Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif

Bruno Blanchet

Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif

Suggested Citation: Bruno Blanchet (2016), "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif", : Vol. 1, No. 1, pp 1–135. DOI: 10.1561/3300000004.

Bruno Blanchet
INRIA Paris, France
Bruno.Blanchet@inria.fr

now
the essence of knowledge
Boston — Delft

Contents

Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif

Bruno Blanchet[1]

[1] *INRIA Paris, France; Bruno.Blanchet@inria.fr*

ABSTRACT

ProVerif is an automatic symbolic protocol verifier. It supports a wide range of cryptographic primitives, defined by rewrite rules or by equations. It can prove various security properties: secrecy, authentication, and process equivalences, for an unbounded message space and an unbounded number of sessions. It takes as input a description of the protocol to verify in a dialect of the applied pi calculus, an extension of the pi calculus with cryptography. It automatically translates this protocol description into Horn clauses and determines whether the desired security properties hold by resolution on these clauses. This survey presents an overview of the research on ProVerif.

Bruno Blanchet (2016), "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif", : Vol. 1, No. 1, pp 1–135. DOI: 10.1561/3300000004.

1

Introduction

1.1 Verifying security protocols

The verification of security protocols has been an active research area since the 1990s. This topic is interesting for several reasons. Security protocols are ubiquitous: they are used for e-commerce, wireless networks, credit cards, e-voting, among others. The design of security protocols is notoriously error-prone. This point can be illustrated by attacks found against many published protocols. For instance, a famous attack was discovered by Lowe, 1996 against the Needham-Schroeder public-key protocol (Needham and Schroeder, 1978) 17 years after its publication. Attacks are also found against many protocols used in practice. Important examples are SSL (Secure Sockets Layer) and its successor TLS (*Transport Layer Security*), which are used for https:// connexions. The first version dates back to 1994, and since then many attacks were discovered, fixed versions were developed, and new attacks are still regularly discovered (Beurdouche *et al.*, 2015; Adrian *et al.*, 2015). Moreover, security errors cannot be detected by functional testing, since they appear only in the presence of a malicious adversary. These errors can also have serious consequences. Hence, the formal verification or proof of protocols is particularly desirable.

1.1.1 Modeling security protocols

In order to verify protocols, two main models have been considered:

- In the *symbolic model*, often called Dolev-Yao model and due to Needham and Schroeder, 1978 and Dolev and Yao, 1983, cryptographic primitives are considered as perfect blackboxes, modeled by function symbols in an algebra of terms, possibly with equations. Messages are terms on these primitives and the adversary can compute only using these primitives. This is the model usually considered by formal method practitioners.

- In contrast, in the *computational model*, messages are bitstrings, cryptographic primitives are functions from bitstrings to bitstrings, and the adversary is any probabilistic Turing machine. This is the model usually considered by cryptographers.

The symbolic model is an abstract model that makes it easier to build automatic verification tools, and many such tools exist: AVISPA (Armando *et al.*, 2005), FDR (Lowe, 1996), Scyther (Cremers, 2008), Tamarin (Schmidt *et al.*, 2012), for instance. The computational model is closer to the real execution of protocols, but the proofs are more difficult to automate; we refer the reader to (Blanchet, 2012a) and to Chapter 6 for some information on the mechanization of proofs in the computational model.

Most often, the relations between cryptographic primitives given in the symbolic model also hold in the computational model.[1] In this case, an attack in the symbolic model directly leads to an attack in the computational model, and a practical attack. However, the converse is not true in general: a protocol may be proved secure in the symbolic model, and still be subject to attacks in the computational model. For this reason, the *computational soundness* approach was introduced: it proves general theorems showing that security in the symbolic model implies security in the computational model, modulo additional assumptions. However, since the two models do not coincide, this approach

[1] Sometimes, one may also overapproximate the capabilities of the adversary in the symbolic model.

typically requires strong assumptions on the cryptographic primitives (for instance, encryption has to hide the length of the messages) and on the protocol (for instance, absence of key cycles, in which a key is encrypted under itself; correctly generated keys, even for the adversary). This approach was pioneered by Abadi and Rogaway, 2002. This work triggered much research in this direction; we refer to (Cortier *et al.*, 2011) for a survey.

Even though the computational model is closer to reality than the symbolic model, we stress that it is still a model. In particular, it does not take into account side channels, such as timing and power consumption, which may give additional information to an adversary and enable new attacks. Moreover, one often studies specifications of protocols. New attacks may appear when the protocol is implemented, either because the specification has not been faithfully implemented, or because the attacks rely on implementation details that do not appear at the specification level.

In this survey, we focus on the verification of specifications of protocols in the symbolic model. Even though it is fairly abstract, this level of verification is relevant in practice as it enables the discovery of many attacks.

1.1.2 Target security properties

Security protocols can aim at a wide variety of security goals. The main security properties can be classified into two categories, *trace properties* and *equivalence properties*. We define these categories and mention two particularly important examples: secrecy and authentication. These are two basic properties required by most security protocols. Some protocols, such as e-voting protocols (Delaune *et al.*, 2009), require more complex and specific security properties, which we will not discuss.

Trace and equivalence properties

Trace properties are properties that can be defined on each execution trace (each run) of the protocol. The protocol satisfies such a property when it holds for all traces. For example, the fact that some states are

unreachable is a trace property.

Equivalence properties mean that the adversary cannot distinguish two processes (that is, protocols). For instance, one of these processes can be the protocol under study, and the other one can be its specification. Then, the equivalence means that the protocol satisfies its specification. Therefore, equivalences can be used to model many subtle security properties. Several variants exist (observational equivalence, testing equivalence, trace equivalence) (Abadi and Gordon, 1999; Abadi and Gordon, 1998; Abadi and Fournet, 2001). Observational equivalence provides compositional proofs: if a protocol P is equivalent to P', P can be replaced with P' in a more complex protocol. However, the proof of equivalences is more difficult to automate than the proof of trace properties: equivalences cannot be expressed on a single trace, they require relations between traces (or processes).

Secrecy

Secrecy, or confidentiality, means that the adversary cannot obtain some information on data manipulated by the protocol. Secrecy can be formalized in two ways:

- Most often, secrecy means that the adversary cannot compute exactly the considered piece of data. In this survey, this property will simply be named *secrecy*, or when emphasis is needed, syntactic secrecy.

- Sometimes, one uses a stronger notion, *strong secrecy*, which means that the adversary cannot detect a change in the value of the secret (Abadi, 1999; Blanchet, 2004). In other words, the adversary has no information at all on the value of the secret.

The difference between syntactic secrecy and strong secrecy can be illustrated by a simple example: consider a piece of data for which the adversary knows half of the bits but not the other half. This piece of data is syntactically secret since the adversary cannot compute it entirely, but not strongly secret, since the adversary can see if one of the bits it knows changes. Syntactic secrecy cannot be used to

express secrecy of data chosen among known constants. For instance, talking about syntactic secrecy of a boolean true or false does not make sense, because the adversary knows the constants true and false from the start. In this case, one has to use strong secrecy: the adversary must not be able to distinguish a protocol using the value true from the same protocol using the value false. These two notions are often equivalent (Cortier *et al.*, 2007), for atomic data (data that cannot be split into several pieces, such as nonces, which are random numbers chosen independently at each run of the protocol) and for probabilistic cryptographic primitives. Syntactic secrecy is a trace property, while strong secrecy is an equivalence property.

Authentication

Authentication means that, if a participant A runs the protocol apparently with a participant B, then B runs the protocol apparently with A, and conversely. One often requires that A and B also share the same values of the parameters of the protocol.

Authentication is generally formalized by correspondence properties (Woo and Lam, 1993; Lowe, 1997), of the form: if A executes a certain event e_1 (for instance, A terminates the protocol with B), then B has executed a certain event e_2 (for instance, B started a session of the protocol with A). There exist several variants of these properties. For instance, one may require that each execution of e_1 corresponds to a distinct execution of e_2 (injective correspondence) or, on the contrary, that if e_1 has been executed, then e_2 has been executed at least once (non-injective correspondence). The events e_1 and e_2 may also include more or fewer parameters depending on the desired property. These properties are trace properties.

1.1.3 Symbolic verification

Basically, to verify protocols in the symbolic model, one computes the set of terms (messages) that the adversary knows. If a message does not belong to this set, then this message is secret. The difficulty is that this set is infinite, for two reasons: the adversary can build terms

of unbounded size, and the considered protocol can be executed any number of times. Several approaches can be considered to solve this problem:

- One can bound the size of messages and the number of executions of the protocols. In this case, the state space is finite, and one can apply standard model-checking techniques. This is the approach taken by FDR (Lowe, 1996) and by SATMC (Armando *et al.*, 2014), for instance.

- If we bound only the number of executions of the protocol, the state space is infinite, but under reasonable assumptions, one can show that the problem of security protocol verification is decidable: protocol insecurity is NP-complete (Rusinowitch and Turuani, 2003). Basically, the non-deterministic Turing machine guesses an attack and polynomially checks that it is actually an attack against the protocol. There exist practical tools that can verify protocols in this case, using for instance constraint solving as in Cl-AtSe (Turuani, 2006) or extensions of model checking as in OFMC (Basin *et al.*, 2005).

- When the number of executions of the protocol is not bounded, the problem is undecidable (Durgin *et al.*, 2004) for a reasonable model of protocols. Hence, there exists no automatic tool that always terminates and solves this problem. However, there are several approaches that can tackle an undecidable problem:

  - One can rely on help from the user. This is the approach taken for example by Isabelle (Paulson, 1998), which is an interactive theorem prover, Tamarin (Schmidt *et al.*, 2012), which just requires the user to give a few lemmas to help the tool, or Cryptyc (Gordon and Jeffrey, 2004), which relies on typing with type annotations.
  - One can have incomplete tools, which sometimes answer "I don't know" but succeed on many practical examples. For instance, one can use abstractions based on tree-automata to

represent the knowledge of the adversary (Monniaux, 2003; Boichut *et al.*, 2006).

– One can allow non-termination, as in Maude-NPA (Meadows, 1996; Escobar *et al.*, 2006).

The symbolic protocol verifier ProVerif represents protocols by Horn clauses, in the line of ideas by Weidenbach, 1999: Horn clauses are first order logical formulas, of the form $F_1 \wedge \cdots \wedge F_n \Rightarrow F$, where $F_1, \ldots, F_n, F$ are facts. This representation introduces abstractions. It is still more precise than tree-automata because it keeps relational information on messages. However, using this approach, termination is not guaranteed in general.

Let us compare ProVerif with some other tools that verify protocol specifications in the symbolic model. AVISPA (Armando *et al.*, 2005) is a platform that offers four different protocol verification back-ends: SATMC (Armando *et al.*, 2014) for bounded attack depth (which implies bounded sessions and messages), Cl-AtSe (Turuani, 2006) and OFMC (Basin *et al.*, 2005; Mödersheim and Viganò, 2009) for bounded sessions, and TA4SP (Boichut *et al.*, 2006) for unbounded sessions. In contrast, ProVerif focuses only on the case of unbounded sessions, and the Horn-clause abstraction it uses is more precise than the tree-automata abstraction of TA4SP, as mentioned above. SATMC supports basic cryptographic primitives that can be defined by rewrite rules. Cl-AtSe additionally supports exclusive or, Diffie-Hellman exponentiation (including equations of the multiplicative group modulo p), and associative concatenation. OFMC supports cryptographic primitives defined by finite equational theories (theories under which every term has a finite equivalence class) and subterm convergent theories (theories generated by rewrite rules that are convergent, that is, terminating and confluent, and whose right-hand side is either a subterm of the left-hand side or a constant). However, in order to guarantee termination, it bounds the number of instantiations of variables. TA4SP handles algebraic properties of exponentiation and exclusive or. ProVerif supports cryptographic primitives defined by rewrite rules and by equations that satisfy the finite variant property (Comon-Lundh and Delaune, 2005),

which excludes associativity. AVISPA focuses on trace properties, while ProVerif can also verify some equivalence properties.

Maude-NPA (Meadows, 1996; Escobar *et al.*, 2006) relies on narrowing in rewrite systems. It is fully automatic and supports an unbounded number of sessions, but in contrast to ProVerif, it does not make any abstraction. Hence, it is sound and complete, but may not terminate. It supports cryptographic primitives defined by convergent rewrite rules plus associativity and commutativity (Escobar *et al.*, 2007), as well as homomorphic encryption (Escobar *et al.*, 2011), while ProVerif does not support associativity nor homomorphic encryption. It initially focused on reachability properties and was recently extended to prove some equivalences (Santiago *et al.*, 2014), using the same idea as ProVerif (see §3.4).

Scyther (Cremers, 2008) is fully automatic, always terminates, and can provide three different results: verification for an unbounded number of sessions, attack, or verification for a bounded number of sessions. It supports only a fixed set of cryptographic primitives (symmetric and asymmetric encryption and signatures). It proves secrecy and authentication properties. A version named scyther-proof generates Isabelle proofs of security of the verified protocols (Meier *et al.*, 2010).

Tamarin (Schmidt *et al.*, 2012) verifies protocols for an unbounded number of sessions, but often relies on the user to provide some lemmas in order to guide the proof. It initially proved trace properties expressed in temporal first-order logic, and was recently extended to prove some equivalences (Basin *et al.*, 2015), using the same idea as ProVerif. It supports cryptographic primitives defined by subterm convergent equations, Diffie-Hellman exponentiation, bilinear pairings, and associative and commutative operators (Schmidt *et al.*, 2014). It also supports mutable state and loops; the lemmas provided by the user basically give loop invariants. Protocols in Tamarin are specified as multiset rewriting systems; Kremer and Künnemann, 2014 wrote a translator from an extension of the applied pi calculus with state.

The rest of this survey focuses on ProVerif. We refer the reader to (Blanchet, 2012b) for a more complete survey of security protocol verification.

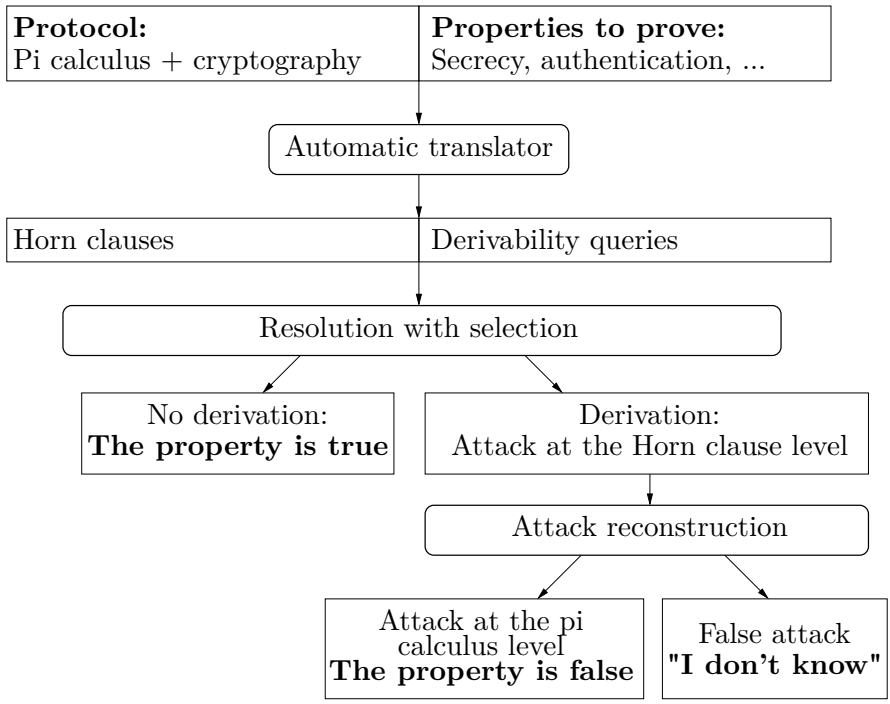

Figure 1.1: Structure of ProVerif

1.2 Structure of ProVerif

The structure of ProVerif is represented in Figure 1.1. ProVerif takes as input a model of the protocol in an extension of the pi calculus with cryptography, similar to the applied pi calculus (Abadi and Fournet, 2001; Abadi *et al.*, 2016) and detailed in the next chapter. It supports a wide variety of cryptographic primitives, modeled by rewrite rules or by equations. ProVerif also takes as input the security properties that we want to prove. It can verify various security properties, including secrecy, authentication, and some observational equivalence properties. It automatically translates this information into an internal representation by Horn clauses: the protocol is translated into a set of Horn clauses, and the security properties to prove are translated into derivability queries on these clauses. ProVerif uses an algorithm based on resolution

with free selection to determine whether a fact is derivable from the clauses. If the fact is *not* derivable, then the desired security property is proved. If the fact is derivable, then there may be an attack against the considered property: the derivation may correspond to an attack, but it may also correspond to a "false attack", because the Horn clause representation makes some abstractions. These abstractions are key to the verification of an unbounded number of sessions of protocols.

Chapter 2 presents the protocol specification language of ProVerif. Chapter 3 explains how ProVerif verifies the desired security properties. Chapter 4 relates the protocol specification language of ProVerif to the applied pi calculus (Abadi and Fournet, 2001; Abadi *et al.*, 2016). Finally, Chapter 5 summarizes some applications of ProVerif and Chapter 6 concludes.

