \{\mathit{sdecrypt}, \mathit{sdecrypt}_{\mathit{prob}}, \mathit{pdecrypt}_{\mathit{prob}}, \mathit{checksign}, \mathit{getmess}\}$: Assume $g(N_1, \dots, N_n) \to z$. Let $N'_1 = N_1\{(t, x_1, \dots, x_m)/z\}$ and $N'_i = N_i$ for $i > 1$. The patterns N'_i for i in $1, \dots, n$ are fully tagged (this can be proved for each of the allowed destructors g). Let $N''_1, \dots, N''_n$ be obtained from $N'_1, \dots, N'_n$ by replacing variables with distinct elements of EVar. The generated clauses are

$$\begin{aligned}[\![P']\!]\rho h = {} & [\![P]\!]((\sigma\rho)[x \mapsto (t, \sigma x_1, \dots, \sigma x_m), y \mapsto t])(\sigma h) \\ & \cup \{h \wedge \mathsf{testunif}((\rho(M_1), \dots, \rho(M_n)), (N''_1, \dots, N''_n)) \Rightarrow \mathsf{bad}\}\end{aligned}$$

  where σ is the most general unifier of $\rho(M_i)$ and N'_i for i in $1, \dots, n$. (We use here the grouping of destructors: the destructor application of g as well as the verification of the tag are handled in one shot.) We know that, for i in $1, \dots, n$, $\rho(M_i)$ is fully tagged, as in the output case. Then σ is the most general unifier of fully tagged patterns, so by Lemma 10, for all x', $\sigma x'$ is fully tagged. So, by Lemma 8, all patterns in the image of $\rho' = (\sigma\rho)[x \mapsto (t, \sigma x_1, \ldots, \sigma x_m), y \mapsto t]$ and in $h' = \sigma h$ are fully tagged. By induction hypothesis, all clauses in $[\![P]\!]\rho' h'$ have fully tagged patterns.

Let $R = h \wedge \mathsf{testunif}((\rho(M_1), \ldots, \rho(M_n)), (N''_1, \ldots, N''_n)) \Rightarrow \mathsf{bad}$. Then $\mathrm{Rref}_1(R) = \sigma' \mathrm{VarSubst}(h) \Rightarrow \mathsf{bad}$, where σ' is the most general unifier of $\mathrm{VarSubst}(\rho(M_i))$ and $\mathrm{VarSubst}(N'_i) = N'_i$ for i in $1, \ldots, n$. By Lemma 14, the patterns of $\mathrm{VarSubst}(h)$ and $\mathrm{VarSubst}(\rho(M_i))$ are fully tagged, so σ' is the most general unifier of fully tagged patterns. Then by Lemmas 10 and 8, the patterns of $\mathrm{Rref}_1(R)$ are fully tagged.

- Case $P' = \mathit{let}\ x = \mathit{nmrchecksign}(M_1, M_2, (t, M'_1, \ldots, M'_m))\ \mathit{in}\ P$: We have $\mathit{nmrchecksign}(\mathit{sign}(y, z), \mathit{pk}(z), y) \to \mathit{true}$. Let e and e' be elements of EVar. The generated clauses are

$$\begin{aligned}[\![P']\!]\rho h = {} & [\![P]\!]((\sigma\rho)[x \mapsto \mathit{true}])(\sigma h) \\ & \cup \{h \wedge \mathsf{testunif}((\rho(M_1), \rho(M_2), \rho((t, M'_1, \ldots, M'_m))), \\ & \qquad (\mathit{nmrsign}(\mathsf{e}, \mathsf{e}'), \mathit{pk}(\mathsf{e}'), \mathsf{e})) \Rightarrow \mathsf{bad}\}\end{aligned}$$

  where the substitution σ is the most general unifier of $(\rho(M_1), \rho(M_2), \rho((t, M'_1, \ldots, M'_m)))$ and $(\mathit{nmrsign}(y, z), \mathit{pk}(z), y)$. Then σ is also the most general unifier of the patterns $(\rho(M_1), \rho(M_2), \rho((t, M'_1, \ldots, M'_m)))$ and $(\mathit{nmrsign}((t, x_1, \ldots, x_m), z), \mathit{pk}(z), (t, x_1, \ldots, x_m))$, hence σ is the most general unifier of fully tagged patterns. We can then conclude as in the previous case.

- Case $P' = \mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P$, with $g \in \{\mathit{eq}, \mathit{getkey}\}$ or g is a projection: Assume $g(N_1, \ldots, N_n) \to N$. Let $N'_1, \ldots, N'_n$ be obtained from $N_1, \ldots, N_n$ by replacing variables with distinct elements of EVar. The generated clauses are

$$\begin{aligned}[\![P']\!]\rho h = {} & [\![P]\!]((\sigma\rho)[x \mapsto \sigma N])(\sigma h) \\ & \cup \{h \wedge \mathsf{testunif}((\rho(M_1), \ldots, \rho(M_n)), (N'_1, \ldots, N'_n)) \Rightarrow \mathsf{bad}\}\end{aligned}$$

  where σ is the most general unifier of $\rho(M_i)$ and N_i for i in $1, \ldots, n$.

  For $i = 1, \ldots, n$, $\rho(M_i)$ is fully tagged, as in the output case. For $i = 1, \ldots, n$, N_i is either a variable when $g = \mathit{eq}$, a term $\mathit{host}(x)$ when $g = \mathit{getkey}$, or a term $(x_1, \ldots, x_n)$ when g is a projection, so in all cases N_i is fully tagged. Hence σ is a most general unifier of fully tagged patterns. We can then conclude as in the case $g \in \{\mathit{sdecrypt}, \mathit{sdecrypt}_{\mathit{prob}}, \mathit{pdecrypt}_{\mathit{prob}}, \mathit{checksign}, \mathit{getmess}\}$.

□

Lemma 16 *Let* $\rho = \{a \mapsto a[\,] \mid a \in \mathit{fn}(P_0)\}$. *For all* $R \in [\![P_0]\!]\rho\emptyset\emptyset$, *the patterns of* $\mathrm{Rref}(R)$ *are fully tagged.*

Proof By applying Lemma 15 to the instrumented process P'_0 corresponding to P_0, the patterns of $\mathrm{Rref}_1(R)$ are fully tagged. Then the patterns of $\mathrm{Rref}(R) = \mathrm{OneKey}(\mathrm{Rref}_1(R))$ are also fully tagged. □

Lemma 17 *Let $f \in \{sencrypt, sencrypt_{prob}, pencrypt_{prob}, sign, nmrsign, H, mac\}$ and c be a constant tag. Assume that $E, P, \mathcal{M} \to E', P', \mathcal{M}'$.*

Assume that all terms of the form $f(\ldots)$ in $E, P, \mathcal{M}$ have a first argument of the form $(c, \ldots)$ for some constant tag c. Then the same is true for $E', P', \mathcal{M}'$.

If a term of the form $f((c, \ldots), \ldots)$ does not occur in $E, P, \mathcal{M}$, then it does not occur in $E', P', \mathcal{M}'$.

If a term of the form $f((c, \ldots), \ldots)$ occurs only once in $E, P, \mathcal{M}$ and this occurrence is in P, then either such a term occurs only once in $E', P', \mathcal{M}'$ and this occurrence is in P', or all occurrences of such a term in $E', P', \mathcal{M}'$ are the same closed term.

If the only term of the form $f((c, \ldots), \ldots)$ in $E, P, \mathcal{M}$ is a closed term M (perhaps occurring several times), then the only term of the form $f((c, \ldots), \ldots)$ in $E', P', \mathcal{M}'$ is M.

Proof By induction on the derivation of $E, P, \mathcal{M} \to E', P', \mathcal{M}'$.

- Case (Red I/O'): We have

  $$E, \overline{c}\langle M\rangle.Q_1 \mid c(x).P_1, \mathcal{M} \;\to\; E[x \mapsto M], Q_1 \mid P_1\{M/x\}, \mathcal{M} \cup \{M, c\}$$

  All terms of the form $f(\ldots)$ in $E, P, \mathcal{M}$ have a first argument of the form $(c, \ldots)$ for some constant tag c. Then this remains true after substitution.

  If a term $f((c, \ldots), \ldots)$ does not occur in M, the occurrences of these terms are unchanged (if they are not closed, the terms may be instantiated, but there is still one occurrence in P; if they are closed, the terms themselves are unchanged).

  If such a term occurs in M, then since M is closed, we are in third case (or in the second one, which reduces to the third one here). Let N be the subterm of M of the form $f((c, \ldots), \ldots)$. The only term of the form $f((c, \ldots), \ldots)$ in $E', P', \mathcal{M}'$ is N. (Note that no term in $E, P, \mathcal{M}$ is of the form $f(y, \ldots)$, so terms $f((c, \ldots), \ldots)$ cannot be created by a substitution. Either they come from the term M substituted to x, or they already existed in the substituted process. In both cases, they are equal to N.)

- Case (Red Destr'): We have

  $$\begin{aligned} &E, let\ x = g(M_1, \ldots, M_n)\ in\ P_1\ else\ 0, \mathcal{M} \\ &\quad \to\; E[x \mapsto M'], P_1\{M'/x\}, \mathcal{M} \cup \{M_1, \ldots, M_n, M'\} \end{aligned}$$

if $g(M_1, \ldots, M_n) \rightarrow M'$. This case is similar to (Red I/O'). Just note that for tagged protocols, the only considered destructors always return a subterm of their arguments or a constant *true*, so M' is a subterm of M_1, ..., or M_n or $M' = \mathit{true}$. So if the considered form occurs in M' then it occurs in $M_1, \ldots, M_n$, and it is the same closed term N.

- Case (Red Repl'): Obvious, the occurrences of the considered terms are unchanged.
- Case (Red Par'): We have

  $$E, P_1, \mathcal{M} \rightarrow E', Q_1, \mathcal{M}' \Rightarrow E, P_1 \mid R_1, \mathcal{M} \rightarrow E', Q_1 \mid R_1, \mathcal{M}'$$

  The first point is obvious by induction hypothesis.

  If a term of the form $f((c, \ldots), \ldots)$ does not occur in $E, P_1 \mid R_1, \mathcal{M}$, it does not occur in $E, P_1, \mathcal{M}$ so it does not occur in $E', Q_1, \mathcal{M}'$ by induction hypothesis. Therefore, it does not occur in $E', Q_1 \mid R_1, \mathcal{M}'$.

  Assume that such a term occurs once in $P_1 \mid R_1$, in $E, P_1 \mid R_1, \mathcal{M}$. If it does not occur in P_1, so not in $E', Q_1, \mathcal{M}'$ by induction hypothesis, and it occurs once in $E', Q_1 \mid R_1, \mathcal{M}'$, in R_1. If it occurs in P_1, then, by induction hypothesis, either it occurs once in $E', Q_1, \mathcal{M}'$, in Q_1, so it occurs once in $E', Q_1 \mid R_1, \mathcal{M}'$, in Q_1, or it occurs as a closed term M in $E', Q_1, \mathcal{M}'$, then it occurs as M in $E', Q_1 \mid R_1, \mathcal{M}'$.

  Assume that such a term occurs as a closed term M in $E, P_1 \mid R_1, \mathcal{M}$, then it occurs as M in $E, P_1, \mathcal{M}$, so also in $E', Q_1, \mathcal{M}'$ by induction hypothesis, and in $E', Q_1 \mid R_1, \mathcal{M}'$.

- Case (Red Res'): By induction hypothesis, similar to (Red Par').
- Case (Red $\equiv$'): Obvious by induction hypothesis, since the property is preserved by $\equiv$.

$\square$

Lemma 18 *Each pattern in tagGen has at most one instance in sub*(S_0).

Proof For $pk(x)$ and $host(x)$, this comes from the fact that OneKey maps all elements of *Keys* to a single element k_0.

For atomic constants, this is obvious (since they have only one instance: themselves).

The only instance of $a[x_1, \ldots, x_n]$ in $sub(S'_0)$ is OneKey$(E_n(a))$, since different restrictions (νa) are associated to different function symbols $a[\ldots]$.

For patterns $f((c, \ldots), \ldots)$ with $f \in \{sencrypt, sencrypt_{prob}, pencrypt_{prob}, sign, nmrsign, H, mac\}$ and c a constant tag, by Condition C3 there is only one occurrence of a term $f((c,\ldots),\ldots)$ in $\{\}, P'_0, fn(P_0)$. By Lemma 17, all occurrences of such a term in $E', (\nu a_1 : p_1) \ldots (\nu a_n : p_n)0, \mathcal{M}'$ are the same closed term N (since all terms are closed in $E', (\nu a_1 : p_1) \ldots (\nu a_n : p_n)0, \mathcal{M}'$). So all occurrences of a pattern $f((c,\ldots),\ldots)$ in subterms of $S_0 = \mathrm{OneKey}(E_n(\mathcal{M}') \cup \{b[\mathrm{Id}_0]\} \cup \{a[\,] \mid a \in fn(P_0)\})$ are equal to $\mathrm{OneKey}(E_n(N))$, which is the expected result. (Applying the substitution E_n to $\mathcal{M}'$ cannot create new patterns $f((c,\ldots),\ldots)$, because the image of E_n consists of patterns $a[\ldots]$, not of the form $(c,\ldots)$. Also note that when we add other terms $f((c,\ldots),\ldots)$ to S_0 because there is no instance of $f(x,y)$ in S_0, we only add one such term, so the result is also true.) $\square$

We can now prove the proposition.

Proof The clauses (Rcom) are in $\mathcal{R}_{\mathrm{Prot1}}$. The clause (Rc) is in $\mathcal{R}_{\mathrm{Prot0}}$, as well as (Rf), (Init), and (Rn) given the value of S_0. (The patterns of (Init) and (Rn) are fully tagged. For (Rc), the strict subterms of $f(x_1,\ldots,x_n)$ are variables, so fully tagged, $f(x_1,\ldots,x_n)$ is more general than any pattern with the same root symbol, and $x_1,\ldots,x_n$ do not occur in strict subterms of patterns of the clause other than $f(x_1,\ldots,x_n)$.)

The clauses (Rg) are:

$$\mathsf{att}(\mathit{sencrypt}(x,y)) \wedge \mathsf{att}(y) \Rightarrow \mathsf{att}(x) \qquad (\mathit{sdecrypt})$$

$$\mathsf{att}(\mathit{sencrypt}_{\mathit{prob}}(x,y,z)) \wedge \mathsf{att}(y) \Rightarrow \mathsf{att}(x) \qquad (\mathit{sdecrypt}_{\mathit{prob}})$$

$$\mathsf{att}(\mathit{pencrypt}_{\mathit{prob}}(x,\mathit{pk}(y),z)) \wedge \mathsf{att}(y) \Rightarrow \mathsf{att}(x) \qquad (\mathit{pdecrypt}_{\mathit{prob}})$$

$$\mathsf{att}(\mathit{sign}(x,y)) \wedge \mathsf{att}(\mathit{pk}(y)) \Rightarrow \mathsf{att}(x) \qquad (\mathit{checksign})$$

$$\mathsf{att}(\mathit{sign}(x,y)) \Rightarrow \mathsf{att}(x) \qquad (\mathit{getmess})$$

$$\mathsf{att}(\mathit{nmrsign}(x,y)) \wedge \mathsf{att}(\mathit{pk}(y)) \wedge \mathsf{att}(x) \Rightarrow \mathsf{att}(\mathit{true}) \qquad (\mathit{nmrchecksign})$$

$$\mathsf{att}(x) \wedge \mathsf{att}(x) \Rightarrow \mathsf{att}(x) \qquad (\mathit{eq})$$

For these clauses, $\mathrm{Rref}(R) = R$. The cases *sdecrypt*, $\mathit{sdecrypt}_{\mathit{prob}}$, and *getmess* are similar to constructor clauses. For $\mathit{pdecrypt}_{\mathit{prob}}$, the strict subterms of $\mathit{pencrypt}_{\mathit{prob}}(x,\mathit{pk}(y),z)$ are fully tagged, since *tagGen* contains $\mathit{pk}(y)$; $\mathit{pencrypt}_{\mathit{prob}}(x,\mathit{pk}(y),z)$ is more general than all patterns of $\mathit{sub}(S_0)$ with the same root symbol (by definition of S_0, using Condition C6), and x,y do not occur in strict subterms of patterns of the clause other than $\mathit{pencrypt}_{\mathit{prob}}(x,\mathit{pk}(y),z)$. For *checksign*, the strict subterms of $\mathit{sign}(x,y)$ are variables, so fully tagged; $\mathit{sign}(x,y)$ is more general than any pattern with the same root symbol; in all instances of $\mathit{sign}(x,y)$ in $\mathit{sub}(S_0)$, the only instance of y is k_0 (since OneKey maps all elements of *Keys* to k_0) and x does not occur in strict subterms of patterns of the clause other than $\mathit{sign}(x,y)$. The case *nmrchecksign* is similar to *checksign*, and *eq* is easy. (In fact, the clauses for *checksign* and *nmrchecksign* will be removed by subsumption and the clause for *eq* is a tautology.)