1.3 Comparison with previous surveys

Previous surveys on ProVerif (Blanchet, 2011; Blanchet, 2014) focus only on secrecy. The general protocol verification survey Blanchet, 2012b also outlines the verification of secrecy in ProVerif. Previous journal papers present individual features of the tool: secrecy (Abadi and Blanchet, 2005a), correspondences (Blanchet, 2009), and equivalences Blanchet *et al.*, 2008. Our habilitation thesis (Blanchet, 2008b), in French, presents a general survey of ProVerif that includes secrecy, correspondences, and equivalences.

This survey is the first one to present all these features in English, in a common framework. Moreover, it includes features that never appeared in previous surveys: the extended destructors of (Cheval and Blanchet, 2013), the proof of equivalences using swapping (Blanchet and Smyth, 2016), as well as the link with the applied pi calculus (Chapter 4), which was never published before.

2

The Protocol Specification Language

This chapter presents the protocol specification language used by ProVerif, by giving its syntax and semantics. Our presentation is based on earlier ones (Abadi and Blanchet, 2005a; Blanchet, 2009; Blanchet, 2014), with some features added: extended destructors (Cheval and Blanchet, 2013), support for equations and phases (Blanchet *et al.*, 2008), enriched terms, pattern-matching, and table of keys (Blanchet *et al.*, 2016).

2.1 Core language: syntax and informal semantics

Figure 2.1 gives the syntax of *terms* (data, messages), *expressions* (computations on terms), and *processes* (programs) of the input language of ProVerif. Terms are typed. The types T include channel for channels, bool for boolean values, and bitstring for bitstrings. The user may declare other types. ProVerif checks that processes given by the user are well-typed. This is helpful in order to detect mistakes in protocol specifications. However, by default, it ignores types for the verification of security properties. This behavior allows the adversary to bypass the type system, therefore enabling the detection of type-flaw

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application
$D ::=$	expressions
M	term
$h(D_1, \ldots, D_n)$	function application
fail	failure
$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

Figure 2.1: Syntax of the core language

attacks (Heather *et al.*, 2000). For this reason, we remove types in the definition of the formal semantics and in the verification of security properties. If desired, ProVerif can also be configured to take into account types in the verification of security properties. We focus on the default configuration in this survey.

The identifiers a, b, c, k, and similar ones range over names, and x, y, and z range over variables. Names represent atomic data, such as keys and nonces. Variables can be substituted by terms. Names and variables are declared with their type. The syntax also assumes a set of function symbols for constructors and destructors; we often use f for a constructor, g for a destructor, and h for a constructor or a destructor. Function symbols are declared with types: $h(T_1, \ldots, T_n) : T$ means that the function h takes n arguments of types $T_1, \ldots, T_n$ respectively, and

returns a result of type T.

Constructors are used to build terms. Therefore, terms are variables, names, and constructor applications of the form $f(M_1, \ldots, M_n)$. For instance, symmetric encryption is typically represented by a constructor senc, and the term $\mathsf{senc}(c, k)$ represents the encryption of c under the key k. On the other hand, destructors do not appear in terms, but manipulate terms in expressions. They are functions on terms that processes can apply, via the expression evaluation construct. A destructor g is defined by a finite ordered list of rewrite rules $\mathrm{def}(g)$ of the form $g(U_1, \ldots, U_n) \to U$ where $U_1, \ldots, U_n, U$ are *may-fail terms*, that is, terms M, the constant fail, which represents the failure of a computation, or may-fail variables u. Hence, we have two kinds of variables: ordinary variables, or simply variables, which can be substituted by terms, and may-fail variables, which can be substituted by may-fail terms. In the rewrite rules, $U_1, \ldots, U_n, U$ do not contain names, and the variables of U also occur in $U_1, \ldots, U_n$. The types of $U_1, \ldots, U_n$ must be the types of the arguments of g, and the type of U must be the type of the result of g. Destructors occur in expressions D, which evaluate either to a term M or to the special constant fail. To evaluate $g(D_1, \ldots, D_n)$, we first evaluate $D_1, \ldots, D_n$, which yields $U_1, \ldots, U_n$, where each U_i is either a term M or the constant fail. Then, we consider the first rewrite rule $g(U'_1, \ldots, U'_n) \to U'$ in $\mathrm{def}(g)$. If there exists a substitution σ such that $\sigma U'_i = U_i$ for all $i \in \{1, \ldots, n\}$, then this rewrite rule applies and $g(D_1, \ldots, D_n)$ evaluates to $\sigma U'$. Otherwise, the first rewrite rule does not apply, and we consider the second one, and so on. If no rewrite rule applies, the evaluation of $g(D_1, \ldots, D_n)$ fails: it evaluates to the constant fail.

Using constructors and destructors, we can represent data structures, such as tuples, and cryptographic operations, for instance as follows:

- $\mathsf{tuple}_{T_1,\ldots,T_n}(M_1, \ldots, M_n)$ is the tuple of the terms $M_1, \ldots, M_n$, where $\mathsf{tuple}_{T_1,\ldots,T_n}$ is a constructor that takes arguments of types $T_1, \ldots, T_n$ and returns a result of type $\mathsf{bitstring}$, that is, the type declaration of $\mathsf{tuple}_{T_1,\ldots,T_n}$ is $\mathsf{tuple}_{T_1,\ldots,T_n}(T_1, \ldots, T_n) : \mathsf{bitstring}$. (We often abbreviate $\mathsf{tuple}_{T_1,\ldots,T_n}(M_1, \ldots, M_n)$ to $(M_1, \ldots, M_n)$.) The n projections are destructors $i\mathsf{th}_{T_1,\ldots,T_n}$ for $i \in \{1, \ldots, n\}$,

defined by

$$i\mathsf{th}_{T_1,\ldots,T_n}(\mathsf{tuple}_{T_1,\ldots,T_n}(x_1,\ldots,x_n)) \to x_i\,.$$

- $\mathsf{senc}(M, N)$ is the symmetric (shared-key) encryption of the message M under the key N, where $\mathsf{senc}(\mathsf{bitstring}, \mathsf{key}) : \mathsf{bitstring}$ is a constructor. The corresponding destructor $\mathsf{sdec}(\mathsf{bitstring}, \mathsf{key}) : \mathsf{bitstring}$ is defined by

$$\mathsf{sdec}(\mathsf{senc}(x, y), y) \to x\,.$$

  Thus, $\mathsf{sdec}(M', N)$ returns the decryption of M' if M' is a message encrypted under N. Otherwise, it fails.

- In order to represent asymmetric (public-key) encryption, we may use two constructors $\mathsf{pk}(\mathsf{skey}) : \mathsf{pkey}$ and $\mathsf{aenc}(\mathsf{bitstring}, \mathsf{pkey}) : \mathsf{bitstring}$, where skey is the type of secret keys and pkey is the type of public keys: $\mathsf{pk}(M)$ builds a public key from a secret key M and $\mathsf{aenc}(M, N)$ encrypts M under the public key N. The corresponding destructor $\mathsf{adec}(\mathsf{bitstring}, \mathsf{skey}) : \mathsf{bitstring}$ is defined by

$$\mathsf{adec}(\mathsf{aenc}(x, \mathsf{pk}(y)), y) \to x\,. \tag{2.1}$$

  It decrypts the ciphertext $\mathsf{aenc}(x, \mathsf{pk}(y))$ using the secret key y corresponding to the public key $\mathsf{pk}(y)$ used to encrypt this ciphertext.

- Cryptographically, a secure asymmetric encryption scheme must be probabilistic. We can represent a probabilistic encryption scheme by adding a third argument to the constructor aenc to represent the randomness. This constructor becomes $\mathsf{aenc}(\mathsf{bitstring}, \mathsf{pkey}, \mathsf{rand}) : \mathsf{bitstring}$ and the rewrite rule for adec becomes

$$\mathsf{adec}(\mathsf{aenc}(x, \mathsf{pk}(y), z), y) \to x\,. \tag{2.2}$$

  For protocols that do not test equality of ciphertexts, for secrecy and authentication, one can use the simpler, deterministic model of encryption (Cortier *et al.*, 2006). However, for equivalence properties, or for protocols that test equality of ciphertexts, using

probabilistic encryption does make a difference. Probabilistic versions can also be defined for symmetric encryption (above) and for digital signatures (below) (Blanchet *et al.*, 2016, §4.2.5); we omit such models here.

- For digital signatures, we may use two constructors $\mathsf{pk}(\mathsf{skey}) : \mathsf{pkey}$ as above and $\mathsf{sign}(\mathsf{bitstring}, \mathsf{skey}) : \mathsf{bitstring}$, where $\mathsf{sign}(M, N)$ represents the message M signed with the signature key N. The corresponding destructors $\mathsf{check}(\mathsf{bitstring}, \mathsf{pkey}) : \mathsf{bitstring}$ and $\mathsf{getmess}(\mathsf{bitstring}) : \mathsf{bitstring}$ are defined by:

$$\mathsf{check}(\mathsf{sign}(x, y), \mathsf{pk}(y)) \to x\,, \tag{2.3}$$

$$\mathsf{getmess}(\mathsf{sign}(x, y)) \to x\,. \tag{2.4}$$

  The destructor check verifies that the signature $\mathsf{sign}(x, y)$ is a correct signature under the secret key y, using the public key $\mathsf{pk}(y)$. When the signature is correct, it returns the message that was signed. The destructor getmess always returns the message that was signed. (This encoding of signatures assumes that the message that was signed can be recovered from the signature.)

- We may represent a one-way hash function by the constructor $h(\mathsf{bitstring}) : \mathsf{bitstring}$. There is no corresponding destructor; so we model that the term M cannot be retrieved from its hash $h(M)$.

- Boolean constants $\mathsf{true} : \mathsf{bool}$ and $\mathsf{false} : \mathsf{bool}$ are nullary constructors, and equality between terms may be defined as a destructor $\mathsf{equal}(T, T) : \mathsf{bool}$ with rewrite rules:

$$\begin{aligned} &\mathsf{equal}(x, x) \to \mathsf{true}\,, \\ &\mathsf{equal}(x, y) \to \mathsf{false}\,. \end{aligned} \tag{2.5}$$

  The second rewrite rule applies only when the first one does not apply, that is, when $x \neq y$. We will write $M = N$ for $\mathsf{equal}(M, N)$. Disequality as well as boolean operations (negation not, conjunction &&, disjunction ||) may also be defined as destructors. For instance, conjunction can be defined as a

destructor $\mathsf{and}(\mathsf{bool}, \mathsf{bool}) : \mathsf{bool}$ with rewrite rules:

$$\mathsf{and}(\mathsf{true}, u) \to u\,,$$
$$\mathsf{and}(x, u) \to \mathsf{false}\,.$$

This destructor returns its second argument when its first argument is true, and false when its first argument is a term different from true. (We consider all boolean terms that are not true as false.) The variable u is a may-fail variable, so that the rewrite rules apply even if the second argument fails. Therefore, $\mathsf{and}(\mathsf{false}, \mathsf{fail})$ evaluates to false. Hence, and may not fail even if its second argument fails. Intuitively, the second argument need not be evaluated when the first argument is not true. If we replaced u with an ordinary variable y, and would fail when one of its arguments fails, corresponding to the intuition that we would evaluate both arguments before computing the conjunction. We write $M \mathbin{\&\&} N$ for $\mathsf{and}(M, N)$ for readability.

We can also define an if-then-else destructor $\mathsf{ifthenelse}(\mathsf{bool}, T, T) : T$ with rewrite rules:

$$\mathsf{ifthenelse}(\mathsf{true}, u, u') \to u\,,$$
$$\mathsf{ifthenelse}(x, u, u') \to u'\,.$$

This destructor returns its second argument when its first argument is true. It returns its third argument when its first argument is a term different from true. Otherwise, that is, when its first argument fails, it fails. Similarly to the conjunction above, the variables u and u' are may-fail variables, so that the rewrite rules apply even if the second or third argument fails. As a consequence, $\mathsf{ifthenelse}(\mathsf{true}, M, \mathsf{fail})$ and $\mathsf{ifthenelse}(\mathsf{false}, \mathsf{fail}, M)$ both evaluate to M. Hence, ifthenelse may not fail even if its second or third argument fails. Intuitively, the third argument need not be evaluated when the first argument is true, and the second argument need not be evaluated when the first argument is not true.

- Type-converter functions are unary functions, whose only goal is to convert a term from a type to another, so that it has the right

type to be passed to other functions. Type-converter functions are declared with the annotation typeConverter so that ProVerif can recognize them. Since it ignores types for the verification of security properties, ProVerif removes type-converter functions at this stage.

For instance, we may define a type-converter function k2b(key) : bitstring, which converts terms from type key to type bitstring, so that keys can be encrypted or signed by the functions defined above. The inverse function b2k(bitstring) : key is defined by the rewrite rule

$$\mathsf{b2k}(\mathsf{k2b}(x)) \to x\,.$$

The function k2b always succeeds: all keys are bitstrings; while the function b2k succeeds only when its argument is of the form k2b(M), that is, the terms k2b(M) are the only terms of type bitstring that are keys and can be converted to type key.

Thus, destructors defined by rewrite rules support many of the operations common in security protocols. This way of specifying cryptographic primitives also has limitations, though: for example, modular exponentiation cannot be directly represented in this framework. To overcome this limitation, ProVerif supports equations, as explained in §2.5.1.

Most constructs of processes in the syntax of Figure 2.1 come from the pi calculus (Milner *et al.*, 1992).

- The nil process **0** does nothing.
- The input process in($N, x : T$); P inputs a message on channel N, and executes P with x bound to the input message.

  The output process out(N, M); P outputs the message M on the channel N and then executes P.

  Here, we use an arbitrary term N to represent a channel: N can be a name, a variable, or a constructor application. The calculus is monadic (in that the messages are terms rather than tuples of terms), but a polyadic calculus can be simulated since tuples are terms. It is also synchronous (in that a process P is executed after the output of a message). We may omit P when it is **0**.

- The process $P \mid Q$ is the parallel composition of P and Q.
- The replication $!P$ represents an unbounded number of copies of P in parallel. It makes it possible to represent an unbounded number of executions of the protocol.
- The restriction $\mathsf{new}\ a : T; P$ creates a new name a of type T, and then executes P. It can model the creation of a fresh key or nonce.
- The process $\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ tries to evaluate D; if D evaluates to a term M, then x is bound to M and P is executed; if the evaluation of D fails, then Q is executed. The type T may be omitted when it can be determined from D, that is, when D is not fail.
- The conditional $\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$ executes P if M is true (or is a variable bound to true); it executes Q if M is different from true.

We may omit an else branch when it consists of $\mathbf{0}$.

The name a is bound in the process $\mathsf{new}\ a : T; P$. The variable x is bound in P in the processes $\mathsf{in}(N, x : T); P$ and $\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$. Processes are considered equal modulo renaming of bound names and variables. We write $fn(P)$ and $fv(P)$ for the sets of names and variables free in P, respectively. A process is closed if it has no free variables; it may have free names. The protocol to verify is represented by a closed process P_0. Free names are declared with their type. They may be public or private: the adversary initially has access to the public free names, but not to the private ones. We write $\mathcal{N}_{\mathrm{pub}}$ for the set of public free names and $\mathcal{N}_{\mathrm{priv}}$ for the set of private free names, so we have $fn(P_0) \subseteq \mathcal{N}_{\mathrm{pub}} \cup \mathcal{N}_{\mathrm{priv}}$. (One may declare free names not present in P_0, but used for instance in security queries.) Similarly, function symbols may be public or private: the adversary has access to the public function symbols, but not to the private ones.

We write $\{^{M_1}/_{x_1}, \ldots, ^{M_n}/_{x_n}\}$ for the substitution that replaces x_1, $\ldots$, x_n with $M_1, \ldots, M_n$, respectively. When D is some term, expression, or process, we write $D\{^{M_1}/_{x_1}, \ldots, ^{M_n}/_{x_n}\}$ for the result of applying this substitution to D, but we write σD when the substitution is

simply denoted σ. Except when stated otherwise, substitutions always map variables (not names) to terms. Furthermore, substitutions never substitute fail or a may-fail variable for an ordinary variable.

The calculus of ProVerif resembles the applied pi calculus (Abadi and Fournet, 2001; Abadi *et al.*, 2016). Both calculi are extensions of the pi calculus with (fairly arbitrary) functions on terms. However, there are also important differences between these calculi. The first one is that ProVerif uses destructors in addition to the equational theories of the applied pi calculus, but does not support all equational theories. (§ 2.5.1 explains how ProVerif handles equations.) The second difference is that ProVerif has a built-in error-handling construct (the else branch of the expression evaluation), whereas in the applied pi calculus the error-handling must be done "by hand". For instance, in ProVerif, the process $\mathsf{let}\ x = \mathsf{sdec}(M, k)\ \mathsf{in}\ P\ \mathsf{else}\ \mathsf{out}(c, \mathsf{error})$ outputs the error message when the decryption of M fails. In the applied pi calculus, decryption always succeeds, but may return junk, and one has to perform an additional test if one wants to check that M is a ciphertext under the key k. Chapter 4 provides a more detailed comparison with the applied pi calculus.