The clauses (Rt) are:

$R = \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \wedge \mathsf{testunif}((x_1, \ldots, x_n), EVar((N_1, \ldots, N_n))) \Rightarrow \mathsf{bad}$

where $g(N_1, \ldots, N_n) \to N$ is in $\mathrm{def}(g)$. Then $\mathrm{Rref}(R)$ is

$$\mathrm{Rref}(R) = \mathsf{att}(N_1) \wedge \ldots \wedge \mathsf{att}(N_n) \Rightarrow \mathsf{bad}$$

We can show as for clauses (Rg) that $\mathrm{Rref}(R)$ satisfies the required conditions, so $R \in \mathcal{R}_{\mathrm{Prot0}}$.

Let $\rho = \{a \mapsto a[\,] \mid a \in fn(P_0)\}$. By Lemmas 13 and 16, the clauses of $[\![P_0]\!]\rho\emptyset\emptyset$ are in $\mathcal{R}_{\mathrm{Prot0}}$. Then we have Condition W3.

Different patterns in *tagGen* do not unify. By Lemma 18, each pattern in *tagGen* has at most one instance in $sub(S_0)$. So we have Condition W2.

Condition W1 is obvious by definition of *tagGen*. □

C.2 Tagged Implies Weakly Tagged (Proof Sketch)

Proposition 9 *If P_0 is a tagged protocol, then $\mathcal{R}_{P_0}$ is a weakly tagged program.*

Proof sketch Let

$$\begin{aligned} tagGen = \{ & f((t_i, x_1, \ldots, x_n), x'_2, \ldots, x'_{n'}) \mid f \in \{sencrypt, sencrypt_{prob}, \\ & \quad pencrypt_{prob}, sign, nmrsign, H, mac\}\} \\ & \cup \{a[x_1, \ldots, x_n] \mid a \text{ name function symbol}\} \\ & \cup \{pk(x), host(x)\} \cup \{c \mid c \text{ atomic constant}\} \end{aligned}$$

Elements of *Secr* do not occur strict subterms of patterns in *tagGen* (but they are included in *tagGen* as atomic constants).

Let S'_0 be the set of patterns corresponding to closed terms manipulated in the execution of Condition C5 to which we add the patterns $b[\mathrm{Id}_0]$ and $a[\,]$ for $a \in fn(P_0)$. Let c, c', c'' be constants. We add the following patterns to the set S'_0: $sencrypt((c, c'), c'')$, $sencrypt_{prob}((c, c'), c'', c')$, $pencrypt_{prob}((c, c'), pk(k_0), c')$, $sign((c, c'), k_0)$, $nmrsign((c, c'), k_0)$, $host(k_0)$ if S'_0 contains no instance of the terms $sencrypt(x, y)$, $sencrypt_{prob}(x, y, z)$, $pencrypt_{prob}(x, y, z)$, $sign(x, y)$, $nmrsign(x, y)$, $host(x)$, respectively. Let $S_0 = \mathrm{OneKey}(S'_0)$. Then S_0 is a finite set of closed patterns.

The clauses (Rcom) are in $\mathcal{R}_{\mathrm{Prot1}}$. The clause (Rc) is in $\mathcal{R}_{\mathrm{Prot0}}$, as well as (Rf), (Init) and (Rn) given the value of S_0. (The patterns of (Init) and (Rn) are fully tagged. For (Rc), the strict subterms of $f(x_1, \ldots, x_n)$ are variables, so fully tagged, $f(x_1, \ldots, x_n)$ is more general than any pattern with the same root symbol, and $x_1, \ldots, x_n$ do not occur in strict subterms of patterns of the clause other than $f(x_1, \ldots, x_n)$.)

The clauses (Rg) are:

$$\begin{aligned}
&\mathsf{att}(\mathit{sencrypt}(x,y)) \wedge \mathsf{att}(y) \Rightarrow \mathsf{att}(x) && (\mathit{sdecrypt})\\
&\mathsf{att}(\mathit{sencrypt}_{\mathit{prob}}(x,y,z)) \wedge \mathsf{att}(y) \Rightarrow \mathsf{att}(x) && (\mathit{sdecrypt}_{\mathit{prob}})\\
&\mathsf{att}(\mathit{pencrypt}_{\mathit{prob}}(x,\mathit{pk}(y),z)) \wedge \mathsf{att}(y) \Rightarrow \mathsf{att}(x) && (\mathit{pdecrypt}_{\mathit{prob}})\\
&\mathsf{att}(\mathit{sign}(x,y)) \wedge \mathsf{att}(\mathit{pk}(y)) \Rightarrow \mathsf{att}(x) && (\mathit{checksign})\\
&\mathsf{att}(\mathit{sign}(x,y)) \Rightarrow \mathsf{att}(x) && (\mathit{getmess})\\
&\mathsf{att}(\mathit{nmrsign}(x,y)) \wedge \mathsf{att}(\mathit{pk}(y)) \wedge \mathsf{att}(x) \Rightarrow \mathsf{att}(\mathit{true}) && (\mathit{nmrchecksign})\\
&\mathsf{att}(x) \wedge \mathsf{att}(x) \Rightarrow \mathsf{att}(x) && (\mathit{eq})
\end{aligned}$$

For these clauses, $\mathrm{Rref}(R) = R$. The cases *sdecrypt*, $\mathit{sdecrypt}_{\mathit{prob}}$, and *getmess* are similar to constructor clauses. For $\mathit{pdecrypt}_{\mathit{prob}}$, the strict subterms of $\mathit{pencrypt}_{\mathit{prob}}(x, \mathit{pk}(y), z)$ are fully tagged, since *tagGen* contains $\mathit{pk}(y)$; $\mathit{pencrypt}_{\mathit{prob}}(x, \mathit{pk}(y), z)$ is more general than all patterns of $\mathit{sub}(S_0)$ with the same root symbol (by definition of S_0, using Condition C6), and x, y do not occur in strict subterms of patterns of the clause other than $\mathit{pencrypt}_{\mathit{prob}}(x, \mathit{pk}(y), z)$. For *checksign*, the strict subterms of $\mathit{sign}(x, y)$ are variables, so fully tagged; $\mathit{sign}(x, y)$ is more general than any pattern with the same root symbol; in all instances of $\mathit{sign}(x, y)$ in $\mathit{sub}(S_0)$, the only instance of y is k_0 (since OneKey maps all elements of *Keys* to k_0) and x does not occur in strict subterms of patterns of the clause other than $\mathit{sign}(x, y)$. The case *nmrchecksign* is similar to *checksign*, and *eq* is easy. (In fact, the clauses for *checksign* and *nmrchecksign* will be removed by subsumption and the clause for *eq* is a tautology.)

The clauses (Rt) are:

$$R = \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \wedge \mathsf{testunif}((x_1, \ldots, x_n), \mathit{EVar}((N_1, \ldots, N_n))) \Rightarrow \mathsf{bad}$$

where $g(N_1, \ldots, N_n) \to N$ is in $\mathrm{def}(g)$. Then $\mathrm{Rref}(R)$ is

$$\mathrm{Rref}(R) = \mathsf{att}(N_1) \wedge \ldots \wedge \mathsf{att}(N_n) \Rightarrow \mathsf{bad}$$

We can show as for clauses (Rg) that $\mathrm{Rref}(R)$ satisfies the required conditions, so $R \in \mathcal{R}_{\mathrm{Prot0}}$.

Let $\rho = \{a \mapsto a[\,] \mid a \in \mathit{fn}(P_0)\}$. Let R be a clause of $[\![P_0]\!]\rho\emptyset\emptyset$. By Condition C5, S_0 contains all patterns corresponding to terms in an execution that reaches all program points of P_0, and all destructor applications succeed. Then we can show that there exists σ such that $\mathit{patterns}(\sigma\mathrm{Rref}(R)) \subseteq \mathit{sub}(S_0)$. By Condition C3, the created terms are tagged, and by Condition C4, the tags are checked. So we can show that all patterns of $\mathrm{Rref}(R)$ are fully tagged. Hence $R \in \mathcal{R}_{\mathrm{Prot0}}$. So we have Condition W3.

Different patterns in *tagGen* do not unify. Moreover, each pattern in *tagGen* has at most one instance in $\mathit{sub}(S_0)$. For $\mathit{pk}(x)$ and $\mathit{host}(x)$, this comes from fact that OneKey maps all elements of *Keys* to k_0. For atomic constants, this is obvious (their only instance is themselves). For other patterns, this comes from the fact that the execution of Condition C5 executes each program point at most once, and that patterns created at different programs points are associated with different symbols (f, c) for $f((c, \ldots), \ldots)$ and a for $a[\ldots]$. (For $f((c, \ldots), \ldots)$, this comes from Condition C3. For $a[\ldots]$, this is because different restrictions use a different function symbol by construction of the clauses.) So we have Condition W2.

Condition W1 is obvious by definition of *tagGen*. □

C.3 Weakly Tagged Implies Termination

Let us define an ordering among patterns in $sub(S_0)$: $p_1 < p_2$ when p_1 is a strict subterm of p_2. We add a top element $\top$ larger than all patterns in $sub(S_0)$, and a bottom element $\bot$, smaller than all patterns in $sub(S_0)$. This is indeed a strict ordering, since $p_1 < p_2$ and $p_2 < p_1$ is impossible. This ordering is well-founded.

We extend this ordering to clauses of $\mathcal{R}_{\mathrm{Prot0}}$ as follows. We denote by $+$ the multiset union.

$$size(F_1 \wedge \ldots \wedge F_n \Rightarrow F) = size(F_1) + \ldots + size(F_n) + size(F)$$

$$size(pred(p_1, \ldots, p_n)) = size(p_1) + \ldots + size(p_n)$$

$$size(f(p_1, \ldots, p_n)) = size(p_1) + \ldots + size(p_n) \text{ if } f \text{ is a data constructor}$$
$$size(x) = \{\bot\}$$
$$size(p) = \{\sigma p\} \text{ such that } \sigma p \in sub(S_0) \text{ when } p \text{ is top-tagged}$$
$$size(p) = \{\top\} \text{ otherwise}$$

We say that $R < R'$ when $size(\mathrm{Rref}(R)) < size(\mathrm{Rref}(R'))$ in the multiset extension of the ordering defined in the previous paragraph.

Lemma 19 *If p is fully tagged, and $\sigma p \in sub(S_0)$ then $size(p)$ is a multiset of subterms of σp.*

Lemma 20 *If $R \in \mathcal{R}_{\mathrm{Prot0}}$, $\mathrm{Rref}_1(R)$ is left unchanged by unify, swap, elimEVar, instantiate, elimvar. A fortiori, $\mathrm{Rref}(R) = \mathrm{OneKey}(\mathrm{Rref}_1(R))$ is also left unchanged by these functions.*

Proof Let us substitute variables for elements of $Secr \cup EVar$ everywhere.

Let $R = H \wedge \mathsf{testunif}(p_1, p_2) \Rightarrow \mathsf{bad}$, $\mathrm{Rref}_1(R) = \sigma_u H \Rightarrow \mathsf{bad}$ where σ_u is the most general unifier of p_1 and p_2 (recall that elements of $Secr \cup EVar$ have been substituted with variables).

The fact $\mathsf{testunif}(p_1, p_2)$ is transformed into $\mathsf{testunif}(p'_1, p'_2)$ by *unify*. As shown in the proof of Lemma 7 for *unify*, for all σ, $\sigma p_1 = \sigma p_2$ if and only if $\sigma p'_1 = \sigma p'_2$ (recall that elements of $Secr \cup EVar$ have been substituted with variables). Then σ_u is also the most general unifier of p'_1 and p'_2. So, with $R' = H \wedge \mathsf{testunif}(p'_1, p'_2) \Rightarrow \mathsf{bad}$, $\mathrm{Rref}_1(R') = \sigma_u H \Rightarrow \mathsf{bad} = \mathrm{Rref}_1(R)$.

The same reasoning applies to *swap* (using the proof of Lemma 7 for *swap*).

For *elimEVar*, $\mathsf{testunif}((\mathsf{e}, p'_1, \dots, p'_n), (p''_0, \dots, p''_n))$ with $\mathsf{e} \in EVar$ is transformed into $\mathsf{testunif}((p'_1, \dots, p'_n), (p''_1, \dots, p''_n))$. Since σ_u is a most general unifier of $(\mathsf{e}, p'_1, \dots, p'_n)$ and $(p''_0, \dots, p''_n)$, $\sigma_{u|\overline{\mathsf{e}}}$ is a most general unifier of $(p'_1, \dots, p'_n)$ and $(p''_1, \dots, p''_n)$. Hence $\mathrm{Rref}_1(R)$ is also unchanged by *elimEVar*.

For *instantiate*, a clause $R = H \wedge \mathsf{testunif}((\dots, x, \dots), (\dots, f(p_1, \dots, p_n), \dots)) \Rightarrow \mathsf{bad}$ becomes $R' = R\{f(x_1, \dots, x_n)/x\}$. The substitution σ_u is the most general unifier of $(\dots, x, \dots)$ and $(\dots, f(p_1, \dots, p_n), \dots)$, so $\sigma_u x = \sigma_u f(p_1, \dots, p_n)$. Then $\sigma_u x$ is of the form $f(p'_1, \dots, p'_n)$, so there exists a substitution σ'_u such that $\sigma_u = \sigma'_u\{f(x_1, \dots, x_n)/x\}$. Then σ'_u is the most general unifier of $(\dots, f(x_1, \dots, x_n), \dots)$ and $(\dots, f(p_1, \dots, p_n), \dots)$ (x does not occur in these tuples, since x occurs only once in $\mathsf{testunif}((\dots, x, \dots), (\dots, f(p_1, \dots, p_n), \dots))$ by *unify*). So $\mathrm{Rref}_1(R') = \sigma'_u H\{f(x_1, \dots, x_n)/x\} \Rightarrow \mathsf{bad} = \sigma_u H \Rightarrow \mathsf{bad} = \mathrm{Rref}_1(R)$.

A similar reasoning applies to *elimvar*. □

One should be careful that the simplification steps are applied to the clauses R, not to the reference clauses $\mathrm{Rref}(R)$, so $\mathrm{Rref}(R)$ may not be simplified. Even though we apply *elimattx* on R, $\mathrm{Rref}(R)$ may contain $\mathsf{att}(x)$ when x does not occur elsewhere in $\mathrm{Rref}(R)$ (x may occur in the $\mathsf{testunif}$ fact in R). Even though duplicate hypotheses are eliminated in R, there may be duplicate hypotheses in $\mathrm{Rref}(R)$. The decomposition of data constructors does not imply that the arguments of att in $\mathrm{Rref}(R)$ are non-data, in the particular case $\mathsf{att}(\mathsf{x}) \wedge \mathsf{testunif}(\mathsf{x}, f(\dots)) \wedge H \Rightarrow \mathsf{bad}$, where f is a data constructor. Whether such clauses can exist and be used to generate other clauses depends on the order of generation of clauses. Indeed, these clauses will be removed since they are subsumed by $\mathsf{att}(\mathsf{x}) \Rightarrow \mathsf{bad}$ which is generated from (Rc) and (Rcom). However, they remain in the set of clauses until $\mathsf{att}(\mathsf{x}) \Rightarrow \mathsf{bad}$ is generated. So in the proof, we use that the arguments of att in R are non-data, but not necessarily in $\mathrm{Rref}(R)$. If $\mathrm{Rref}(R)$ is a tautology, R is also a tautology (if the conclusion of $\mathrm{Rref}(R)$ is in its hypothesis, the same is true for R), so it is removed. R remains in the set of clauses even if $\mathrm{Rref}(R)$ is subsumed by some clause in the set.

In the following proof, we denote by $\mathrm{mgu}(p_1 = p'_1, \dots, p_n = p'_n)$ the most general unifier of $p_1, \dots, p_n$ and $p'_1, \dots, p'_n$. Obviously, the order of equalities does not change the value of the most general unifier. Moreover, if $\sigma_u = \mathrm{mgu}(p_1 = p'_1)$, then $\mathrm{mgu}(p_2 = p'_2, p_1 = p'_1) = \mathrm{mgu}(\sigma_u p_2 = \sigma_u p'_2)\sigma_u$. This remark is the key of the following proof.