2.2 An example of protocol

We use as a running example a simplified version of the Denning-Sacco key distribution protocol (Denning and Sacco, 1981), omitting certificates and timestamps:

$$\begin{array}{ll} \text{Message 1.} & A \to B : \mathsf{aenc}(\mathsf{sign}(\mathsf{k2b}(k), ssk_A), pk_B) \\ \text{Message 2.} & B \to A : \mathsf{senc}(s, k) \end{array}$$

This protocol is illustrated in Figure 2.2. It involves two principals A and B. The key ssk_A is the secret, signature key of A and spk_A its public, signature verification key. (We add an s prefix for signature keys to distinguish them from encryption keys.) The key sk_B is the secret, decryption key of B and pk_B its public, encryption key. The key k is a fresh session key created by A. A sends this key, signed with its private key ssk_A, and encrypted under the public key of B, pk_B. The key k is first converted to a bitstring by k2b, so that it has the right type for

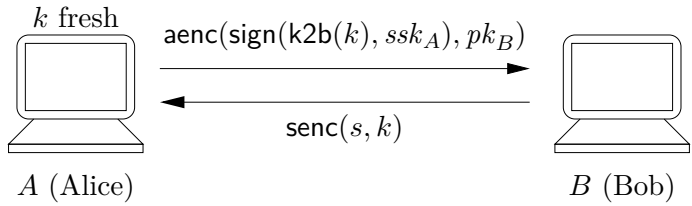

Figure 2.2: An example of protocol

signing it. When B receives this message, B decrypts it and assumes, seeing the signature, that the key k has been generated by A. Then B sends a secret s encrypted under k. Only A should be able to decrypt the message and get the secret s. (The second message is not really part of the protocol; we use it to check if the key k can be used to exchange secrets between A and B. In fact, there is an attack against this protocol (Abadi and Needham, 1996), so s will not remain secret. The attack is explained in Example 3.2.)

This protocol can be encoded by the following process:

$$
\begin{aligned}
P_0 = {} & \mathsf{new}\ ssk_A : \mathsf{skey}; \mathsf{new}\ sk_B : \mathsf{skey}; \mathsf{let}\ spk_A = \mathsf{pk}(ssk_A)\ \mathsf{in} \\
& \mathsf{let}\ pk_B = \mathsf{pk}(sk_B)\ \mathsf{in}\ \mathsf{out}(c, spk_A); \mathsf{out}(c, pk_B); \\
& (P_A(ssk_A, pk_B) \mid P_B(sk_B, spk_A)) \\
P_A(ssk_A, pk_B) = {} & !\ \mathsf{new}\ k : \mathsf{key}; \\
& \mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}(\mathsf{k2b}(k), ssk_A), pk_B)); \\
& \mathsf{in}(c, x : \mathsf{bitstring}); \mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ \mathbf{0} \\
P_B(sk_B, spk_A) = {} & !\ \mathsf{in}(c, y : \mathsf{bitstring}); \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in} \\
& \mathsf{let}\ x_k = \mathsf{b2k}(\mathsf{check}(y', spk_A))\ \mathsf{in}\ \mathsf{out}(c, \mathsf{senc}(s, x_k))
\end{aligned}
$$

Such a process can be given as input to ProVerif. This process first creates the secret keys ssk_A and sk_B, computes the corresponding public keys spk_A and pk_B, and sends these keys on the public channel c, so that the adversary has these public keys. Then, it runs the processes P_A and P_B in parallel. These processes correspond respectively to the roles of A and B in the protocol. They both start with a replication, which makes it possible to model an unbounded number of sessions of the protocol.

The process P_A executes the role of A: it creates a fresh key k, converts it to a bitstring by $\mathsf{k2b}$, signs it with its secret key ssk_A, then encrypts this message under pk_B, and sends the obtained message on channel c. P_A then expects the second message of the protocol on channel c, stores it in x and decrypts it. If decryption succeeds, the result (normally the secret s) is stored in z.

The process P_B receives the first message of the protocol on channel c, stores it in y, decrypts it with sk_B, and verifies the signature with spk_A. If these verifications succeed, B believes that x_k is a key shared between A and B, and it sends the secret s encrypted under x_k. If the protocol is correct, s should remain secret. In this example, there are two free names, c and s; c is public and s is private, so $\mathcal{N}_{\text{pub}} = \{c\}$ and $\mathcal{N}_{\text{priv}} = \{s\}$.

This model of the protocol is weak, because A and B talk only to each other: they do not interact with other, possibly dishonest participants. We can strengthen the model as follows. We replace the process P_A with the following process:

$$\begin{aligned} P_A(ssk_A, pk_B) = {} & !\,\mathsf{in}(c, x_{pk_B} : \mathsf{pkey}); \mathsf{new}\ k : \mathsf{key}; \\ & \mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}(\mathsf{k2b}(k), ssk_A), x_{pk_B})); \\ & \mathsf{in}(c, x : \mathsf{bitstring}); \mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ \mathbf{0} \end{aligned}$$

This process P_A first receives on the public channel c the key x_{pk_B}, which is the public key of A's interlocutor in the protocol. This message is not part of the protocol; it allows the adversary to choose with whom A is going to execute a session. In a standard session of the protocol, this key is pk_B, but the adversary can also choose another key, for instance one of his own keys, so that A can interact with the adversary playing the role of a dishonest participant. Then P_A executes the role of A as before, with the key x_{pk_B} instead of pk_B. The process P_B does not need to be modified because B sends the second message $\mathsf{senc}(s, k)$ only if its interlocutor is the honest participant A. (Otherwise, the secret would obviously be leaked.) Hence the signature is verified with the key spk_A of A and not with an arbitrary key chosen by the adversary.

The above model still assumes for simplicity that A and B each play only one role of the protocol. One could easily write an even more

general model in which they play both roles, or one could provide the adversary with an interface that allows it to dynamically create new protocol participants. We consider such a model in §2.5.5.

2.3 Core language: type system

Processes must be well-typed in the type system of Figure 2.3. For simplicity, all bound variables and names are renamed so that they are pairwise distinct and distinct from free names. The type system uses a type environment Γ that maps variables and names to their type. This type environment initially contains the types of the free names of the closed process under consideration. The type system defines three judgments: the judgment $\Gamma \vdash M : T$ means that the term M is well-typed of type T in the type environment Γ; the judgment $\Gamma \vdash D : T$ means that the expression D is well-typed of type T in the type environment Γ; and the judgment $\Gamma \vdash P$ means that the process P is well-typed in the type environment Γ.

The typing rules are straightforward. The type of variables and names is obtained from the type environment Γ. Function applications $h(D_1, \ldots, D_n)$ are well-typed when their arguments are well-typed, of the type expected by the function h, and the type of $h(D_1, \ldots, D_n)$ is the type of the result of h. The constant fail can be of any type. The rules require that input and output channels be of type $\mathsf{channel}$, that output messages be well-typed, and that conditions be of type bool.

Example 2.1. It is easy to see that the process P_0 of §2.2 is well-typed in the type environment $\Gamma = c : \mathsf{channel}, s : \mathsf{bitstring}$, that is, $\Gamma \vdash P_0$.

As mentioned in §2.1, ProVerif checks that processes given by the user are well-typed, but it ignores types and removes type-converter functions for the verification of security properties.

Example 2.2. With types and type-converter functions removed, the process P_0 of §2.2 (version in which A may interact with dishonest participants) becomes:

$$P_0 = \mathsf{new}\ ssk_A; \mathsf{new}\ sk_B; \mathsf{let}\ spk_A = \mathsf{pk}(ssk_A)\ \mathsf{in}\ \mathsf{let}\ pk_B = \mathsf{pk}(sk_B)\ \mathsf{in}$$
$$\mathsf{out}(c, spk_A); \mathsf{out}(c, pk_B); (P_A(ssk_A, pk_B) \mid P_B(sk_B, spk_A))$$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{(a : T) \in \Gamma}{\Gamma \vdash a : T}$$

$$\frac{f(T_1, \ldots, T_n) : T \quad \Gamma \vdash M_1 : T_1 \quad \ldots \quad \Gamma \vdash M_n : T_n}{\Gamma \vdash f(M_1, \ldots, M_n) : T}$$

$$\frac{h(T_1, \ldots, T_n) : T \quad \Gamma \vdash D_1 : T_1 \quad \ldots \quad \Gamma \vdash D_n : T_n}{\Gamma \vdash h(D_1, \ldots, D_n) : T}$$

$$\Gamma \vdash \mathsf{fail} : T$$

$$\frac{\Gamma \vdash N : \mathsf{channel} \quad \Gamma \vdash M : T \quad \Gamma \vdash P}{\Gamma \vdash \mathsf{out}(N, M); P}$$

$$\frac{\Gamma \vdash N : \mathsf{channel} \quad \Gamma, x : T \vdash P}{\Gamma \vdash \mathsf{in}(N, x : T); P}$$

$$\Gamma \vdash \mathbf{0} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P} \qquad \frac{\Gamma, a : T \vdash P}{\Gamma \vdash \mathsf{new}\ a : T; P}$$

$$\frac{\Gamma \vdash D : T \quad \Gamma, x : T \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q}$$

$$\frac{\Gamma \vdash M : \mathsf{bool} \quad \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

Figure 2.3: Type system

$$P_A(ssk_A, pk_B) = !\,\mathsf{in}(c, x_{pk_B}); \mathsf{new}\ k;$$
$$\quad \mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}(k, ssk_A), x_{pk_B})); \mathsf{in}(c, x); \mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ \mathbf{0}$$
$$P_B(sk_B, spk_A) = !\,\mathsf{in}(c, y); \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in}$$
$$\quad \mathsf{let}\ x_k = \mathsf{check}(y', spk_A)\ \mathsf{in}\ \mathsf{out}(c, \mathsf{senc}(s, x_k))$$

ProVerif verifies this process when it is given the process of §2.2.

2.4 Core language: formal semantics

The formal semantics of this language is defined in Figure 2.4. The definition proceeds in two steps. First, we define the semantics of

$M \Downarrow M$
$\mathsf{fail} \Downarrow \mathsf{fail}$
$h(D_1, \ldots, D_n) \Downarrow \sigma U'_j$ if and only if
$D_1 \Downarrow U_1, \ldots, D_n \Downarrow U_n$,
$\mathrm{def}(h)$ consists of the rewrite rules
$h(U'_{i,1}, \ldots, U'_{i,n}) \to U'_i$ for $i \in \{1, \ldots, k\}$,
$\sigma U'_{j,1} = U_1, \ldots, \sigma U'_{j,n} = U_n$, and
for all $i \leq j$, for all σ', $\sigma' U'_{i,1} \neq U_1, \ldots, \sigma' U'_{i,n} \neq U_n$.

$$E, \mathcal{P} \cup \{\, \mathbf{0} \,\} \to E, \mathcal{P} \qquad \text{(Red Nil)}$$

$$E, \mathcal{P} \cup \{\, P \mid Q \,\} \to E, \mathcal{P} \cup \{\, P, Q \,\} \qquad \text{(Red Par)}$$

$$E, \mathcal{P} \cup \{\, !P \,\} \to E, \mathcal{P} \cup \{\, P, !P \,\} \qquad \text{(Red Repl)}$$

$$(\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}), \mathcal{P} \cup \{\, \mathsf{new}\ a; P \,\} \to (\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}} \cup \{a'\}), \mathcal{P} \cup \{\, P\{^{a'}/_{a}\} \,\}$$
$$\text{where } a' \notin \mathcal{N}_{\mathrm{pub}} \cup \mathcal{N}_{\mathrm{priv}} \qquad \text{(Red Res)}$$

$$E, \mathcal{P} \cup \{\, \mathsf{out}(N, M); Q, \mathsf{in}(N, x); P \,\} \to E, \mathcal{P} \cup \{\, Q, P\{^{M}/_{x}\} \,\} \qquad \text{(Red I/O)}$$

$$E, \mathcal{P} \cup \{\, \mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \,\} \to E, \mathcal{P} \cup \{\, P\{^{M}/_{x}\} \,\}$$
$$\text{if } D \Downarrow M \qquad \text{(Red Eval 1)}$$

$$E, \mathcal{P} \cup \{\, \mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \,\} \to E, \mathcal{P} \cup \{\, Q \,\}$$
$$\text{if } D \Downarrow \mathsf{fail} \qquad \text{(Red Eval 2)}$$

$$E, \mathcal{P} \cup \{\, \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ P\ \mathsf{else}\ Q \,\} \to E, \mathcal{P} \cup \{\, P \,\} \qquad \text{(Red Cond 1)}$$

$$E, \mathcal{P} \cup \{\, \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q \,\} \to E, \mathcal{P} \cup \{\, Q \,\}$$
$$\text{if } M \neq \mathsf{true} \qquad \text{(Red Cond 2)}$$

Figure 2.4: Operational semantics

expressions: the relation $D \Downarrow U$ means that the closed expression D evaluates to the closed may-fail term U, which may be a closed term M or the constant fail. To ease the definition of this relation, we associate to each constructor f of arity n a list of rewrite rules $\mathrm{def}(f)$ that contains the identity rewrite rule

$$f(x_1, \ldots, x_n) \to f(x_1, \ldots, x_n)\,.$$

Furthermore, for each constructor or destructor h of arity n, we add the following rewrite rule as final rewrite rule of $\mathrm{def}(h)$:

$$h(u_1, \ldots, u_n) \to \mathsf{fail}\,, \tag{2.6}$$

where $u_1, \ldots, u_n$ are may-fail variables. This rewrite rule expresses that the function h returns fail when no other rewrite rule applies. In particular, constructors return fail when one of their argument fails. In Figure 2.4, the first two rules of the definition of $\Downarrow$ express that a closed term and fail evaluate to themselves; the third rule deals with function application. It first evaluates the arguments of the function $D_1, \ldots, D_n$ to $U_1, \ldots, U_n$ respectively. Then, it applies the j-th rewrite rule of h, $h(U'_{j,1}, \ldots, U'_{j,n}) \to U'_j$, instantiated with the substitution σ, so $h(U_1, \ldots, U_n) = h(\sigma U'_{j,1}, \ldots, \sigma U'_{j,n})$ reduces into $\sigma U'_j$. The last line checks that the rewrite rules before the j-th cannot be applied.

Example 2.3. For instance, we have

$$\mathsf{adec}(\mathsf{aenc}(\mathsf{sign}(k, ssk_A), \mathsf{pk}(sk_B)), sk_B) \Downarrow \mathsf{sign}(k, ssk_A)$$

by the rewrite rule (2.1). Hence

$$\mathsf{check}(\mathsf{adec}(\mathsf{aenc}(\mathsf{sign}(k, ssk_A), \mathsf{pk}(sk_B)), sk_B), \mathsf{pk}(ssk_A)) \Downarrow k$$

by the rewrite rule (2.3). On the other hand,

$$\mathsf{adec}(\mathsf{aenc}(\mathsf{sign}(k, ssk_A), \mathsf{pk}(sk_B)), ssk_A) \Downarrow \mathsf{fail}$$

by the rewrite rule (2.6). (The rewrite rule (2.1) cannot be applied because the decryption key does not match the encryption key.) Hence,

$$\mathsf{check}(\mathsf{adec}(\mathsf{aenc}(\mathsf{sign}(k, ssk_A), \mathsf{pk}(sk_B)), ssk_A), \mathsf{pk}(ssk_A)) \Downarrow \mathsf{fail}$$

by the rewrite rule (2.6) again.

Second, we define the semantics of processes, by reduction of semantic configurations. A semantic configuration is a pair $E, \mathcal{P}$ where the environment E is a pair of two finite sets of names $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}})$ and $\mathcal{P}$ is a finite multiset of closed processes. The set $\mathcal{N}_{\text{pub}}$ contains the public names, the set $\mathcal{N}_{\text{priv}}$ contains the private names, and the multiset of processes $\mathcal{P}$ contains the processes currently running.[1] The configuration $(\{a_1, \ldots, a_n\}, \{b_1, \ldots, b_m\}), \{P_1, \ldots, P_k\}$ corresponds intuitively to the process $\mathsf{new}\ b_1; \ldots \mathsf{new}\ b_m; (P_1 \mid \ldots \mid P_k)$. (Recall that types are ignored, so we need not type the names $b_1, \ldots, b_m$.) A configuration $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}$ is *valid* when $\mathcal{N}_{\text{pub}}$ and $\mathcal{N}_{\text{priv}}$ are disjoint and $fn(\mathcal{P}) \subseteq \mathcal{N}_{\text{pub}} \cup \mathcal{N}_{\text{priv}}$. We only consider valid configurations. The reduction relation $\rightarrow$ on semantic configurations is defined in Figure 2.4. The reduction rules define the semantics of each language construct. The rule (Red Nil) removes processes $\mathbf{0}$, since they do nothing. The rule (Red Par) expands parallel compositions. The rule (Red Repl) creates an additional copy of a replicated process; since this rule can be applied again on the resulting configuration, it allows creating an unbounded number of copies of the replicated process. The rule (Red Res) creates a fresh name a', substitutes it for a, and adds it to the private names $\mathcal{N}_{\text{priv}}$. The fresh name a' is required to occur neither in $\mathcal{N}_{\text{priv}}$, which contains the initial private free names as well as any fresh name created by a previous application of (Red Res), nor in $\mathcal{N}_{\text{pub}}$, which contains the public free names. The rule (Red I/O) allows communication between processes. The message M is sent by the output and received by the input in the variable x, provided the output and input channels are equal. The rules (Red Eval 1) and (Red Eval 2) define the semantics of expression evaluations. They evaluate D. In case of success, (Red Eval 1) runs P with the result M of the evaluation substituted for x. In case of failure, (Red Eval 2) runs Q. The rules (Red Cond 1) and (Red Cond 2) define the semantics of conditionals. When M is true, (Red Cond 1) runs P. When M is different from true, (Red Cond 2) runs Q.