Lemma 21 *If $R \in \mathcal{R}_{\mathrm{Prot0}}$ and $R' \in \mathcal{R}_{\mathrm{Prot0}}$, $R \circ_{F_0} R'$ is defined, and is not removed by unify, then we have:*

- $\mathrm{Rref}_1(R \circ_{F_0} R') = \mathrm{Rref}_1(R) \circ_{F_0'} \mathrm{Rref}_1(R')$, *where F_0' is the image of F_0 by the transformation of R' into $\mathrm{Rref}_1(R')$ (and both members of the equality are defined).*

- $\mathrm{OneKey}(R \circ_{F_0} R') = \mathrm{OneKey}(R) \circ_{F_0'} \mathrm{OneKey}(R')$, *where F_0' is the image of F_0 by the transformation of R' into $\mathrm{OneKey}(R')$.*

- $\mathrm{Rref}(R \circ_{F_0} R') = \mathrm{Rref}(R) \circ_{F_0'} \mathrm{Rref}(R')$, *where F_0' is the image of F_0 by the transformation of R' into $\mathrm{Rref}(R')$ (and both members of the equality are defined).*

Note that, for this result, it is important to keep duplicate hypotheses in $\mathrm{Rref}(R')$.

Proof *First case:* R and R' do not contain $\mathsf{testunif}$ facts.

Let $\mathrm{VarSubst}(R) = H \Rightarrow C$, $\mathrm{VarSubst}(R') = F_0' \wedge H' \Rightarrow C'$, $\mathit{Secr} = \{\mathsf{x}_1, \dots, \mathsf{x}_n\}$, x_i the variable substituted for x_i in $\mathrm{VarSubst}(R)$, and x_i' the variable substituted for x_i in $\mathrm{VarSubst}(R')$. Then we have

$$\begin{aligned}
&R = \mathrm{mgu}(x_i = \mathsf{x}_i)(H \Rightarrow C)\\
&R' = \mathrm{mgu}(x_i' = \mathsf{x}_i)(F_0' \wedge H' \Rightarrow C')\\
&R \circ_{F_0} R' = \mathrm{mgu}(C = F_0', x_i = \mathsf{x}_i, x_i' = \mathsf{x}_i)(H \wedge H' \Rightarrow C')
\end{aligned}$$

Let $\sigma_u = \mathrm{mgu}(C = F_0')$. Then $R \circ_{F_0} R' = \mathrm{mgu}(\sigma_u x_i = \mathsf{x}_i, \sigma_u x_i' = \mathsf{x}_i)\sigma_u(H \wedge H' \Rightarrow C')$. So $\sigma_u x_i$ and $\sigma_u x_i'$ unify with x_i and they cannot be in Secr since elements of Secr do not occur in C and F_0', then they are variables. Moreover, $\sigma_u(H \wedge H' \Rightarrow C')$ does not contain elements of Secr. Then

$$\mathrm{Rref}_1(R \circ_{F_0} R') = \mathrm{VarSubst}(R \circ_{F_0} R') = \sigma_u(H \wedge H' \Rightarrow C').$$

On the other hand,

$$\begin{aligned}
\mathrm{Rref}_1(R) &= H \Rightarrow C\\
\mathrm{Rref}_1(R') &= F_0' \wedge H' \Rightarrow C'\\
\text{so } \mathrm{Rref}_1(R) \circ_{F_0'} \mathrm{Rref}_1(R') &= \sigma_u(H \wedge H' \Rightarrow C')
\end{aligned}$$

which yields the required equality.

Second case: R or R' contains a $\mathsf{testunif}$ fact.

Since the conclusion of R unifies with some hypothesis of R', the conclusion of R is not bad, so R does not contain a $\mathsf{testunif}$ fact (since $R \in \mathcal{R}_{\mathrm{Prot0}}$), and R' contains a $\mathsf{testunif}$ fact. Let $\mathrm{VarSubst}'$ be defined as $\mathrm{VarSubst}$, but using $\mathit{Secr} \cup \mathit{EVar}$ instead of Secr. Let $\mathrm{VarSubst}'(R) = H \Rightarrow \mathsf{att}(p)$, $\mathrm{VarSubst}'(R') = \mathsf{att}(p') \wedge H' \wedge \mathsf{testunif}(p_1, p_2) \Rightarrow \mathsf{bad}$, and $F_0'' = \mathsf{att}(p')$. Let

$Secr = \{\mathsf{x}_1, \dots, \mathsf{x}_n\}$, $EVar = \{\mathsf{e}_1, \dots, \mathsf{e}_m\}$, x_i the variable substituted for x_i in $\mathrm{VarSubst}'(R)$, y_j substituted for e_j in $\mathrm{VarSubst}'(R)$, x'_i substituted for x_i in $\mathrm{VarSubst}'(R')$, and y'_j substituted for e_j in $\mathrm{VarSubst}'(R')$.

Similarly to the case without $\mathsf{testunif}$, we have then

$$
\begin{aligned}
&R = \mathrm{mgu}(x_i = \mathsf{x}_i, y_j = \mathsf{e}_j)(H \Rightarrow \mathsf{att}(p))\\
&R' = \mathrm{mgu}(x'_i = \mathsf{x}_i, y'_j = \mathsf{e}_j)(\mathsf{att}(p') \wedge H' \wedge \mathsf{testunif}(p_1, p_2) \Rightarrow \mathsf{bad})\\
&R \circ_{F_0} R' = \mathrm{mgu}(x_i = \mathsf{x}_i, y_j = \mathsf{e}_j, x'_i = \mathsf{x}_i, y'_j = \mathsf{e}_j, p = p')\\
&\qquad\qquad (H \wedge H' \wedge \mathsf{testunif}(p_1, p_2) \Rightarrow \mathsf{bad})\\
&\mathrm{VarSubst}'(R \circ_{F_0} R') = \mathrm{mgu}(p = p')(H \wedge H' \wedge \mathsf{testunif}(p_1, p_2) \Rightarrow \mathsf{bad})
\end{aligned}
$$

so $\mathrm{Rref}_1(R \circ_{F_0} R') = \mathrm{mgu}(p_1 = p_2, p = p')(H \wedge H' \Rightarrow \mathsf{bad})$. This is defined because $R \circ_{F_0} R'$ is not removed by *unify*, so $\mathrm{mgu}(p = p')p_1$ and $\mathrm{mgu}(p = p')p_2$ unify.

On the other hand,

$$
\begin{aligned}
&\mathrm{Rref}_1(R) = H \Rightarrow \mathsf{att}(p)\\
&\mathrm{Rref}_1(R') = \mathrm{mgu}(p_1 = p_2)(\mathsf{att}(p') \wedge H' \Rightarrow \mathsf{bad})\\
\text{so } &\mathrm{Rref}_1(R) \circ_{F'_0} \mathrm{Rref}_1(R') = \mathrm{mgu}(p = p', p_1 = p_2)(H \wedge H' \Rightarrow \mathsf{bad})
\end{aligned}
$$

which yields the required equality.

The same property for OneKey is easy, and the third point immediately comes from the first two. □

Proposition 10 *If $\mathcal{R}_0$ is a weakly tagged program (Definition 9), then the computation of* $\mathsf{saturate}(\mathcal{R}_0)$ *terminates.*

Proof We show by induction that all rules R generated from $\mathcal{R}_0$ are in $\mathcal{R}_{\mathrm{Prot0}} \cup \mathcal{R}_{\mathrm{Prot1}}$, and the patterns of facts $\mathsf{att}(\dots)$ in R are non-data.

First, by hypothesis, all rules in $\mathcal{R}_0$ are in $\mathcal{R}_{\mathrm{Prot0}} \cup \mathcal{R}_{\mathrm{Prot1}}$.

First note that the set of clauses $\mathcal{R}_{\mathrm{Prot0}}$ is preserved by simplification, and that the size of a simplified clause is at most the size of the clause before simplification. The decomposition of data constructors takes subterms of the patterns of the clause, without changing the size. The *elimattx* removes patterns without changing the size. The various simplifications for $\mathsf{testunif}$ facts change nothing by Lemma 20. The set $\mathcal{R}_{\mathrm{Prot1}}$ is obviously preserved by simplification, since no simplification modifies clauses in $\mathcal{R}_{\mathrm{Prot1}}$.

The patterns of facts $\mathsf{att}(\dots)$ in clauses obtained after simplification are non-data, because of the decomposition of data constructors.

Then all clauses R in $condense(\mathcal{R}_0)$ are in $\mathcal{R}_{\mathrm{Prot0}} \cup \mathcal{R}_{\mathrm{Prot1}}$, and the patterns of facts $\mathsf{att}(\dots)$ in R are non-data.

Next, the algorithm combines a clause R such that $\mathsf{sel}(R) \neq \emptyset$ with a clause R' such that $\mathsf{sel}(R') = \emptyset$ by resolution. Let R'' be the obtained clause before simplification, and R_s any obtained clause after simplification. We prove that $R'' \in \mathcal{R}_{\text{Prot0}}$, and, if R is not (Ratteq) and $R \notin \mathcal{R}_{\text{Prot1}}$, then $R'' < R$.

Case 1: $R, R' \in \mathcal{R}_{\text{Prot0}}$ and R is not (Ratteq).

Let $R = \mathsf{att}(p'_0) \wedge H \Rightarrow C$, $\mathsf{att}(p'_0)$ being the atom on which we resolve. By definition of the selection function, p'_0 is not a variable. We have $R' = \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_{n'}) \Rightarrow \mathsf{att}(p'_1)$ (since $\mathsf{sel}(R') = \emptyset$), the variables $x_1, \ldots, x_{n'}$ occur in p'_1 (by the optimization *elimattx*), and $\mathsf{att}(p'_1)$ is the atom of R' on which we resolve. Moreover, by Lemma 2, the variables in the conclusion of R' also occur in the hypothesis of R', so if p'_1 was a variable, R' would be a tautology, so it would have been eliminated. Then p'_1 is not a variable.

Case 1.1: p'_1 or p'_0 is in *Secr*.

Since p'_1 and p'_0 unify and they are not variables, then they are both in *Secr* and equal. Then $R' = \mathsf{att}(p'_0)$, $R'' = H \Rightarrow C$. Since $R \in \mathcal{R}_{\text{Prot0}}$, it is easy to see that $R'' \in \mathcal{R}_{\text{Prot0}}$. Moreover $R'' < R$.

Case 1.2: p'_1 and p'_0 are not in *Secr*.

Let $p_0, \ldots, p_n$ be the patterns of $\text{Rref}(R)$, p_0 corresponding to p'_0. Then p_0 is not a variable since p'_0 is not a variable and not in *Secr*. Moreover, p_0 has the same root symbol as p'_0, so p_0 is non-data since p'_0 is non-data. Let $p', x_1, \ldots, x_n$ be the patterns of $\text{Rref}(R')$, p' corresponding to p'_1. Then the variables $x_1, \ldots, x_{n'}$ occur in p' and p' is not a variable since p'_1 is not a variable and not in *Secr*. Moreover, p' has the same root symbol as p'_1, so p' is non-data since p'_1 is non-data. By Lemma 21, $\text{Rref}(R'') = \text{Rref}(R') \circ_{\mathsf{att}(p_0)} \text{Rref}(R)$, so the patterns p_0 and p' unify, let σ_u be their most general unifier. Then the patterns of $\text{Rref}(R'')$ are $\sigma_u x_1, \ldots, \sigma_u x_{n'}, \sigma_u p_1, \ldots, \sigma_u p_n$.

We show that there exist substitutions σ and σ' such that $\sigma p_0, \ldots, \sigma p_n$ are in $sub(S_0)$ and $\sigma' p' = \sigma p'_0$.

- If p' is not fully tagged, let σ such that $\sigma p_0, \ldots, \sigma p_n$ are in $sub(S_0)$. Moreover, p' has the same root symbol as p_0 since they unify. So p' is more general than σp_0 (by definition of $\mathcal{R}_{\text{Prot0}}$). Let σ' be a substitution such that $\sigma' p' = \sigma p_0$.

- If p_0 and p' are fully tagged, let σ such that $\sigma p_0, \ldots, \sigma p_n$ are in $sub(S_0)$. p_0 is non-variable, non-data, fully tagged, so it is top-tagged. Then p_0 is an instance of some p''_0 in *tagGen*. Let σ' such that $\sigma' p'$ is in $sub(S_0)$. p' is non-variable, non-data, fully tagged, so it is top-tagged. Then p' is an instance of some p'' in *tagGen*. Moreover, p'' and p''_0 unify since p' and p_0 unify. Then p' and p_0 in *tagGen* unify, $\sigma' p'$ is an instance of p'' in $sub(S_0)$, and σp_0 is an instance of p''_0 in $sub(S_0)$, so by Condition W2, $\sigma' p' = \sigma p_0$.

- If p_0 is not fully tagged and p' is fully tagged, let σ' such that $\sigma' p' \in sub(S_0)$.

p_0 has the same root symbol as p', since they unify. So p_0 is more general than $\sigma p'$ (by definition of $\mathcal{R}_{\mathrm{Prot0}}$). Let σ be a substitution such that $\sigma' p' = \sigma p_0$.

By hypothesis, there exists σ_2 such that $\mathit{patterns}(\sigma_2 \mathrm{Rref}(R)) \subseteq \mathit{sub}(S_0)$ so we can extend σ to all variables of $\mathrm{Rref}(R)$ by taking $\sigma x = \sigma_2 x$ for variables not in p_0. Then $\sigma p_0, \ldots, \sigma p_n$ are in $\mathit{sub}(S_0)$. Indeed, we have:

- By definition of σ, we have $\sigma p_0 \in \mathit{sub}(S_0)$.
- Consider σp_i when p_i contains a variable in common with p_0. If p_i is this common variable, σp_i is a subterm of σp_0, so in $\mathit{sub}(S_0)$. If p_i is not a variable, the variables x common between p_i and p_0 have the same image by all substitutions that map p_0 to some element of $\mathit{sub}(S_0)$, so $\sigma_2 x = \sigma x$. Then $\sigma p_i = \sigma_2 p_i \in \mathit{sub}(S_0)$.
- For σp_i containing no common variable with p_0, $\sigma p_i = \sigma_2 p_i \in \mathit{sub}(S_0)$.

In all cases, the substitution equal to σ on the variables of $\mathrm{Rref}(R)$ and to σ' on the variables of $\mathrm{Rref}(R')$ is then a unifier of p_0 and p'. Since σ_u is the most general unifier, there exists σ'' such that $\sigma''\sigma_u$ is equal to σ on the variables of $\mathrm{Rref}(R)$, and to σ' on the variables of $\mathrm{Rref}(R')$. Thus the patterns of $\sigma''\mathrm{Rref}(R'')$ are $\sigma' x_1, \ldots, \sigma' x_{n'}, \sigma p_1, \ldots, \sigma p_n$. The patterns $\sigma' x_1, \ldots, \sigma' x_{n'}$ are subterms of $\sigma' p' = \sigma p_0$ which is in $\mathit{sub}(S_0)$, so they are also in $\mathit{sub}(S_0)$. So all patterns of $\sigma''\mathrm{Rref}(R'')$ are in $\mathit{sub}(S_0)$.

We have $p' = f(p'_1, \ldots, p'_m)$ and $p_0 = f(p''_1, \ldots, p''_m)$, so σ_u is also the most general unifier of the pairs (p'_1, p''_1), $\ldots$, (p'_m, p''_m) of fully tagged patterns. So by Lemma 10, for all x, $\sigma_u x$ is fully tagged, so by Lemma 8, for all fully tagged patterns p, $\sigma_u p$ is fully tagged. We already have that $\sigma_u x_1, \ldots, \sigma_u x_n$ are fully tagged. Let $p_i, i > 0$ be a pattern of $\mathrm{Rref}(R)$.

- If $\sigma_u p_i$ is an argument of com in $\mathrm{Rref}(R'')$, p_i is an argument of com in $\mathrm{Rref}(R)$, so p_i is fully tagged, so $\sigma_u p_i$ is fully tagged.

- If $\sigma_u p_i$ is an argument of att in $\mathrm{Rref}(R'')$, p_i is an argument of att in $\mathrm{Rref}(R)$.

  If p_i is fully tagged, then $\sigma_u p_i$ is fully tagged.

  Otherwise, p_i is not fully tagged. Its strict subterms are fully tagged. Since for all x, $\sigma_u x$ is fully tagged, by Lemma 8, the strict subterms of $\sigma_u p_i$ are also fully tagged.

  Let p'_i be some pattern in $\mathit{sub}(S_0)$ with the same root symbol as p_i. By hypothesis, p_i is more general than p'_i: there exists σ_1 such that $p'_i = \sigma_1 p_i$.

If x is a common variable of p_i and p_0, by hypothesis, x has the same image by all substitutions that map p_i to some term in $sub(S_0)$. Since we have seen that $\sigma''\sigma_u p_i$ is in $sub(S_0)$, $\sigma''\sigma_u x = \sigma_1 x$.