[1] In (Blanchet, 2009), the environment E is simply a set of names, corresponding to $\mathcal{N}_{\text{pub}} \cup \mathcal{N}_{\text{priv}}$. Indeed, the distinction between public and private names is not essential for trace properties. It is useful for treating equivalences in §3.4.

The process P_0, which represents the protocol to verify, is usually run in parallel with an adversary Q. In this case, the initial configuration is $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \{P_0, Q\}$.

Example 2.4. Let us consider the process P_0 of Example 2.2. This process cannot run alone since it sends messages on channel c without any process to receive them. Hence, we must run P_0 in parallel with an adversary Q. As an example, we consider the following adversary:

$$Q = \mathsf{in}(c, x_{spk_A}); \mathsf{in}(c, x_{pk_B}); \mathsf{out}(c, x_{pk_B}) \,,$$

which receives the two public keys, and sends the B's public key to A, so that A runs a session with B. This system runs a correct session of the protocol between A and B as follows:

$$\begin{aligned}
&(\{c\}, \{s\}), \{P_0, Q\} \\
&\to^* E_1, \{\mathsf{out}(c, \mathsf{pk}(ssk_A)); \mathsf{out}(c, \mathsf{pk}(sk_B)); \\
&\qquad (P_A(ssk_A, \mathsf{pk}(sk_B)) \mid P_B(sk_B, \mathsf{pk}(ssk_A))), Q\} \\
&\qquad\qquad \text{by (Red Res) twice and (Red Eval 1) twice} \\
&\to^* E_1, \{P_A(ssk_A, \mathsf{pk}(sk_B)), P_B(sk_B, \mathsf{pk}(ssk_A)), \\
&\qquad \mathsf{out}(c, \mathsf{pk}(sk_B))\} \qquad \text{by (Red I/O) twice and (Red Par)} \\
&\to^* E_1, \mathcal{P} \cup \{P'_A, P'_B, \mathsf{out}(c, \mathsf{pk}(sk_B))\} \qquad \text{by (Red Repl) twice}
\end{aligned}$$

where

$$\begin{aligned}
E_1 &= (\{c\}, \{s, ssk_A, sk_B\}) \\
\mathcal{P} &= \{P_A(ssk_A, \mathsf{pk}(sk_B)), P_B(sk_B, \mathsf{pk}(ssk_A))\} \\
P'_A &= \mathsf{in}(c, x_{pk_B}); \mathsf{new}\ k; \mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}(k, ssk_A), x_{pk_B})); P''_A \\
P''_A &= \mathsf{in}(c, x); \mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ \mathbf{0} \\
P'_B &= \mathsf{in}(c, y); \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in} \\
&\qquad \mathsf{let}\ x_k = \mathsf{check}(y', \mathsf{pk}(ssk_A))\ \mathsf{in}\ \mathsf{out}(c, \mathsf{senc}(s, x_k)) \,.
\end{aligned}$$

By (Red I/O), P'_A reduces with $\mathsf{out}(c, \mathsf{pk}(sk_B))$; by (Red Nil), we remove the omitted final $\mathbf{0}$. With $E_2 = (\{c\}, \{s, ssk_A, sk_B, k\})$, we obtain

$$E_1, \mathcal{P} \cup \{\mathsf{new}\ k; \mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}(k, ssk_A), \mathsf{pk}(sk_B))); P''_A, P'_B\}$$

$\to E_2, \mathcal{P} \cup \{\mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}(k, ssk_A), \mathsf{pk}(sk_B))); P''_A, P'_B\}$
by (Red Res) (A creates the fresh key k)

$\to E_2, \mathcal{P} \cup \{P''_A,$
$\quad \mathsf{let}\ y' = \mathsf{adec}(\mathsf{aenc}(\mathsf{sign}(k, ssk_A), \mathsf{pk}(sk_B)), sk_B)\ \mathsf{in}$
$\quad \mathsf{let}\ x_k = \mathsf{check}(y', \mathsf{pk}(ssk_A))\ \mathsf{in}\ \mathsf{out}(c, \mathsf{senc}(s, x_k))\}$
by (Red I/O) (A sends message 1 to B)

$\to^* E_2, \mathcal{P} \cup \{P''_A, \mathsf{out}(c, \mathsf{senc}(s, k))\}$ by (Red Eval 1) twice

$\to E_2, \mathcal{P} \cup \{\mathsf{let}\ z = \mathsf{sdec}(\mathsf{senc}(s, k), k)\ \mathsf{in}\ \mathbf{0}, \mathbf{0}\}$
by (Red I/O) (B sends message 2 to A)

$\to E_2, \mathcal{P} \cup \{\mathbf{0}, \mathbf{0}\}$ by (Red Eval 1)

$\to^* E_2, \mathcal{P}$ by (Red Nil) twice

There exist other approaches to define the semantics of such a language. We discuss one such approach in Chapter 4.

2.5 Extensions

This section presents the main extensions of the core language implemented in ProVerif, and gives their operational semantics, by extending the semantics given in §2.4.

2.5.1 Equations

In addition to cryptographic primitives defined by rewrite rules, ProVerif also supports primitives defined by equations. We consider an equational theory $\mathcal{E}$ that consists of a finite set of equations $M = N$ between terms M, N without names and of the same type. Equality modulo the equational theory $\mathcal{E}$ is obtained from these equations by reflexive, symmetric, and transitive closure, closure under application of function symbols, and closure under substitution of terms for variables. We write $M =_{\mathcal{E}} N$ an equality modulo $\mathcal{E}$, and $M \neq_{\mathcal{E}} N$ a disequality modulo $\mathcal{E}$. We only consider non-trivial equational theories, that is, for each type T, there exist terms M and N of type T such that $M \neq_{\mathcal{E}} N$.

As an example, the Diffie-Hellman key agreement (Diffie and Hellman, 1976) can be modeled using equations. The Diffie-Hellman key

agreement relies on the following property of modular exponentiation: $(g^a)^b = (g^b)^a = g^{ab}$ in a cyclic multiplicative subgroup G of $\mathbb{Z}_p^*$, where p is a large prime number and g is a generator of G, and on the assumption that it is difficult to compute g^{ab} from g^a and g^b, without knowing the random numbers a and b (computational Diffie-Hellman assumption), or on the stronger assumption that it is difficult to distinguish g^a, g^b, g^{ab} from g^a, g^b, g^c without knowing the random numbers a, b, and c (decisional Diffie-Hellman assumption). These properties are exploited to establish a shared key between two participants A and B of a protocol: A chooses randomly a and sends g^a to B; symmetrically, B chooses randomly b and sends g^b to A. A can then compute $(g^b)^a$, since it has a and receives g^b, while B computes $(g^a)^b$. These two values are equal, so they can be used to compute the shared key. The adversary, on the other hand, has g^a and g^b but not a and b so by the computational Diffie-Hellman assumption, it cannot compute the key. (This exchange resists passive attacks only; to resist active attacks, we need additional ingredients, for instance signatures.) We can model the Diffie-Hellman key agreement by the equation (Abadi and Fournet, 2001; Abadi *et al.*, 2007)

$$\mathsf{exp}(\mathsf{exp}(\mathsf{g}, x), y) = \mathsf{exp}(\mathsf{exp}(\mathsf{g}, y), x) \tag{2.7}$$

where $\mathsf{g} : \mathsf{G}$ is a constant and $\mathsf{exp}(\mathsf{G}, \mathsf{Z}) : \mathsf{G}$ is modular exponentiation. Obviously, this is a basic model: it models the main functional equation but misses many algebraic relations that exist in the group G.

We can also model a symmetric encryption scheme in which decryption always succeeds (but may return a meaningless message) by the equations

$$\begin{aligned} \mathsf{sdec}(\mathsf{senc}(x, y), y) &= x \\ \mathsf{senc}(\mathsf{sdec}(x, y), y) &= x \end{aligned} \tag{2.8}$$

where $\mathsf{senc}(\mathsf{bitstring}, \mathsf{key}) : \mathsf{bitstring}$ and $\mathsf{sdec}(\mathsf{bitstring}, \mathsf{key}) : \mathsf{bitstring}$ are constructors. In this model, decryption always succeeds, because $\mathsf{sdec}(M, N)$ is always a term, even when M is not of the form $\mathsf{senc}(M', N)$. The first equation is standard; the second one avoids that the equality test $\mathsf{senc}(\mathsf{sdec}(M, N), N) = M$ reveals that M is a ciphertext under N: in the presence of the second equation, this equality always

holds, even when M is not a ciphertext under N. These equations are satisfied by block ciphers, which are bijective.

Equations allow one to model cryptographic primitives that cannot be modeled by destructors with rewrite rules. However, rewrite rules are easier to handle for ProVerif, so they should be preferred when they are sufficient to express the desired properties.

The formal semantics of the language is extended to equations as follows. We extend the evaluation of expressions by considering equality modulo $\mathcal{E}$:

$h(D_1, \ldots, D_n) \Downarrow \sigma U'_j$ if and only if
$D_1 \Downarrow U_1, \ldots, D_n \Downarrow U_n$,
$\mathrm{def}(h)$ consists of the rewrite rules
$\quad h(U'_{i,1}, \ldots, U'_{i,n}) \to U'_i$ for $i \in \{1, \ldots, k\}$,
$\sigma U'_{j,1} =_{\mathcal{E}} U_1, \ldots, \sigma U'_{j,n} =_{\mathcal{E}} U_n$, and
for all $i \leq j$, for all σ', $\sigma' U'_{i,1} \neq_{\mathcal{E}} U_1, \ldots, \sigma' U'_{i,n} \neq_{\mathcal{E}} U_n$.

We also extend the reduction rules by considering equality modulo $\mathcal{E}$:

$$E, \mathcal{P} \cup \{ \mathsf{out}(N, M); Q, \mathsf{in}(N', x); P \} \to E, \mathcal{P} \cup \{ Q, P\{^M/_x\} \} \quad \text{if } N =_{\mathcal{E}} N' \qquad \text{(Red I/O}')$$

$$E, \mathcal{P} \cup \{ \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q \} \to E, \mathcal{P} \cup \{ P \} \quad \text{if } M =_{\mathcal{E}} \mathsf{true} \qquad \text{(Red Cond 1}')$$

$$E, \mathcal{P} \cup \{ \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q \} \to E, \mathcal{P} \cup \{ Q \} \quad \text{if } M \neq_{\mathcal{E}} \mathsf{true} \qquad \text{(Red Cond 2}')$$

To handle equations, ProVerif translates them into a set of rewrite rules associated to constructors. For instance, the equations (2.8) are translated into the rewrite rules

$$\begin{array}{ll} \mathsf{senc}(x, y) \to \mathsf{senc}(x, y) & \mathsf{sdec}(x, y) \to \mathsf{sdec}(x, y) \\ \mathsf{senc}(\mathsf{sdec}(x, y), y) \to x & \mathsf{sdec}(\mathsf{senc}(x, y), y) \to x \end{array} \qquad (2.9)$$

while the equation (2.7) is translated into

$$\mathsf{exp}(x, y) \to \mathsf{exp}(x, y) \qquad \mathsf{exp}(\mathsf{exp}(\mathsf{g}, x), y) \to \mathsf{exp}(\mathsf{exp}(\mathsf{g}, y), x) \qquad (2.10)$$

Intuitively, these rewrite rules allow one, by applying them *exactly once* for each constructor, to obtain the various forms of the terms modulo

the considered equational theory.[2] These forms are named variants, and since the number of rewrite rules must be finite, ProVerif only supports equational theories that have the finite variant property (Comon-Lundh and Delaune, 2005). Constructors are then evaluated similarly to destructors. The only difference is that the rewrite rules that come from equations are not ordered: all applicable rewrite rules are applied. With Abadi and Fournet, we have formally defined when a set of rewrite rules models an equational theory, and designed algorithms that translate equations into rewrite rules that model them (Blanchet *et al.*, 2008, §5). Then, each trace in the calculus with equational theory corresponds to a trace in the calculus with rewrite rules, and conversely (Blanchet *et al.*, 2008, Lemma 1).[3] Hence, we reduce to the calculus with rewrite rules. The main advantage of this approach is that resolution can still use ordinary syntactic unification (instead of having to use unification modulo the equational theory), and therefore remains efficient: it avoids the explosion of the number of clauses that occurs when many unifiers modulo

This approach still has limitations: associative operations, such as exclusive or, are not supported, because they would require an infinite number of rewrite rules. It may be possible to handle these operations using unification modulo the equational theory instead of syntactic unification, at the cost of a larger complexity. In the case of a bounded number of sessions, exclusive or is handled in (Comon-Lundh and Shmatikov, 2003; Chevalier *et al.*, 2005) and a more complete theory of modular exponentiation is handled in (Chevalier *et al.*, 2003). A unification algorithm for modular exponentiation is presented in (Meadows and Narendran, 2002). For an unbounded number of sessions, extensions of the Horn clause

[2]The rewrite rules like $\mathsf{sdec}(x, y) \to \mathsf{sdec}(x, y)$ are necessary so that sdec always succeeds when its arguments succeed. Thanks to this rule, the evaluation of $\mathsf{sdec}(M, N)$ succeeds and leaves this term unchanged when M is not of the form $\mathsf{senc}(M', N)$.

[3]More precisely, the disequality tests in $D \Downarrow U$ and (Red Cond 2′) must still be performed modulo the equational theory, even in the calculus with rewrite rules. The impact on the performance of the resolution algorithm is limited because disequalities are less common than other facts and, when there are several unifiers modulo the equational theory in a disequality, that leads to several disequalities in a single clause, not to several clauses.

approach have been proposed. Küsters and Truderung, 2008 support exclusive or provided one of its two arguments is a constant in the clauses that model the protocol. Küsters and Truderung, 2009 support Diffie-Hellman key agreements with more detailed algebraic relations (including equations of the multiplicative group modulo p), provided the exponents are constants in the clauses that model the protocol. By extending that line of research, Pankova and Laud, 2012 support bilinear pairings, provided the exponents are constants in the clauses that model the protocol. These approaches proceed by transforming the initial clauses into richer clauses on which the standard resolution algorithm is applied. The tools Maude-NPA (Meadows, 1996; Escobar *et al.*, 2006) and Tamarin (Schmidt *et al.*, 2012) support equational theories that ProVerif does not support, at the cost of a more costly verification or by requiring user intervention. Maude-NPA supports equations defined by convergent rewrite rules plus associativity and commutativity (Escobar *et al.*, 2007) (which includes Diffie-Hellman exponentiation and exclusive or), as well as homomorphic encryption (Escobar *et al.*, 2011). Tamarin supports subterm convergent equations, Diffie-Hellman exponentiation (including equations of the multiplicative group modulo p), bilinear pairings, and associative and commutative operators (Schmidt *et al.*, 2014).

2.5.2 Enriched terms

ProVerif allows all occurrences of terms M in processes (input and output channels, output messages, and conditions) to be replaced with expressions D, which may contain destructors, hence the syntax of processes becomes:

$P, Q ::=$	processes
...	
$\mathsf{out}(D, D'); P$	output
$\mathsf{in}(D, x : T); P$	input
$\mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

Each such expression D is evaluated; when the evaluation fails, the process does nothing; when it succeeds with result M, it executes the

process with M instead of D. Hence the formal semantics can be defined as follows:

$$E, \mathcal{P} \cup \{ \mathsf{out}(D', D); Q, \mathsf{in}(D'', x); P \} \rightarrow E, \mathcal{P} \cup \{ Q, P\{^{M}/_{x}\} \}$$
$$\text{if } D \Downarrow M,\ D' \Downarrow M',\ D'' \Downarrow M'', \text{ and } M' =_{\mathcal{E}} M'' \qquad \text{(Red I/O}'')$$

$$E, \mathcal{P} \cup \{ \mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q \} \rightarrow E, \mathcal{P} \cup \{ P \}$$
$$\text{if } D \Downarrow M \text{ and } M =_{\mathcal{E}} \mathsf{true} \qquad \text{(Red Cond 1}'')$$

$$E, \mathcal{P} \cup \{ \mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q \} \rightarrow E, \mathcal{P} \cup \{ Q \}$$
$$\text{if } D \Downarrow M \text{ and } M \neq_{\mathcal{E}} \mathsf{true} \qquad \text{(Red Cond 2}'')$$

Alternatively, processes with such expressions can easily be encoded into the core calculus by introducing additional expression evaluations. For instance, $\mathsf{in}(D, x : T); P$ can be encoded as

$$\mathsf{let}\ y = D\ \mathsf{in}\ \mathsf{in}(y, x : T); P\,.$$

Furthermore, ProVerif allows expressions to contain some constructs from processes: restriction, expression evaluation, and conditional, so that the syntax of expressions is extended as follows:

$D ::=$	expressions
...	
$\mathsf{new}\ a : T; D$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ D'\ \mathsf{else}\ D''$	expression evaluation
$\mathsf{if}\ D\ \mathsf{then}\ D'\ \mathsf{else}\ D''$	conditional

Processes that contain such expressions are handled by transforming them into processes without such expressions, by moving restrictions, expression evaluations, and conditionals from expressions to processes.

2.5.3 Pattern-matching

We enrich the input and expression evaluation constructs with pattern-matching as follows:

$P, Q ::=$	processes
...	
$\mathsf{in}(D, pat); P$	input
$\mathsf{let}\ pat = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation

where patterns pat are defined by the following grammar:

$pat ::=$	patterns
$x : T$	variable
$=D$	equality test
$f(pat_1, \ldots, pat_n)$	data constructor

The pattern $x : T$ matches any term and stores it in variable x. The type T can be omitted when it can be inferred. The variables x inside a pattern must be pairwise distinct.

The pattern $=D$ matches only terms that are equal to the result of the evaluation of D. When the evaluation of D fails, the pattern-matching fails.

The pattern $f(pat_1, \ldots, pat_n)$ matches terms of the form $f(M_1, \ldots, M_n)$, when pat_i matches M_i for all $i \leq n$. This pattern can be used only when f is *data constructor*. A data constructor is a constructor f of arity n that comes with associated destructors g_i for $i \in \{1, \ldots, n\}$ defined by $g_i(f(x_1, \ldots, x_n)) \to x_i$. Data constructors are typically used for representing data structures. Tuples are examples of data constructors. Pattern-matching with data constructors allows one to extract the contents of the data structures. To be able to pattern-match on data constructors without considering the equational theory, we require that the equational theory satisfies the following condition: for all data constructors f, $f(M_1, \ldots, M_n) =_{\mathcal{E}} M'$ if and only if there exist $M'_1, \ldots, M'_n$ such that $M' = f(M'_1, \ldots, M'_n)$ and $M_i =_{\mathcal{E}} M'_i$ for all $i \in \{1, \ldots, n\}$.

When the pattern-matching fails, the expression evaluation $\mathsf{let}\ pat = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ runs Q. For an input $\mathsf{in}(D, pat); P$, the communication is executed even when the pattern-matching fails, but the receiver process does nothing after the input. In other words, $\mathsf{in}(D, pat); P$ is an abbreviation for $\mathsf{in}(D, x : T); \mathsf{let}\ pat = x\ \mathsf{in}\ P\ \mathsf{else}\ \mathbf{0}$ where T is the type of the pattern pat.

Example 2.5. We consider again the protocol of §2.2. We may add the public key of B inside the first message, which becomes

$$\text{Message 1.} \quad A \to B : \mathsf{aenc}(\mathsf{sign}((pk_B, k), ssk_A), pk_B)$$

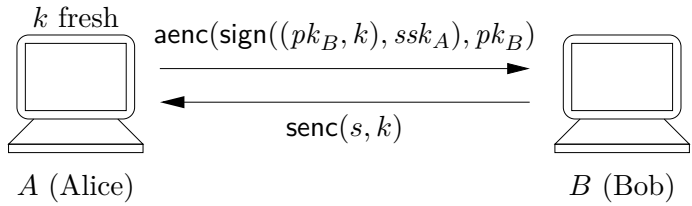

Figure 2.5: Modified example of protocol

The modified protocol is illustrated in Figure 2.5. The processes P_A and P_B are then modified accordingly:

$$\begin{aligned} P_A(ssk_A, pk_B) = {} & !\,\mathsf{in}(c, x_{pk_B} : \mathsf{pkey}); \mathsf{new}\ k : \mathsf{key}; \\ & \mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}((x_{pk_B}, k), ssk_A), x_{pk_B})); \\ & \mathsf{in}(c, x : \mathsf{bitstring}); \mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ \mathbf{0} \\ P_B(sk_B, spk_A) = {} & !\,\mathsf{in}(c, y : \mathsf{bitstring}); \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in} \\ & \mathsf{let}\ (=\mathsf{pk}(sk_B), x_k : \mathsf{key}) = \mathsf{check}(y', spk_A)\ \mathsf{in}\ \mathsf{out}(c, \mathsf{senc}(s, x_k)) \end{aligned}$$

The process P_A signs the pair (x_{pk_B}, k). When P_B verifies the signature, it uses pattern-matching to verify that the signed message is a pair whose first component is its own public key $pk_B = \mathsf{pk}(sk_B)$, and to store the second component in the variable x_k. Because of the overloading of tuples, the type of x_k, namely key, must be explicitly mentioned, so that ProVerif knows that the pair $(\_, \_)$ stands for $\mathsf{tuple}_{\mathsf{pkey},\mathsf{key}}(\_, \_)$.

Pattern-matching can be encoded into the core language using equality tests and the destructors associated to data constructors, as shown by the following example.

Example 2.6. The process P_B of Example 2.5 can be encoded into the calculus without pattern-matching as follows:

$$\begin{aligned} P_B(sk_B, spk_A) = {} & !\,\mathsf{in}(c, y : \mathsf{bitstring}); \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in} \\ & \mathsf{let}\ z = \mathsf{check}(y', spk_A)\ \mathsf{in}\ \mathsf{if}\ 1\mathsf{th}_{\mathsf{pkey},\mathsf{key}}(z) = \mathsf{pk}(sk_B)\ \mathsf{then} \\ & \mathsf{let}\ x_k = 2\mathsf{th}_{\mathsf{pkey},\mathsf{key}}(z)\ \mathsf{in}\ \mathsf{out}(c, \mathsf{senc}(s, x_k)) \end{aligned}$$

This process uses the extension of §2.5.2 to allow destructors in $\mathsf{if}\ 1\mathsf{th}_{\mathsf{pkey},\mathsf{key}}(z) = \mathsf{pk}(sk_B)\ \mathsf{then}$. It can be encoded in the core calculus as

follows:

$$P_B(sk_B, spk_A) = !\,\mathsf{in}(c, y : \mathsf{bitstring}); \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in}$$
$$\mathsf{let}\ z = \mathsf{check}(y', spk_A)\ \mathsf{in}\ \mathsf{let}\ z' = (1\mathsf{th}_{\mathsf{pkey,key}}(z) = \mathsf{pk}(sk_B))\ \mathsf{in}$$
$$\mathsf{if}\ z'\ \mathsf{then}\ \mathsf{let}\ x_k = 2\mathsf{th}_{\mathsf{pkey,key}}(z)\ \mathsf{in}\ \mathsf{out}(c, \mathsf{senc}(s, x_k))$$

Obviously, the formal semantics of calculus can also be extended with pattern-matching as well as the following extensions described in this section. We omit these formal details for simplicity.

2.5.4 Phases

Some situations can be modeled with several phases or stages (Blanchet *et al.*, 2008, §8). For instance, the protocol may run in a first phase, and a long-term key may be compromised in a later phase. One may hope that the secret messages exchanged in the first phase remain secret even after the compromise of the key. This property is named forward secrecy. Voting protocols may also include several phases, for instance a registration phase, a voting phase, and a tallying phase (Kremer and Ryan, 2005). ProVerif includes a phase construct to model such situations:

$P, Q ::=$	processes
...	
$\mathsf{phase}\ n; P$	phase

The phase construct acts as a global synchronization. The processes initially run in phase 0. Then at some point, phase 1 starts. All processes that did not reach a phase n construct with $n \geq 1$ are discarded, and processes that start with phase 1 run. Phases continue being incremented in the same way.

Example 2.7. The following process

$$P_0 = \mathsf{new}\ ssk_A : \mathsf{skey}; (P \mid \mathsf{phase}\ 1; \mathsf{out}(c, ssk_A))\,,$$

where P does not contain phases, models a protocol P that runs in phase 0, using the secret key ssk_A. This key is then compromised in phase 1, by sending it on the public channel c.

2.5.5 Tables

Tables are often useful in the modeling of security protocols. For instance, an SSH client stores a list of names and keys of the servers it has contacted so far; this list can be modeled as a table. ProVerif supports specific constructs for modeling tables:

$P, Q ::=$	processes
...	
insert $tbl(D_1, \ldots, D_n); P$	insertion in a table
get $tbl(pat_1, \ldots, pat_n)$ suchthat D in P else Q	lookup in a table

The process insert $tbl(D_1, \ldots, D_n); P$ inserts the record $(M_1, \ldots, M_n)$ in the table tbl, where $D_1, \ldots, D_n$ evaluate to $M_1, \ldots, M_n$ respectively, then runs P. The process get $tbl(pat_1, \ldots, pat_n)$ suchthat D in P else Q looks for a record $(M_1, \ldots, M_n)$ in the table tbl that matches the patterns $(pat_1, \ldots, pat_n)$ and such that the condition D evaluates to true. When such a record is found, it runs P with the variables of $pat_1, \ldots, pat_n$ bound to the corresponding values. Otherwise, it runs Q. The condition "suchthat D" can be omitted when D is true. The branch "else Q" can be omitted when Q is $\mathbf{0}$.

Example 2.8. We consider again the protocol of §2.2. Instead of adding the public key of B to the first message, as in Example 2.5, we may add the name of B. The first message would then become

$$\text{Message 1.} \quad A \to B : \mathsf{aenc}(\mathsf{sign}((B, k), ssk_A), pk_B)\,.$$

However, with such a protocol, the participants need to relate the keys (e.g. pk_B) to the identities (e.g. B). We establish this relation using certificates, in the same way as in the original Denning-Sacco key distribution protocol (Denning and Sacco, 1981): we use $\mathsf{sign}((\mathsf{e}, B, pk_B), ssk_S)$ as a certificate that pk_B is the public encryption key of B and $\mathsf{sign}((\mathsf{s}, A, spk_A), ssk_S)$ as a certificate that spk_A is the public signature verification key of A, where ssk_S is the signature key of a trusted certificate authority S (server). The protocol becomes:

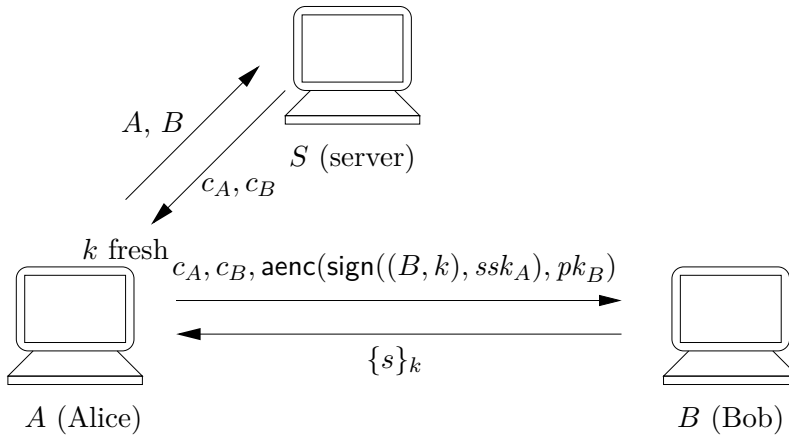

Figure 2.6: Protocol with certificates

Message 1. $A \to S : A, B$
Message 2. $S \to A : c_A, c_B$
Message 3. $A \to B : c_A, c_B, \mathsf{aenc}(\mathsf{sign}((B, k), ssk_A), pk_B)$
Message 4. $B \to A : \mathsf{senc}(s, k)$

where $c_A = \mathsf{sign}((\mathsf{s}, A, spk_A), ssk_S)$ and $c_B = \mathsf{sign}((\mathsf{e}, B, pk_B), ssk_S)$. This protocol is illustrated in Figure 2.6. A first asks the server for the certificate for its own signature key spk_A and for the encryption key pk_B of B. The second certificate allows A to obtain the key pk_B. Then it sends the message $\mathsf{aenc}(\mathsf{sign}((B, k), ssk_A), pk_B)$ with the certificates to B. B uses c_A to obtain A's signature verification key spk_A. It decrypts and verifies the signature as before; it additionally verifies that B is the first component of the signed message.

A ProVerif model for this protocol is shown in Figure 2.7. This model considers two participants *Alice* and *Bob*, which can play both roles A and B of the protocol. The process P_0 first generates the encryption and signature keys for *Alice* and *Bob*, and inserts them in a table *keys*, which contains triples (identity, encryption key, signature verification key) for each participant. It also generates the signature keys for the server S, and publishes all public keys by sending them on the public channel c. Finally, it runs the processes that model *Alice*, *Bob*, and

$$
\begin{aligned}
&P_0 = \mathsf{new}\ ssk_A : \mathsf{skey}; \mathsf{let}\ spk_A = \mathsf{pk}(ssk_A)\ \mathsf{in}\\
&\quad \mathsf{new}\ sk_A : \mathsf{skey}; \mathsf{let}\ pk_A = \mathsf{pk}(sk_A)\ \mathsf{in}\\
&\quad \mathsf{insert}\ keys(Alice, pk_A, spk_A);\\
&\quad \mathsf{new}\ ssk_B : \mathsf{skey}; \mathsf{let}\ spk_B = \mathsf{pk}(ssk_B)\ \mathsf{in}\\
&\quad \mathsf{new}\ sk_B : \mathsf{skey}; \mathsf{let}\ pk_B = \mathsf{pk}(sk_B)\ \mathsf{in}\\
&\quad \mathsf{insert}\ keys(Bob, pk_B, spk_B);\\
&\quad \mathsf{new}\ ssk_S : \mathsf{skey}; \mathsf{let}\ spk_S = \mathsf{pk}(ssk_S)\ \mathsf{in}\\
&\quad \mathsf{out}(c, (pk_A, spk_A, pk_B, spk_B, spk_S));\\
&\quad (P_A(spk_S, Alice, ssk_A, spk_A) \mid P_A(spk_S, Bob, ssk_B, spk_B) \mid\\
&\quad\ P_B(spk_S, Bob, sk_B, pk_B) \mid P_B(spk_S, Alice, sk_A, pk_A) \mid\\
&\quad\ P_S(ssk_S) \mid P_K)\\
&P_K = !\ \mathsf{in}(c, (h : \mathsf{host}, pk : \mathsf{pkey}, spk : \mathsf{pkey}));\\
&\quad \mathsf{if}\ h \neq Alice\ \&\&\ h \neq Bob\ \mathsf{then}\ \mathsf{insert}\ keys(h, pk, spk)\\
&P_S(ssk_S) = !\ \mathsf{in}(c, (h_1 : \mathsf{host}, h_2 : \mathsf{host}));\\
&\quad \mathsf{get}\ keys(=h_1, xx, spk_1)\ \mathsf{in}\ \mathsf{get}\ keys(=h_2, pk_2, yy)\ \mathsf{in}\\
&\quad \mathsf{out}(c, (\mathsf{sign}((\mathsf{s}, h_1, spk_1), ssk_S), \mathsf{sign}((\mathsf{e}, h_2, pk_2), ssk_S)))\\
&P_A(spk_S, A, ssk_A, spk_A) = !\ \mathsf{in}(c, (c_A : \mathsf{bitstring}, c_B : \mathsf{bitstring}));\\
&\quad \mathsf{if}\ (\mathsf{s}, A, spk_A) = \mathsf{check}(c_A, spk_S)\ \mathsf{then}\\
&\quad \mathsf{let}\ (=\mathsf{e}, B, pk_B) = \mathsf{check}(c_B, spk_S)\ \mathsf{in}\ \mathsf{new}\ k : \mathsf{key};\\
&\quad \mathsf{out}(c, (c_A, c_B, \mathsf{aenc}(\mathsf{sign}((B, k), ssk_A), pk_B)));\\
&\quad \mathsf{in}(c, x : \mathsf{bitstring}); \mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ \mathbf{0}\\
&P_B(spk_S, B, sk_B, pk_B) =\\
&\quad !\ \mathsf{in}(c, (c_A : \mathsf{bitstring}, c_B : \mathsf{bitstring}, y : \mathsf{bitstring}));\\
&\quad \mathsf{if}\ (\mathsf{e}, B, pk_B) = \mathsf{check}(c_B, spk_S)\ \mathsf{then}\\
&\quad \mathsf{let}\ (=\mathsf{s}, A, spk_A) = \mathsf{check}(c_A, spk_S)\ \mathsf{in}\ \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in}\\
&\quad \mathsf{let}\ (=B, x_k : \mathsf{key}) = \mathsf{check}(y', spk_A)\ \mathsf{in}\\
&\quad \mathsf{if}\ A = Alice\ ||\ A = Bob\ \mathsf{then}\ \mathsf{out}(c, \mathsf{senc}(s, x_k))
\end{aligned}
$$

Figure 2.7: ProVerif example with tables

S, as well as the process P_K. This process allows the adversary to register its own keys for participants other than *Alice* and *Bob* by inserting these keys in the table *keys*. Therefore, the adversary can implement any number of dishonest participants to the protocol. The process P_S for the server receives two participant names h_1 and h_2, obtains their keys from the table *keys*, and outputs a certificate for the signature verification key of h_1 and one for the encryption key of h_2. The process $P_A(spk_S, A, ssk_A, spk_A)$ models a participant A with secret signature key ssk_A and public signature verification key spk_A, running role A of the protocol. We suppose that the adversary sends the first message of the protocol himself. (This is possible since the participant names are public.) So this message does not appear explicitly in P_A. The process P_A receives message 2 containing the certificates, checks them, outputs message 3, and receives message 4. The process $P_B(spk_S, B, sk_B, pk_B)$ models a participant B with secret decryption key sk_B and public encryption key pk_B, running role B of the protocol. It receives message 3, checks the certificates, and when its interlocutor A is honest (that is, this interlocutor is *Alice* or *Bob*), it sends a secret s encrypted under the shared key x_k as message 4. This message is only sent when the interlocutor is honest, because it is perfectly normal that the adversary can decrypt the last message when the interlocutor of B is dishonest, so the secrecy of s would not be preserved in this case.

Tables can be encoded in the core calculus using private channels. ProVerif provides a specific construct as it is frequently used and it is probably easier to understand for users.