If x is in p_i but not in p_0, then $\sigma_u x = x$, and x does not occur in $\sigma_u y$ for any y common between p_0 and p_i.

So let us define σ_2 by σ_2 is σ'' on the variables in $\sigma_u x$ for x common between p_0 and p_i, and σ_2 is σ_1 on other variables. Then $\sigma_2\sigma_u p_i = \sigma_1 p_i = p'_i$, so $\sigma_u p_i$ is more general than p'_i.

Consider a variable x of $\sigma_u p_i$. If x does not occur in some $\sigma_u y$ such that $\sigma_u y \neq y$, then $\sigma_u x = x$ and x inherits its properties from the properties of x in p_i. If x occurs in some $\sigma_u y$ such that $\sigma_u y \neq y$, then y is a common variable of p_i and p_0, so by hypothesis, y has the same image by all substitutions that map p_i to some term in $sub(S_0)$. A fortiori, $\sigma_u y$ has the same image by all substitutions that map $\sigma_u p_i$ to some term in $sub(S_0)$, so x has the same image by all substitutions that map $\sigma_u p_i$ to some term in $sub(S_0)$.

Therefore, $R'' \in \mathcal{R}_{\text{Prot0}}$.

Let us show that $R'' < R$. We have $size(\text{Rref}(R)) = \sum_{i=0}^{n} size(p_i)$ and $size(\text{Rref}(R'')) = \sum_{i=1}^{n} size(\sigma_u p_i) + \sum_{i=1}^{n'} size(\sigma_u x_i)$.

Let $i > 0$. Let us show that $size(\sigma_u p_i) \leq size(p_i) + \sum_{\text{some } x} size(\sigma_u x)$.

- If p_i is not fully tagged, its strict subterms are fully tagged, so p_i is non-data, not top-tagged, and not a variable, since otherwise, p_i would be fully tagged. Then $size(p_i) = \top \geq size(\sigma_u p_i)$.

- Assume that p_i is fully tagged. Let p'' be a subterm of p_i, and let us show by induction on p'' that $size(\sigma_u p'') \leq size(p'') + \sum_{\text{some } x} size(\sigma_u x)$. This will yield the expected result.

  If p'' is data, $p'' = f(p'_1, \ldots, p'_m)$ with f a data constructor. Then $size(p'') = size(p'_1) + \ldots + size(p'_m)$ and $size(\sigma_u p'') = size(\sigma_u p'_1) + \ldots + size(\sigma_u p'_m)$, so we get the result by applying the induction hypothesis to $p'_1, \ldots, p'_m$.

  If p'' is a variable, $size(p'') = \{\bot\}$, $size(\sigma_u p'') = size(\sigma_u x)$ for the variable $x = p''$.

  Otherwise, p'' is non-variable, non-data, fully tagged (since it is a subterm of p_i), so top-tagged, $size(p'') = \{\sigma p''\} = \{\sigma''\sigma_u p''\} = size(\sigma_u p'')$, since $\sigma p'' = \sigma''\sigma_u p'' \in sub(S_0)$ because p'' is a subterm of p_i.

Hence, $size(\text{Rref}(R'')) \leq \sum_{i=1}^{n} size(p_i) + \sum_{\text{some } x} size(\sigma_u x)$. It is enough to show that $size(p_0) > \sum_{\text{some } x} size(\sigma_u x)$.

- If p_0 is not fully tagged, $size(p_0) = \top$ as above. On the other hand, for all x, $\sigma_u x$ is fully tagged and $\sigma''\sigma_u x \in sub(S_0)$ so by Lemma 19,

$size(\sigma_u x)$ is a multiset of subterms of $\sigma''\sigma_u x$. So $size(\sigma_u x)$ does not contain $\top$. Hence $size(p_0) > \sum_{\text{some } x} size(\sigma_u x)$.

- Assume that p_0 is fully tagged. Then p_0 is non-variable, non-data, fully tagged, so top-tagged. So $size(p_0) = \{\sigma p_0\}$. If x is not modified by σ_u, $size(\sigma_u x) = \{\perp\}$. If x is modified by σ_u, x occurs in p_0 or p', so $\sigma''\sigma_u x$ is a strict subterm of $\sigma p_0 = \sigma' p'$. For all x, $\sigma_u x$ is fully tagged and $\sigma''\sigma_u x \in sub(S_0)$ so by Lemma 19, $size(\sigma_u x)$ is a multiset of subterms of $\sigma''\sigma_u x$, so strict subterms of σp_0. Therefore, $size(p_0) > \sum_{\text{some } x} size(\sigma_u x)$.

So $size(\mathrm{Rref}(R'')) < size(\mathrm{Rref}(R))$.

Case 2: R is (Ratteq) and $R' \in \mathcal{R}_{\mathrm{Prot0}}$.

Then $\mathsf{sel}(R) = \{\mathsf{att}(x_1)\}$, $R' = \mathsf{att}(x'_1) \wedge \ldots \wedge \mathsf{att}(x'_n) \Rightarrow \mathsf{att}(p)$, $\mathrm{Rref}(R') = \mathsf{att}(x'_1) \wedge \ldots \wedge \mathsf{att}(x'_n) \Rightarrow \mathsf{att}(p')$, where p' is obtained from p by substituting variables for elements of $Secr \cup EVar$.

Then $R' \circ_{F_0} R = \mathsf{att}(x'_1) \wedge \ldots \wedge \mathsf{att}(x'_n) \wedge \mathsf{att}(x_2) \wedge \mathsf{testunif}(p, x_2) \Rightarrow \mathsf{bad}$, and $\mathrm{Rref}(R' \circ_{F_0} R) = \mathsf{att}(x'_1) \wedge \ldots \wedge \mathsf{att}(x'_n) \wedge \mathsf{att}(p') \Rightarrow \mathsf{bad}$. Since $R' \in \mathcal{R}_{\mathrm{Prot0}}$, it is then easy to show that $R'' = R' \circ_{F_0} R \in \mathcal{R}_{\mathrm{Prot0}}$. (The terms of $\mathrm{Rref}(R' \circ_{F_0} R)$ are exactly the same as the ones of $\mathrm{Rref}(R')$.)

Case 3: $R \in \mathcal{R}_{\mathrm{Prot1}}$ and $R' \in \mathcal{R}_{\mathrm{Prot0}}$.

Then $R = \mathsf{com}(\mathsf{x}) \Rightarrow \mathsf{bad}$, $\mathsf{sel}(R) = \mathsf{com}(\mathsf{x})$, $R' = \mathsf{att}(x'_1) \wedge \ldots \wedge \mathsf{att}(x'_n) \Rightarrow \mathsf{com}(p)$. Since p unifies with x, either p is a variable or $p = \mathsf{x}$. By *elimattx*, if $p = \mathsf{x}$, $R' = \mathsf{com}(\mathsf{x})$, $R' \circ_{F_0} R = \mathsf{bad}$, so $R' \circ_{F_0} R \in \mathcal{R}_{\mathrm{Prot0}}$. If p is a variable, $R' = \mathsf{att}(x) \Rightarrow \mathsf{com}(x)$, $R' \circ_{F_0} R = \mathsf{att}(\mathsf{x}) \Rightarrow \mathsf{bad}$, $\mathrm{Rref}(R' \circ_{F_0} R) = \mathsf{att}(y) \Rightarrow \mathsf{bad}$, so $R' \circ_{F_0} R \in \mathcal{R}_{\mathrm{Prot0}}$.

Termination proof: The set of clauses R of $\mathcal{R}_{\mathrm{Prot0}}$ with $\mathsf{sel}(R) = \emptyset$ is finite. Indeed, it contains clauses $\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \Rightarrow C$ with $C = \mathsf{att}(p)$ or $C = \mathsf{com}(p)$, plus the clause bad. (If such a clause contained a $\mathsf{testunif}$ hypothesis, since $\mathrm{Rref}(R)$ is defined, its conclusion would be bad. Then by Lemma 1, $\mathsf{sel}(R)$ would not be empty.) After substitution of variables for elements of $Secr \cup EVar$, and application of OneKey, the pattern p has instances in the finite set S_0, so there is a finite number of such patterns (since *Keys* is finite by construction). The variables $x_1, \ldots, x_n$ must also occur in the conclusion of R (by *elimattx*), so there is a finite number of possible hypotheses for R.