3

Verifying Security Properties

This chapter presents the method used by ProVerif for verifying protocols. This method based on Horn clauses. The idea of using Horn clauses for verifying protocols was introduced by Weidenbach, 1999. We extended his work by defining a systematic translation from a formal model of protocols to clauses (while he built the clauses manually) and by proving properties other than secrecy. We first formalize the notion of an adversary, then deal with the various security properties verified by ProVerif, starting with the simplest one, secrecy, then considering more complex ones, correspondences and equivalences. The main reference for the proof of secrecy and correspondences in ProVerif is (Blanchet, 2009) and for equivalences (Blanchet *et al.*, 2008). For simplicity, this chapter only deals with the core calculus of §2.1. The results can be adapted to the extended calculus, and ProVerif supports that calculus.

3.1 Adversary

We assume that the protocol is executed in the presence of an adversary that intercept all messages, compute, and send all messages it has, following the Dolev-Yao model (Dolev and Yao, 1983). In our calculus,

an adversary can be represented by any process that has the set of public free names $\mathcal{N}_{\text{pub}}$ in its initial knowledge. (Although the initial knowledge of the adversary contains only names in $\mathcal{N}_{\text{pub}}$, one can give any terms to the adversary by sending them on a channel in $\mathcal{N}_{\text{pub}}$.)

Definition 3.1. Let $\mathcal{N}_{\text{pub}}$ be a finite set of names. The closed process Q is an $\mathcal{N}_{\text{pub}}$-*adversary* if and only if $fn(Q) \subseteq \mathcal{N}_{\text{pub}}$ and all function symbols in Q are public. (The process Q is not necessarily well-typed.)

3.2 Secrecy

This section deals with the verification of secrecy, the most basic security property. We first define secrecy formally, then explain how to verify it by translating the protocol into Horn clauses and using resolution on these clauses.

3.2.1 Definition

Intuitively, a process P preserves the secrecy of M when M cannot be output on a public channel, in a run of P with any adversary. Formally, we define that a trace outputs M as follows:

Definition 3.2. We say that a trace $Tr = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}_0 \rightarrow^* E', \mathcal{P}'$ outputs M publicly if and only if Tr contains a reduction $E, \mathcal{P} \cup \{\, \mathsf{out}(N, M); Q, \mathsf{in}(N, x); P \,\} \rightarrow E, \mathcal{P} \cup \{\, Q, P\{^M/_x\} \,\}$ for some E, $\mathcal{P}$, N, x, P, Q, with $N \in \mathcal{N}_{\text{pub}}$.

We can finally define secrecy:

Definition 3.3. The closed process P_0 *preserves the secrecy of* the closed term M from $\mathcal{N}_{\text{pub}}$ if and only if for some $\mathcal{N}_{\text{priv}}$ disjoint from $\mathcal{N}_{\text{pub}}$ such that $fn(M) \cup fn(P_0) \subseteq \mathcal{N}_{\text{pub}} \cup \mathcal{N}_{\text{priv}}$, for any $\mathcal{N}_{\text{pub}}$-adversary Q, for any trace $Tr = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \{P_0, Q\} \rightarrow^* E', \mathcal{P}'$, the trace Tr does not output M publicly.

The choice of $\mathcal{N}_{\text{priv}}$ does not matter, provided the conditions of Definition 3.3 are satisfied.

3.2.2 From the pi calculus to Horn clauses

Given a closed process P_0 in the language of Chapter 2 and a set of names $\mathcal{N}_{\text{pub}}$ representing the initial knowledge of the adversary, ProVerif builds a set of Horn clauses, representing the protocol P_0 in parallel with any $\mathcal{N}_{\text{pub}}$-adversary. This translation was originally given in (Abadi and Blanchet, 2005a); we extend it to the richer destructors of (Cheval and Blanchet, 2013). We suppose that the bound names of P_0 are pairwise distinct and distinct from the free names and the names in $\mathcal{N}_{\text{pub}}$.

In these clauses, messages are represented by *patterns* p, defined by the following grammar:

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

Patterns are terms; we use the word patterns to distinguish them from terms of the process calculus. Patterns differ from those terms by the representation of names. We assign a pattern $a[p_1, \ldots, p_n]$ to each name a of P_0. We treat a as a function symbol, and write $a[p_1, \ldots, p_n]$ rather than $a(p_1, \ldots, p_n)$ only to distinguish functions that come from names from other functions. If a is a free name, then its pattern is $a[\,]$. To define the patterns of bound names, we first assign a distinct, fresh session identifier variable i to each replication of P_0. (We will use a distinct value for i for each copy of the replicated process.) If a is bound by a restriction $\mathsf{new}\ a$ in P_0, then its pattern takes as arguments the terms received as inputs and the session identifiers of replications above the restriction. For example, in the process $!\mathsf{in}(c, x); \mathsf{new}\ a; P$, each name created by $\mathsf{new}\ a$ is represented by $a[i, x]$ where i is the session identifier for the replication and x is the message received as input in $\mathsf{in}(c, x)$. Session identifiers enable us to distinguish names created in different copies of processes. Hence, each name created in the process calculus is represented by a different pattern in the Horn clauses.

The evaluation of expressions may fail. We reflect that in the Horn clauses by introducing *may-fail patterns*, which can be the constant fail or a may-fail variable in addition to an ordinary pattern:

$mp ::=$	may-fail pattern
p	pattern
u	may-fail variable
fail	failure

As in messages and may-fail messages, a may-fail variable u can be instantiated by a pattern or fail, whereas a variable x cannot be instantiated by fail.

The clauses use *facts* defined by the following grammar:

$F ::=$	facts
$\mathsf{attacker}(mp)$	adversary knowledge
$\mathsf{message}(p, p')$	message on a channel
$\forall \widetilde{v}, (mp_1, \ldots, mp_n) \neq (mp'_1, \ldots, mp'_n)$	disequality

The fact $\mathsf{attacker}(mp)$ means that the adversary may have mp. The fact $\mathsf{message}(p, p')$ means that the message p' may appear on channel p. The fact $\forall \widetilde{v}, (mp_1, \ldots, mp_n) \neq (mp'_1, \ldots, mp'_n)$ means that $(mp_1, \ldots, mp_n) \neq (mp'_1, \ldots, mp'_n)$ for all values of variables in $\widetilde{v}$, where $\widetilde{v}$ may contain variables as well as may-fail variables. We write v for a variable or a may-fail variable. We omit $\forall \widetilde{v}$ when $\widetilde{v}$ contains no variable. We omit the parentheses around $mp_1, \ldots, mp_n$ and $mp'_1, \ldots, mp'_n$ when $n = 1$. We write $\widetilde{mp}$ as an abbreviation for $(mp_1, \ldots, mp_n)$.

The clauses are of the form $F_1 \wedge \ldots \wedge F_n \Rightarrow F$, where $F_1, \ldots, F_n, F$ are facts. They comprise clauses for the adversary and clauses for the protocol, defined below. These clauses form the set $\mathcal{R}_{P_0, \mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}}$.

Clauses for the adversary

The abilities of the adversary are represented by the following clauses:

$$\text{For each } a \in \mathcal{N}_{\mathrm{pub}},\ \mathsf{attacker}(a[\,]) \tag{Init}$$

$$\mathsf{attacker}(b_0[x]) \text{ where } b_0 \text{ occurs neither in } P_0 \text{ nor in } \mathcal{N}_{\mathrm{pub}} \cup \mathcal{N}_{\mathrm{priv}} \tag{Rn}$$

$$\mathsf{attacker}(\mathsf{fail}) \tag{Rfail}$$

For each public function h, such that $\mathrm{def}(h)$ consists of the rewrite rules $h(U_{i,1}, \ldots, U_{i,n}) \to U_i$ for $i \in \{1, \ldots, k\}$, the variables in these rewrite rules are renamed so that distinct rewrite rules use different variables, and $\widetilde{v}_i = fv(U_i) \cup \bigcup_{l=1}^{n} fv(U_{i,l})$,

for each $i \in \{1, \ldots, k\}$,

$$\mathsf{attacker}(U_{i,1}) \wedge \ldots \wedge \mathsf{attacker}(U_{i,n}) \wedge \bigwedge_{j<i} \forall \widetilde{v}_j, (U_{i,1}, \ldots, U_{i,n}) \neq (U_{j,1}, \ldots, U_{j,n}) \Rightarrow \mathsf{attacker}(U_i) \tag{Rh}$$

$$\mathsf{message}(x, y) \wedge \mathsf{attacker}(x) \Rightarrow \mathsf{attacker}(y) \tag{Rl}$$

$$\mathsf{attacker}(x) \wedge \mathsf{attacker}(y) \Rightarrow \mathsf{message}(x, y) \tag{Rs}$$

Clause (Init) represents the initial knowledge of the adversary. Clause (Rn) means that the adversary can generate an unbounded number of new names; these names are represented by patterns of the form $b_0[x]$. Clause (Rfail) means that the adversary has the special constant fail. Clauses (Rh) mean that the adversary can apply all public functions to all terms it has: if it has an instance of $U_{i,1}, \ldots, U_{i,n}$, it can compute the corresponding instance of U_i by applying the rewrite rule $h(U_{i,1}, \ldots, U_{i,n}) \to U_i$, provided no previous rewrite rule applies, which is checked by $\bigwedge_{j<i} \forall \widetilde{v}_j, (U_{i,1}, \ldots, U_{i,n}) \neq (U_{j,1}, \ldots, U_{j,n})$. The rewrite rules in $\mathrm{def}(h)$ do not contain names, and terms without names are also patterns, so these clauses have the required format. When h is a constructor f of arity n, these clauses simplify into

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

and when h is a destructor g with a single rewrite rule $g(U_1, \ldots, U_n) \to U$, they simplify into

$$\mathsf{attacker}(U_1) \wedge \ldots \wedge \mathsf{attacker}(U_n) \Rightarrow \mathsf{attacker}(U) \tag{Rg}$$

(The clauses that conclude fail can be removed since they are subsumed by (Rfail).) Clause (Rl) means that the adversary can listen on all channels it has, and (Rs) that it can send all messages it has on all channels it has. When a message is sent on a channel, both the message and the channel cannot be fail, so x and y are ordinary variables.

Example 3.1. We suppose that public-key encryption is defined by (2.1). The constructors senc, pk, aenc, and sign yield the following clauses:

$$\mathsf{attacker}(m) \wedge \mathsf{attacker}(k) \Rightarrow \mathsf{attacker}(\mathsf{senc}(m,k)) \quad (\mathsf{senc})$$
$$\mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(\mathsf{pk}(sk)) \quad (\mathsf{pk})$$
$$\mathsf{attacker}(m) \wedge \mathsf{attacker}(pk) \Rightarrow \mathsf{attacker}(\mathsf{aenc}(m,pk)) \quad (\mathsf{aenc})$$
$$\mathsf{attacker}(m) \wedge \mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(\mathsf{sign}(m,sk)) \quad (\mathsf{sign})$$

The corresponding destructors sdec, adec, check, and getmess yield the following clauses:

$$\mathsf{attacker}(\mathsf{senc}(m,k)) \wedge \mathsf{attacker}(k) \Rightarrow \mathsf{attacker}(m) \quad (\mathsf{sdec})$$
$$\mathsf{attacker}(\mathsf{aenc}(m,\mathsf{pk}(sk))) \wedge \mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(m) \quad (\mathsf{adec})$$
$$\mathsf{attacker}(\mathsf{sign}(m,sk)) \wedge \mathsf{attacker}(\mathsf{pk}(sk)) \Rightarrow \mathsf{attacker}(m) \quad (\mathsf{check})$$
$$\mathsf{attacker}(\mathsf{sign}(m,sk)) \Rightarrow \mathsf{attacker}(m) \quad (\mathsf{getmess})$$

For instance, the clause for sdec models that, when the adversary has the ciphertext $\mathsf{senc}(m,k)$ and the key k, then it can obtain the cleartext m by decryption. The clause for check is less general than the one for getmess since, in the former clause, the attacker additionally needs to know the key $\mathsf{pk}(sk)$; more formally, the clause for check is subsumed by the one for getmess, so it is discarded during the resolution algorithm (see "Elimination of subsumed clauses" in §3.2.3).

Clauses for the protocol

To translate the protocol into clauses, we first need to define evaluation on patterns expressions, defined by the following grammar:

$Dp ::=$	pattern expression
$\quad mp$	may-fail pattern
$\quad h(Dp_1, \ldots, Dp_n)$	function application

Evaluation of open pattern expressions is defined as a relation $Dp \Downarrow' (mp, \sigma, \phi)$, where the substitution σ collects instantiations of Dp obtained by unification and the formula ϕ collects the side conditions that express that certain rewrite rules of destructors do not apply. More formally,

the relation $Dp \Downarrow' (mp, \sigma, \phi)$ specifies how instances of Dp evaluate: if $Dp \Downarrow' (mp, \sigma, \phi)$, then for any substitution σ' such that $\sigma'\phi$ holds, we have $\sigma'\sigma Dp \Downarrow \sigma' mp$. There may be several (mp, σ, ϕ) such that $Dp \Downarrow' (mp, \sigma, \phi)$ in case several instances of Dp reduce in a different way. This relation is defined as follows:

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

$h(Dp_1, \ldots, Dp_n) \Downarrow' (\sigma_u U_i, \sigma_u\sigma', \sigma_u\phi' \wedge \sigma_u\phi)$
if $(Dp_1, \ldots, Dp_n) \Downarrow' ((mp_1, \ldots, mp_n), \sigma', \phi')$,
σ_u is a most general unifier of $(mp_1, U_{i,1})$, $\ldots$, $(mp_n, U_{i,n})$, and
$\phi = \bigwedge_{j<i} \forall \widetilde{v}_j, (mp_1, \ldots, mp_n) \neq (U_{j,1}, \ldots, U_{j,n})$
for some $i \in \{1, \ldots, k\}$,
where $\mathrm{def}(h)$ consists of the rewrite rules $h(U_{i,1}, \ldots, U_{i,n}) \to U_i$ for $i \in \{1, \ldots, k\}$, the variables in these rewrite rules are renamed so that distinct rewrite rules use different variables, and $\widetilde{v}_i = fv(U_i) \cup \bigcup_{l=1}^{n} fv(U_{i,l})$ for all $i \in \{1, \ldots, k\}$.

$(Dp_1, \ldots, Dp_n) \Downarrow' ((\sigma_n mp_1, \ldots, \sigma_n mp_{n-1}, mp_n), \sigma_n\sigma, \sigma_n\phi \wedge \phi_n)$
if $(Dp_1, \ldots, Dp_{n-1}) \Downarrow' ((mp_1, \ldots, mp_{n-1}), \sigma, \phi)$
and $\sigma Dp_n \Downarrow' (mp_n, \sigma_n, \phi_n)$

The substitution $\emptyset$ is the identity. The formula $\top$ denotes the empty conjunction, that is, a formula that always holds. The most general unifier of may-fail patterns is computed similarly to usual most general unifiers, even though specific cases hold due to may-fail variables and ordinary variables: there is no unifier of p and fail, for any pattern p (including variables x, because these variables can be instantiated only by messages); the most general unifier of u and mp is $\{^{mp}/_{u}\}$; the most general unifier of fail and fail is the identity; finally, the most general unifier of p and p' is computed as usual.

The first rule of the definition $\Downarrow'$ treats cases without destructors. The second one treats function application: it applies each rewrite rule of h. To apply the i-th rewrite rule, it evaluates the arguments of h to $mp_1, \ldots, mp_n$, checks that the rewrite rule actually applies by unifying $mp_1, \ldots, mp_n$ with the arguments of h in the rewrite rule, and checks that all rewrite rules before the i-th do not apply thanks to the formula ϕ. Finally, the third rule treats the evaluation of a tuple.

The translation $[\![P]\!]\rho sH$ of a process P is a set of clauses, where ρ is an environment that associates a pattern with each name and variable, s is a sequence of patterns, and H is a sequence of facts. We extend ρ as a substitution by $\rho(h(M_1, \ldots, M_n)) = h(\rho(M_1), \ldots, \rho(M_n))$ and $\rho(\mathsf{fail}) = \mathsf{fail}$. The empty sequence is written $\emptyset$; the concatenation of a pattern p to the sequence s is written s, p; the concatenation of a fact F to the sequence H is written $H \wedge F$. Intuitively, H represents the hypothesis of the clauses, ρ represents the names and variables that are already associated with a pattern, and s represents the current values of session identifiers and inputs. The translation $[\![P]\!]\rho sH$ is defined as follows:

$[\![\mathbf{0}]\!]\rho sH = \emptyset$

$[\![P \mid Q]\!]\rho sH = [\![P]\!]\rho sH \cup [\![Q]\!]\rho sH$

$[\![!P]\!]\rho sH = [\![P]\!]\rho(s, i)H$ where i is a fresh variable

$[\![\mathsf{new}\ a; P]\!]\rho sH = [\![P]\!](\rho[a \mapsto a[s]])sH$

$[\![\mathsf{in}(M, x); P]\!]\rho sH = [\![P]\!](\rho[x \mapsto x'])(s, x')(H \wedge \mathsf{message}(\rho(M), x'))$
where x' is a fresh variable

$[\![\mathsf{out}(M, N); P]\!]\rho sH = [\![P]\!]\rho sH \cup \{H \Rightarrow \mathsf{message}(\rho(M), \rho(N))\}$

$[\![\mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]\rho sH =$
$\bigcup\{[\![P]\!]((\sigma\rho)[x \mapsto p])(\sigma s)(\sigma H \wedge \phi) \mid \rho(D) \Downarrow' (p, \sigma, \phi)\}$
$\cup \bigcup\{[\![Q]\!](\sigma\rho)(\sigma s)(\sigma H \wedge \phi) \mid \rho(D) \Downarrow' (\mathsf{fail}, \sigma, \phi)\}$

$[\![\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!]\rho sH =$

$$\begin{cases} [\![P]\!](\sigma\rho)(\sigma s)(\sigma H) \cup [\![Q]\!]\rho s(H \wedge \rho(M) \neq \mathsf{true}) \\ \quad \text{where } \sigma \text{ is the most general unifier of } \rho(M) \text{ and } \mathsf{true}, \\ \quad \text{when } \rho(M) \text{ and } \mathsf{true} \text{ unify} \\ [\![Q]\!]\rho sH \quad \text{when } \rho(M) \text{ and } \mathsf{true} \text{ do not unify} \end{cases}$$

The translation of a process is a set of Horn clauses that express that it may send certain messages.