The set of clauses $\mathcal{R}_{\mathrm{Prot1}}$ is finite by construction. The set $\mathcal{R}_1$ of clauses obtained from $\mathcal{R}_{\mathrm{Prot1}}$ or from (Ratteq) by one resolution step is then also finite.

The set $\mathcal{R}_2$ of initial clauses different from (Ratteq) and not in $\mathcal{R}_{\mathrm{Prot1}}$ is also finite.

All other clauses are obtained by resolution steps starting from a clause R in $\mathcal{R}_1 \cup \mathcal{R}_2$, with clauses R' such that $\mathsf{sel}(R') = \emptyset$. The sequences of obtained clauses are strictly decreasing by the previous proof. Since the ordering is well-founded, the sequences are then finite. The resolution tree is then finite, since the branching is finite. So $\mathsf{saturate}(\mathcal{R}_0)$ terminates. $\square$

From the previous results, we obtain immediately Proposition 5.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2004-NWG3-001	M. Magnor	Axisymmetric Reconstruction and 3D Visualization of Bipolar Planetary Nebulae
MPI-I-2004-NWG1-001	B. Blanchet	?
MPI-I-2004-5-001	S. Siersdorfer, S. Sizov, G. Weikum	Goal-oriented Methods and Meta Methods for Document Classification and their Parameter Tuning
MPI-I-2004-4-004	R. Zayer, C. Rössl, H. Seidel	r-Adaptive Parameterization of Surfaces
MPI-I-2004-4-003	Y. Ohtake, A. Belyaev, H. Seidel	3D Scattered Data Interpolation and Approximation with Multilevel Compactly Supported RBFs
MPI-I-2004-4-002	Y. Ohtake, A. Belyaev, H. Seidel	Quadric-Based Mesh Reconstruction from Scattered Data
MPI-I-2004-4-001	J. Haber, C. Schmitt, M. Koster, H. Seidel	Modeling Hair using a Wisp Hair Model
MPI-I-2004-2-002	P. Maier	Intuitionistic LTL and a New Characterization of Safety and Liveness
MPI-I-2004-2-001	H.d. Nivelle, Y. Kazakov	Resolution Decision Procedures for the Guarded Fragment with Transitive Guards
MPI-I-2004-1-005	S. Schmitt, L. Fousse	A comparison of polynomial evaluation schemes
MPI-I-2004-1-004	N. Sivadasan, P. Sanders, M. Skutella	Online Scheduling with Bounded Migration
MPI-I-2004-1-003	I. Katriel	On Algorithms for Online Topological Ordering and Sorting
MPI-I-2004-1-002	P. Sanders, S. Pettie	A Simpler Linear Time 2/3 - epsilon Approximation for Maximum Weight Matching
MPI-I-2004-1-001	N. Beldiceanu, I. Katriel, S. Thiel	Filtering algorithms for the Same and UsedBy constraints
MPI-I-2003-NWG2-002	F. Eisenbrand	Fast integer programming in fixed dimension
MPI-I-2003-NWG2-001	L.S. Chandran, C.R. Subramanian	Girth and Treewidth
MPI-I-2003-4-009	N. Zakaria	FaceSketch: An Interface for Sketching and Coloring Cartoon Faces
MPI-I-2003-4-008	C. Roessl, I. Ivrissimtzis, H. Seidel	Tree-based triangle mesh connectivity encoding
MPI-I-2003-4-007	I. Ivrissimtzis, W. Jeong, H. Seidel	Neural Meshes: Statistical Learning Methods in Surface Reconstruction
MPI-I-2003-4-006	C. Roessl, F. Zeilfelder, G. Nürnberger, H. Seidel	Visualization of Volume Data with Quadratic Super Splines
MPI-I-2003-4-005	T. Hangelbroek, G. Nürnberger, C. Roessl, H.S. Seidel, F. Zeilfelder	The Dimension of C^1 Splines of Arbitrary Degree on a Tetrahedral Partition

MPI-I-2003-4-004	P. Bekaert, P. Slusallek, R. Cools, V. Havran, H. Seidel	A custom designed density estimation method for light transport
MPI-I-2003-4-003	R. Zayer, C. Roessl, H. Seidel	Convex Boundary Angle Based Flattening
MPI-I-2003-4-002	C. Theobalt, M. Li, M. Magnor, H. Seidel	A Flexible and Versatile Studio for Synchronized Multi-view Video Recording
MPI-I-2003-4-001	M. Tarini, H.P.A. Lensch, M. Goesele, H. Seidel	3D Acquisition of Mirroring Objects
MPI-I-2003-2-004	A. Podelski, A. Rybalchenko	Software Model Checking of Liveness Properties via Transition Invariants
MPI-I-2003-2-003	Y. Kazakov, H. Nivelle	Subsumption of concepts in $DL\ \mathcal{FL}_0$ for (cyclic) terminologies with respect to descriptive semantics is PSPACE-complete
MPI-I-2003-2-002	M. Jaeger	A Representation Theorem and Applications to Measure Selection and Noninformative Priors
MPI-I-2003-2-001	P. Maier	Compositional Circular Assume-Guarantee Rules Cannot Be Sound And Complete
MPI-I-2003-1-018	G. Schaefer	A Note on the Smoothed Complexity of the Single-Source Shortest Path Problem
MPI-I-2003-1-017	G. Schäfer, S. Leonardi	Cross-Monotonic Cost Sharing Methods for Connected Facility Location Games
MPI-I-2003-1-016	G. Schäfer, N. Sivadasan	Topology Matters: Smoothed Competitive Analysis of Metrical Task Systems
MPI-I-2003-1-015	A. Kovács	Sum-Multicoloring on Paths
MPI-I-2003-1-014	G. Schäfer, L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, T. Vredeveld	Average Case and Smoothed Competitive Analysis of the Multi-Level Feedback Algorithm
MPI-I-2003-1-013	I. Katriel, S. Thiel	Fast Bound Consistency for the Global Cardinality Constraint
MPI-I-2003-1-012		- not published -
MPI-I-2003-1-011	P. Krysta, A. Czumaj, B. Voecking	Selfish Traffic Allocation for Server Farms
MPI-I-2003-1-010	H. Tamaki	A linear time heuristic for the branch-decomposition of planar graphs
MPI-I-2003-1-009	B. Csaba	On the Bollobás – Eldridge conjecture for bipartite graphs
MPI-I-2003-1-008	P. Sanders	Polynomial Time Algorithms for Network Information Flow
MPI-I-2003-1-007	H. Tamaki	Alternating cycles contribution: a strategy of tour-merging for the traveling salesman problem
MPI-I-2003-1-006	M. Dietzfelbinger, H. Tamaki	On the probability of rendezvous in graphs
MPI-I-2003-1-005	M. Dietzfelbinger, P. Woelfel	Almost Random Graphs with Simple Hash Functions
MPI-I-2003-1-004	E. Althaus, T. Polzin, S.V. Daneshmand	Improving Linear Programming Approaches for the Steiner Tree Problem
MPI-I-2003-1-003	R. Beier, B. Vöcking	Random Knapsack in Expected Polynomial Time
MPI-I-2003-1-002	P. Krysta, P. Sanders, B. Vöcking	Scheduling and Traffic Allocation for Tasks with Bounded Splittability
MPI-I-2003-1-001	P. Sanders, R. Dementiev	Asynchronous Parallel Disk Sorting
MPI-I-2002-4-002	F. Drago, W. Martens, K. Myszkowski, H. Seidel	Perceptual Evaluation of Tone Mapping Operators with Regard to Similarity and Preference
MPI-I-2002-4-001	M. Goesele, J. Kautz, J. Lang, H.P.A. Lensch, H. Seidel	Tutorial Notes ACM SM 02 A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2002-2-008	W. Charatonik, J. Talbot	Atomic Set Constraints with Projection
MPI-I-2002-2-007	W. Charatonik, H. Ganzinger	Symposium on the Effectiveness of Logic in Computer Science in Honour of Moshe Vardi
MPI-I-2002-1-008	P. Sanders, J.L. Träff	The Factor Algorithm for All-to-all Communication on Clusters of SMP Nodes

MPI-I-2002-1-005	M. Hoefer	Performance of heuristic and approximation algorithms for the uncapacitated facility location problem
MPI-I-2002-1-004	S. Hert, T. Polzin, L. Kettner, G. Schäfer	Exp Lab A Tool Set for Computational Experiments
MPI-I-2002-1-003	I. Katriel, P. Sanders, J.L. Träff	A Practical Minimum Scanning Tree Algorithm Using the Cycle Property