- The nil process does nothing, so its translation is empty.
- The clauses for the parallel composition of processes P and Q are

the union of clauses for P and Q.

- For the replication, we create a fresh session identifier i and add it to the sequence s. The replication is otherwise ignored, because all Horn clauses are applicable arbitrarily many times.

- For the restriction, we replace the restricted name a in question with the pattern $a[s]$, where the sequence s contains the previous inputs and session identifiers.

- The sequence H is extended in the translation of an input, with the input in question. The sequence s is also extended with the received message.

- The translation of an output adds a clause, meaning that the output is triggered when all conditions in H are true.

- The translation of an expression evaluation is the union of the clauses for the cases where the evaluation succeeds and where the evaluation fails. In case of success, the variable x is bound to the result p in ρ. In both cases, ρ, s, and H are instantiated by σ and the formula ϕ is added to the hypothesis H, where $\rho(D) \Downarrow' (mp, \sigma, \phi)$.

- The translation of a conditional is also the union of the clauses for the cases where the condition is true and where it is not true. (In the second case of the definition, when $\rho(M)$ and true do not unify, the condition is never true, so the else branch is always executed.)

The clauses corresponding to the process P_0 are computed by $[\![P_0]\!]\rho_0\emptyset\emptyset$ where $\rho_0 = \{a \mapsto a[\,] \mid a \in fn(P_0)\}$. These clauses are of the form $\mathsf{message}(p_1, p_1') \wedge \ldots \wedge \mathsf{message}(p_n, p_n') \wedge \phi \Rightarrow \mathsf{message}(p, p')$ when the process P_0 sends message p' on channel p after receiving messages $p_1', \ldots, p_n'$ on channels $p_1, \ldots, p_n$ respectively, provided the disequalities in ϕ hold. If $c \in \mathcal{N}_{\mathrm{pub}}$, ProVerif replaces all occurrences of $\mathsf{message}(c[\,], p)$ with $\mathsf{attacker}(p)$ in the clauses. Indeed, these facts are equivalent by the clauses (Rl) and (Rs).

Results and example

We define the clauses corresponding to the process P_0 as:

$$\mathcal{R}_{P_0,\mathcal{N}_{\text{pub}},\mathcal{N}_{\text{priv}}} = [\![P_0]\!]\rho_0\emptyset\emptyset \cup \{\mathsf{attacker}(a[\,]) \mid a \in \mathcal{N}_{\text{pub}}\} \cup \{(\text{Rn}), (\text{Rfail}), (\text{Rh}), (\text{Rl}), (\text{Rs})\}$$

The following theorem allows one to prove secrecy using Horn clauses:

Theorem 3.1 (Soundness of the clauses)**.** Let P_0 be a closed process and $\mathcal{N}_{\text{pub}}$ be a set of names. Let M be a closed term and p be the pattern obtained from the term M by replacing all names a with $a[\,]$. Let $\mathcal{N}_{\text{priv}}$ be a set of names disjoint from $\mathcal{N}_{\text{pub}}$ and such that $fn(P_0) \cup fn(M) \subseteq \mathcal{N}_{\text{pub}} \cup \mathcal{N}_{\text{priv}}$. If $\mathsf{attacker}(p)$ is not derivable from $\mathcal{R}_{P_0,\mathcal{N}_{\text{pub}},\mathcal{N}_{\text{priv}}}$, then P_0 preserves the secrecy of M from $\mathcal{N}_{\text{pub}}$.

The proof of this result relies on a type system to express the soundness of the clauses on P_0, and on the subject reduction of this type system to show that soundness of the clauses is preserved during all executions of the process. This technique was introduced in (Abadi and Blanchet, 2005a) where a similar result is proved. (Abadi and Blanchet, 2005a) also shows an equivalence between an instance of a generic type system for proving secrecy properties of protocols and the Horn clause verification method. This instance is the most precise instance of this generic type system: if a secrecy property can be proved by any instance of this type system, then it can be proved by the Horn clause approach. To use Theorem 3.1, one needs to determine whether a fact is derivable from the clauses. This is done using a resolution algorithm described in §3.2.3.

Example 3.2. Let $\mathcal{N}_{\text{pub}} = \{c\}$ be the initial knowledge of the adversary and $\mathcal{N}_{\text{priv}} = \{s\}$ be the private free names. For the process P_0 of Example 2.2, the clauses $[\![P_0]\!]\rho_0\emptyset\emptyset$ are, after replacing $\mathsf{message}(c[\,], p)$ with $\mathsf{attacker}(p)$:

$$\mathsf{attacker}(\mathsf{pk}(ssk_A[\,])) \tag{3.1}$$

$$\mathsf{attacker}(\mathsf{pk}(sk_B[\,])) \tag{3.2}$$

$$\mathsf{attacker}(x_{pk_B}) \Rightarrow \mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(k[i, x_{pk_B}], ssk_A[\,]), x_{pk_B})) \tag{3.3}$$

$$\mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(x_m, ssk_A[\,]), \mathsf{pk}(sk_B[\,]))) \Rightarrow \\ \mathsf{attacker}(\mathsf{senc}(s[\,], x_m)) \tag{3.4}$$

Clauses (3.1) and (3.2) correspond to the two outputs in P_0 itself, $\mathsf{out}(c, pk_A); \mathsf{out}(c, pk_B)$. They express that the adversary has the public keys. Clause (3.3) corresponds to the output in P_A: if the adversary has x_{pk_B}, it can send it to the first input of P_A, and P_A then replies with the message $\mathsf{aenc}(\mathsf{sign}(k[i, x_{pk_B}], ssk_A[\,]), x_{pk_B})$, which the adversary intercepts. The second input of P_A and the subsequent expression evaluation do not generate any clause, since no message is sent. Finally, Clause (3.4) corresponds to the output in P_B: if the adversary obtains a message of the form $\mathsf{aenc}(\mathsf{sign}(x_m, ssk_A[\,]), \mathsf{pk}(sk_B[\,]))$, it can send this message to P_B. The decryption and signature verification succeed, so P_B replies by sending s encrypted under x_m, which the adversary intercepts.

The fact $\mathsf{attacker}(s[\,])$ is derivable from the clauses $\mathcal{R}_{P_0,\mathcal{N}_{\text{pub}},\mathcal{N}_{\text{priv}}}$. Hence the secrecy of s cannot be proved by Theorem 3.1 for this protocol. The derivation obtained by ProVerif is shown in Figure 3.1. This derivation corresponds to the following well-known attack (Abadi and Needham, 1996) against this protocol:

Message 1.	$A \to C :$	$\mathsf{aenc}(\mathsf{sign}(k, ssk_A), pk_C)$
Message 1'.	$C(A) \to B :$	$\mathsf{aenc}(\mathsf{sign}(k, ssk_A), pk_B)$
Message 2.	$B \to C(A) :$	$\mathsf{senc}(s, k)$

This attack is illustrated in Figure 3.2. In this attack, A runs the protocol with a dishonest principal C. This principal gets the first message of the protocol $\mathsf{aenc}(\mathsf{sign}(k, ssk_A), pk_C)$, decrypts it and re-encrypts it under the public key of B. The obtained message $\mathsf{aenc}(\mathsf{sign}(k, ssk_A), pk_B)$ corresponds exactly to the first message of a session between A and B. Then, C sends this message to B impersonating A. B replies with the secret s, intended for A, encrypted under k. C, having obtained the key k by the first message, can decrypt this message and obtain the secret s.

The key sk_C corresponds to $b_0[x]$ in the derivation of Figure 3.1, and is generated by the adversary using Clause (Rn). The key pk_C corresponds to $\mathsf{pk}(b_0[x])$ and is computed by the application of Clause (Rf)

$$\dfrac{\dfrac{\dfrac{}{F_1 = \mathsf{attacker}(b_0[x])}\ \text{by (Rn)}}{\mathsf{attacker}(\mathsf{pk}(b_0[x]))}\ \text{by (Rf) for pk}}{F_2 = \mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(k[i, \mathsf{pk}(b_0[x])], ssk_A[\,]), \mathsf{pk}(b_0[x])))}\ \text{by (3.3)}$$

$$\dfrac{F_2 \qquad F_1}{F_3 = \mathsf{attacker}(\mathsf{sign}(k[i, \mathsf{pk}(b_0[x])], ssk_A[\,])}\ \text{by (Rg) for adec}$$

$$\dfrac{\dfrac{\dfrac{F_3 \qquad \dfrac{}{\mathsf{attacker}(\mathsf{pk}(sk_B[\,]))}\ \text{by (3.2)}}{\mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(k[i, \mathsf{pk}(b_0[x])], ssk_A[\,]), \mathsf{pk}(sk_B[\,])))}\ \text{by (Rf) for aenc}}{F_4 = \mathsf{attacker}(\mathsf{senc}(s[\,], k[i, \mathsf{pk}(b_0[x])]))}\ \text{by (3.4)}$$

$$\dfrac{F_4 \qquad \dfrac{F_3}{\mathsf{attacker}(k[i, \mathsf{pk}(b_0[x])])}\ \text{by (Rg) for getmess}}{\mathsf{attacker}(s[\,])}\ \text{by (Rg) for sdec}$$

Figure 3.1: Derivation of $\mathsf{attacker}(s[\,])$

for pk. The output of Message 1 corresponds to the application of Clause (3.3); then the adversary computes Message 1' by applying destructor adec to decrypt it and constructor aenc to reencrypt it under $pk_B = \mathsf{pk}(sk_B[])$, corresponding to Clauses (Rg) for adec and (Rf) for aenc of the derivation. The input of Message 1' and the output of Message 2 corresponds to Clause (3.4). Finally, the adversary computes k (represented in the derivation by $k[i, \mathsf{pk}(b_0[x])]$) using Clause (Rg) for getmess and decrypts Message 2 by Clause (Rg) for sdec.

We can model the fixed version of the protocol, mentioned in Example 2.5, in a similar way, and compute the corresponding clauses. ProVerif then shows that $\mathsf{attacker}(s[\,])$ is not derivable from these clauses. By Theorem 3.1, we obtain that the fixed protocol preserves the secrecy of s from $\{c\}$.

In the example above, we explained informally that the obtained derivation corresponds to an attack. Allamigeon and Blanchet, 2005 extended ProVerif so that it automatically reconstructs an attack from

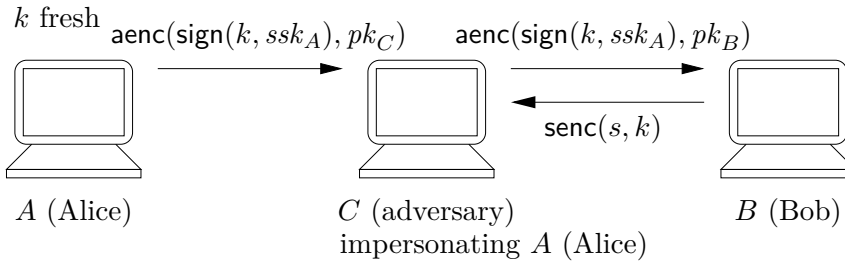

Figure 3.2: Attack

a derivation. The reconstructed attack is a trace of the process P_0 that outputs s publicly. The strategy for reconstructing this trace consists in executing the semantics of process P_0, guided by the derivation: a reduction of the process is executed only if a clause in the derivation corresponds to this reduction. We showed the soundness and termination of this algorithm. We also gave a formal definition of this correspondence between clauses and reductions, by giving an explicit construction of a derivation from a trace of P_0. We have then shown a partial completeness result for attack reconstruction: if all outputs in P_0 are of the form $\mathsf{out}(M, N); P$ where M is a name in $\mathcal{N}_{\text{pub}}$ not bound in P_0 or $P = 0$, and the derivation corresponds to a trace, then our algorithm succeeds in reconstructing a trace corresponding to the derivation. Moreover, with the same assumptions, our algorithm reconstructs a trace without backtracking. It is therefore very efficient in this case, and in practice it is generally very fast. We successfully tested this attack reconstruction algorithm on many protocols of the literature. To mention an extreme example, we could reconstruct an attack with 200 parallel sessions against the protocol $f^{200}g^{200}$ (Millen, 1999). (The protocol $f^n g^n$ has an attack with n parallel sessions.)

In the example above, the obtained derivation corresponds to an attack. This is unfortunately not always the case, since Horn clauses introduce approximations. These approximations are useful in order to handle an infinite state space, but because of these approximations, ProVerif may sometimes find a derivation while secrecy is preserved. In this case, attack reconstruction fails, obviously. The case in which

ProVerif finds a derivation but attack reconstruction fails corresponds to an "I do not know" answer. In all other cases, ProVerif gives a correct answer: either it finds no derivation and the desired property is proved, or attack reconstruction succeeds, and an attack against the property in question is found.

The main approximation done by ProVerif is that clauses can be applied any number of times, so the repetitions (or not) of actions are ignored. As a consequence, protocols with temporary secrets cannot be proved secure by ProVerif: when some value first needs to be kept secret and is revealed later in the protocol, the Horn clause model considers that this value can be reused in the beginning of the protocol, thus breaking the protocol. For instance, the process $P_0 = \mathsf{new}\ c; (\mathsf{out}(c, s) \mid \mathsf{in}(c, x); \mathsf{out}(d, c))$ preserves the secrecy of s from $\{d\}$, but ProVerif cannot prove it, because $\mathsf{attacker}(s[\,])$ is derivable from the clauses (Rl), $\mathsf{message}(c[\,], s[\,])$ and $\mathsf{message}(c[\,], x) \Rightarrow \mathsf{attacker}(c[\,])$, which are in $\mathcal{R}_{P_0,\{d\},\{s\}}$. (The fact $\mathsf{message}(d[\,], c[\,])$ is equivalent to $\mathsf{attacker}(c[\,])$ since d is a public channel.) The clauses do not take into account that the output $\mathsf{out}(c, s)$ must have been executed before the adversary obtains the channel c. (In this example, c is the temporary secret.) This example can also be understood by noticing that the generated clauses are the same as for the process $P_0' = \mathsf{new}\ c; (!\mathsf{out}(c, s) \mid !\mathsf{in}(c, x); \mathsf{out}(d, c))$, in which actions are repeated, and which does not preserve the secrecy of s from $\{d\}$.

An additional approximation occurs with outputs on private channels: for the output $\mathsf{out}(M, N); P$, the Horn clause representation considers that the process P can always be executed, as if the process was $\mathsf{out}(M, N) \mid P$, while in fact P can be executed only after sending N on channel M.

3.2.3 Resolution algorithm

The internal protocol representation is a set of Horn clauses, and our goal is to determine whether a given fact can be derived from these clauses or not. This is exactly the problem solved by Prolog systems. However, we cannot use such systems here, because they would not terminate. They use SLD-resolution (Kowalski, 1974), so for instance,

with the goal $\mathsf{attacker}(s[\,])$, the clause

$$\mathsf{attacker}(\mathsf{aenc}(m, \mathsf{pk}(sk))) \wedge \mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(m)$$

leads to considering more and more complex facts

$$\mathsf{attacker}(\mathsf{aenc}(\ldots \mathsf{aenc}(s[\,], \mathsf{pk}(sk_1)) \ldots, \mathsf{pk}(sk_n))$$

with an unbounded number of encryptions. We could of course limit arbitrarily the depth of terms to solve the problem, but we can do much better than that, by using another resolution strategy. We use resolution with free selection (Bachmair and Ganzinger, 2001). We could also use ordered resolution with selection, used by Weidenbach, 1999 and implemented in SPASS (http://www.spass-prover.org/), which is similar but adds ordering constraints. An advantage of implementing our own resolution prover is that we can use domain-specific optimizations, described below, and extensions such as the one needed for proving correspondences (§3.3).

Since a term is secret when a fact is *not* derivable from the clauses, soundness in terms of security (if the verifier claims that there is no attack, then there is no attack) corresponds to the completeness of the resolution algorithm in terms of logic programming (if the algorithm claims that a fact is not derivable, then it is not). The resolution algorithm that we use must therefore be complete.

Basic algorithm

A resolution step combines two clauses $R = H \Rightarrow C$ and $R' = F \wedge H' \Rightarrow C'$ (where F is any hypothesis of R') to infer $R \circ_F R' = \sigma H \wedge \sigma H' \Rightarrow \sigma C'$ where C and F are unifiable and σ is the most general unifier of C and F. Hence, the clause $R \circ_F R'$ is the result of resolving R' with R upon F; it combines R and R', so that R is used in order to prove the hypothesis F of R'. The resolution is guided by a selection function *sel*: $sel(R)$ returns a hypothesis of R or the empty (meaning that the conclusion of R is selected), and the resolution step above is performed only when $sel(R) = \emptyset$ and $sel(R') = \{F\}$, that is, the facts upon which we resolve, the conclusion of R and the hypothesis F of R', are selected. The saturation algorithm $\mathsf{saturate}(\mathcal{R}_0)$ applies these resolution steps to

$\mathcal{R}_0$ until a fixpoint is reached, that is, no new clause is created. When the fixpoint is reached, $\mathsf{saturate}(\mathcal{R}_0)$ returns the subset of the clauses R in the fixpoint such that $sel(R) = \emptyset$.

Resolution with free selection is sound and complete for any selection function, but the choice of this function considerably influences its speed (and its termination). The fact $\mathsf{attacker}(v)$ where v is a variable or a may-fail variable unifies with any fact $\mathsf{attacker}(p)$, so if $\mathsf{attacker}(v)$ is selected, the algorithm will almost never terminate. Therefore, we avoid selecting $\mathsf{attacker}(v)$. Furthermore, disequalities are treated by special simplification steps, shown below, so we never select them. A natural selection function is then:

$$sel_0(H \Rightarrow C) = \begin{cases} \emptyset & \text{if all elements of } H \text{ are disequalities or of the} \\ & \text{form } \mathsf{attacker}(v),\ v \text{ variable or may-fail variable} \\ \{F\} & \text{where } F \text{ is not a disequality,} \\ & F \neq \mathsf{attacker}(v) \text{ and } F \in H, \text{ otherwise} \end{cases}$$

Optimizations

The resolution algorithm uses several optimizations, in order to speed up resolution. We mention the main optimizations in this section. Others are presented in (Blanchet, 2009). These optimizations are applied on the initial clauses and after each resolution step. The first three optimizations are standard, while the last two are domain-specific.

- *Elimination of subsumed clauses.* The clause $H_1 \Rightarrow C_1$ *subsumes* $H_2 \Rightarrow C_2$ if and only if there exists a substitution σ such that $\sigma H_1 \subseteq H_2$ (multiset inclusion) and $\sigma C_1 = C_2$. All clauses subsumed by another clause of the current clause set are removed.

- *Elimination of duplicate hypotheses.* Only one copy of duplicate hypotheses in a clause is kept.

- *Elimination of tautologies.* Tautologies (clauses whose conclusion is already present in the hypotheses) are removed.

- *Elimination of hypotheses* $\mathsf{attacker}(v)$. Hypotheses $\mathsf{attacker}(v)$, where the variable v does not occur elsewhere in the clause, are removed. Indeed, such an hypothesis is always satisfied for some value of v, for instance by (Rn).

- *Decomposition of data constructors.* For each data constructor f, the following clauses are generated:

$$\mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \Rightarrow \mathsf{attacker}(f(x_1, \ldots, x_n)) \quad \text{(Rf)}$$
$$\mathsf{attacker}(f(x_1, \ldots, x_n)) \Rightarrow \mathsf{attacker}(x_i) \quad \text{(Rg)}$$

  Therefore, $\mathsf{attacker}(f(p_1, \ldots, p_n))$ is derivable if and only if $\forall i \in \{1, \ldots, n\}$, $\mathsf{attacker}(p_i)$ is derivable. When a fact of the form $\mathsf{attacker}(f(p_1, \ldots, p_n))$ is met, it is replaced with $\mathsf{attacker}(p_1) \wedge \ldots \wedge \mathsf{attacker}(p_n)$. If this replacement is done in the conclusion of a clause $H \Rightarrow \mathsf{attacker}(f(p_1, \ldots, p_n))$, n clauses are created: $H \Rightarrow \mathsf{attacker}(p_i)$ for each $i \in \{1, \ldots, n\}$. This replacement is of course done recursively: if p_i itself is a data constructor application, it is replaced again. The clauses (Rf) and (Rg) for data constructors are left unchanged.

Treatment of disequalities

The disequalities are simplified using the following transformations, adapted from (Blanchet *et al.*, 2008), where the disequality predicate is written nounif:

- *Elimination of free may-fail variables.* Clauses of the form $H \wedge \forall \widetilde{v}, (mp_1, \ldots, mp_n) \neq (mp'_1, \ldots, mp'_n) \Rightarrow C$ are transformed as follows.

  When mp_i or mp'_i is a may-fail variable u not in $\widetilde{v}$, we replace the clause with two clauses

$$H \wedge \forall \widetilde{v}, (mp_1, \ldots, mp_n)\{^{x}/_{u}\} \neq (mp'_1, \ldots, mp'_n)\{^{x}/_{u}\} \Rightarrow C\,,$$
$$H \wedge \forall \widetilde{v}, (mp_1, \ldots, mp_n)\{^{\mathsf{fail}}/_{u}\} \neq (mp'_1, \ldots, mp'_n)\{^{\mathsf{fail}}/_{u}\} \Rightarrow C\,,$$

  where x is an ordinary variable.

This transformation is repeated until all free may-fail variables have been removed from disequalities. It preserves the semantics of clauses because may-fail variables can be instantiated either with fail or with a pattern, while ordinary variables can be instantiated with a pattern.

- *Unification.* The transformation unify transforms clauses of the form $H \wedge \forall \widetilde{v}, \widetilde{mp} \neq \widetilde{mp}' \Rightarrow C$ as follows. It tries to unify $\widetilde{mp}$ and $\widetilde{mp}'$. If this unification fails, then the clause becomes $H \Rightarrow C$, because $\forall \widetilde{v}, \widetilde{mp} \neq \widetilde{mp}'$ holds when $\sigma\widetilde{mp} \neq \sigma\widetilde{mp}'$ for all σ. Otherwise, unify replaces the clause with

$$H \wedge \forall \widetilde{v}, (v_1, \ldots, v_k) \neq (\sigma v_1, \ldots, \sigma v_k) \Rightarrow C$$

where σ is the most general unifier of $\widetilde{mp}$ and $\widetilde{mp}'$ and $v_1, \ldots, v_k$ are all variables affected by σ. In this unification, σ is built so that all variables in its domain and its image are variables of $\widetilde{mp}$ and $\widetilde{mp}'$, and the variables in its domain do not occur in its image. Note that an instance of $\forall \widetilde{v}, (v_1, \ldots, v_k) \neq (\sigma v_1, \ldots, \sigma v_k)$ holds if and only if the same instance of $\forall \widetilde{v}, \widetilde{mp} \neq \widetilde{mp}'$ does, because $\sigma'\widetilde{mp} = \sigma'\widetilde{mp}'$ if and only if $\sigma'(v_1, \ldots, v_k) = \sigma'\sigma(v_1, \ldots, v_k)$, for all σ'.

For instance, unify transforms the clause

$$H \wedge \forall z, (\mathsf{senc}(x', y'), z') \neq (\mathsf{senc}(z, y), z) \Rightarrow C$$

into

$$H \wedge \forall z, (x', y', z') \neq (z, y, z) \Rightarrow C \tag{3.5}$$

- *Elimination of elements of $\widetilde{v}$.* The transformation elimGVar transforms facts $\forall \widetilde{v}, (mp_1, \ldots, mp_n) \neq (mp'_1, \ldots, mp'_n)$ in clauses obtained after unify, as follows:

  1. When $mp_i = u \in \widetilde{v}$ is a may-fail variable, it eliminates the pair mp_i, mp'_i from the disequality and removes u from $\widetilde{v}$.
  2. When $mp_i = x \in \widetilde{v}$ is a variable and mp'_i is a pattern, it eliminates the pair mp_i, mp'_i from the disequality and removes x from $\widetilde{v}$.

3. Otherwise, when $mp'_i \in \widetilde{v}$, it swaps mp_i and mp'_i everywhere in the disequality $(mp_1, \ldots, mp_n) \neq (mp'_1, \ldots, mp'_n)$, then applies one of the first two cases.

The cases in which mp_i is an ordinary variable and $mp'_i = \mathsf{fail}$, or mp'_i is an ordinary variable and $mp_i = \mathsf{fail}$ cannot happen by the previous application of unify.

We show that, in all cases, an instance of the initial disequality holds if and only if the same instance of the transformed disequality holds.

In Case 1, u does not occur elsewhere in the disequality by the previous application of unify, so for any substitution σ,

$$\sigma(\forall \widetilde{v}, (mp_1, \ldots, mp_n) \neq (mp'_1, \ldots, mp'_n))$$

is equivalent to

$$\sigma(\forall \widetilde{v}', (\forall u, u \neq mp'_i) \vee (mp_1, \ldots, mp_{i-1}, mp_{i+1}, \ldots, mp_n) \neq (mp'_1, \ldots, mp'_{i-1}, mp'_{i+1}, \ldots, mp'_n))$$

which is equivalent to

$$\sigma(\forall \widetilde{v}', (mp_1, \ldots, mp_{i-1}, mp_{i+1}, \ldots, mp_n) \neq (mp'_1, \ldots, mp'_{i-1}, mp'_{i+1}, \ldots, mp'_n))$$

since $\forall u, u \neq \sigma mp'_i$ is false for all σ.

In Case 2, $\forall x, x \neq \sigma mp'_i$ is false, so we can proceed as in Case 1.

In Case 3, swapping preserves the semantics of the disequality, since the unification constraints remain the same. Furthermore, swapping also preserves the property that mp_i does not occur elsewhere.

For instance, elimGVar transforms Clause (3.5) into

$$H \wedge \forall z, (z, y', z') \neq (x', y, x') \Rightarrow C$$

by swapping z and x' (Case 3), then transforms it into

$$H \wedge (y', z') \neq (y, x') \Rightarrow C$$

by Case 2.

- *Detection of failed disequality.* Clauses that contain the hypothesis $\forall \widetilde{v}, () \neq ()$ are removed.

Soundness

The soundness and completeness of this algorithm is justified by the following theorem:

Theorem 3.2. Let F be a closed fact. The fact F is derivable from $\mathcal{R}_0$ if and only if it is derivable from $\mathsf{saturate}(\mathcal{R}_0)$.

This theorem is a particular case of (Blanchet, 2009, Lemma 2), with the treatment of disequalities handled as in (Blanchet *et al.*, 2008, Lemma 36). It shows that one can saturate the clauses by saturate without changing the set of derivable facts. One can then determine which instances of $pred(p_1, \ldots, p_n)$ are derivable by the following computation:

$$\mathsf{solve}_{P_0,\mathcal{N}_{\mathrm{pub}},\mathcal{N}_{\mathrm{priv}}}(pred(p_1, \ldots, p_n)) = \\ \{H \Rightarrow pred(p'_1, \ldots, p'_n) \mid H \Rightarrow pred'(p'_1, \ldots, p'_n) \in \mathsf{saturate}(\mathcal{R}_0)\},$$

where $pred'$ is a new predicate and $\mathcal{R}_0 = \mathcal{R}_{P_0,\mathcal{N}_{\mathrm{pub}},\mathcal{N}_{\mathrm{priv}}} \cup \{pred(p_1, \ldots, p_n) \Rightarrow pred'(p_1, \ldots, p_n)\}$. Indeed, $\sigma pred(p_1, \ldots, p_n)$ is derivable from $\mathcal{R}_{P_0,\mathcal{N}_{\mathrm{pub}},\mathcal{N}_{\mathrm{priv}}}$ if and only if $\sigma pred'(p_1, \ldots, p_n)$ is derivable from $\mathcal{R}_0$, so by Theorem 3.2, if and only if $\sigma pred'(p_1, \ldots, p_n)$ is derivable from $\mathsf{saturate}(\mathcal{R}_0)$, so if and only if there exist a clause $H \Rightarrow pred(p'_1, \ldots, p'_n)$ in $\mathsf{solve}_{P_0,\mathcal{N}_{\mathrm{pub}},\mathcal{N}_{\mathrm{priv}}}(pred(p_1, \ldots, p_n))$ and a substitution σ' such that $\sigma' pred(p'_1, \ldots, p'_n) = \sigma pred(p_1, \ldots, p_n)$ and $\sigma' H$ is derivable from $\mathsf{saturate}(\mathcal{R}_0)$. In particular, if $\mathsf{solve}_{P_0,\mathcal{N}_{\mathrm{pub}},\mathcal{N}_{\mathrm{priv}}}(\mathsf{attacker}(p)) = \emptyset$, then $\mathsf{attacker}(p)$ is not derivable from $\mathcal{R}_{P_0,\mathcal{N}_{\mathrm{pub}},\mathcal{N}_{\mathrm{priv}}}$. Moreover, if the selection function is sel_0 and $\mathsf{solve}_{P_0,\mathcal{N}_{\mathrm{pub}},\mathcal{N}_{\mathrm{priv}}}(\mathsf{attacker}(p))$ is non-empty, then at least one instance of $\mathsf{attacker}(p)$ is derivable. Indeed, in that case $\mathsf{solve}_{P_0,\mathcal{N}_{\mathrm{pub}},\mathcal{N}_{\mathrm{priv}}}(\mathsf{attacker}(p))$ contains a clause $R = H \Rightarrow \mathsf{attacker}(\sigma p)$ with $sel_0(R) = \emptyset$, so the hypothesis H contains facts of the form $\mathsf{attacker}(x)$ and disequalities. Consider a substitution σ' that maps each variable to $b_0[M]$ for a distinct closed term M. Any fact $\sigma' \mathsf{attacker}(x)$ is derivable by (Rn). Using invariants of the clauses and properties of the simplification of disequalities, we can show that for all disequalities F that occur in H, $\sigma' F$ holds. Hence $\sigma' H$ is derivable, so $\mathsf{attacker}(\sigma' \sigma p)$ is derivable.

Termination

In general, the resolution algorithm may not terminate. (The derivability problem is undecidable.) In practice, however, it terminates in most examples.

Blanchet and Podelski, 2005 have shown that it always terminates on a large and interesting class of protocols, the *tagged protocols*. They consider protocols that use as cryptographic primitives only public-key encryption and signatures with atomic keys, shared-key encryption, message authentication codes, and hash functions. Basically, a protocol is tagged when each application of a cryptographic primitive is marked with a distinct constant tag. It is easy to transform a protocol into a tagged protocol by adding tags. For instance, our running example of protocol can be transformed into a tagged protocol, by adding the tags c_0, c_1, c_2 to distinguish the encryptions and signature:

$$\begin{array}{ll} \text{Message 1.} & A \to B : \mathsf{aenc}((c_1, \mathsf{sign}((c_0, k), ssk_A)), pk_B) \\ \text{Message 2.} & B \to A : \mathsf{senc}((c_2, s), k) \end{array}$$

Adding tags preserves the expected behavior of the protocol, that is, the attack-free executions are unchanged. In the presence of attacks, the tagged protocol may be more secure. Hence, tagging is a feature of good protocol design, as explained e.g. in (Abadi and Needham, 1996): the tags are checked when the messages are received; they facilitate the decoding of the received messages and prevent confusions between messages. More formally, tagging prevents type-flaw attacks (Heather *et al.*, 2000), which occur when a message is taken for another message. Hence, the tagged protocol is potentially more secure than its untagged version, so, in other words, a proof of security for the tagged protocol does not imply the security of its untagged version.

We illustrate the effect of tagging on the Needham-Schroeder shared-key protocol (Needham and Schroeder, 1978). The algorithm does not terminate on its original version, which is untagged. It terminates after adding tags. This protocol contains the following messages:

$$\begin{array}{lll} \text{Message 4.} & B \to A : & \mathsf{senc}(b, k) \\ \text{Message 5.} & A \to B : & \mathsf{senc}(b - 1, k) \end{array}$$

where b is a nonce. Representing $b-1$ using a function $\mathsf{minusone}(x) = x-1$, the algorithm does not terminate. Indeed, the output of message 5 is represented by a clause of the form:

$$H \wedge \mathsf{attacker}(\mathsf{senc}(x, y)) \Rightarrow \mathsf{attacker}(\mathsf{senc}(\mathsf{minusone}(x), y))$$

where the hypothesis H describes other messages previously received by A. After some resolution steps, we obtain a clause of the form

$$\mathsf{attacker}(\mathsf{senc}(x, M_k)) \Rightarrow \mathsf{attacker}(\mathsf{senc}(\mathsf{minusone}(x), M_k)) \qquad \text{(Loop)}$$

for some term M_k that represents the key k.

The fact $\mathsf{attacker}(\mathsf{senc}(\mathsf{minusone}(M_b), M_k))$ is also derived for some term M_b that represents the nonce b, so by resolution with (Loop), we derive: $\mathsf{attacker}(\mathsf{senc}(\mathsf{minusone}(\mathsf{minusone}(M_b)), M_k))$. This fact can again be resolved with (Loop), so that we obtain a cycle that derives $\mathsf{attacker}(\mathsf{senc}(\mathsf{minusone}^n(M_b), M_k))$ for all n.

When tags are added, the clause (Loop) becomes:

$$\mathsf{attacker}(\mathsf{senc}((c_1, x), M_k)) \Rightarrow \mathsf{attacker}(\mathsf{senc}((c_2, \mathsf{minusone}(x)), M_k)) \qquad \text{(NoLoop)}$$

and the previous loop disappears because c_2 does not unify with c_1. The fact $\mathsf{attacker}(\mathsf{senc}((c_2, \mathsf{minusone}(M_b)), M_k))$ is derived, without yielding a loop.

Other authors have proved related results: Ramanujam and Suresh, 2003 have shown that secrecy is decidable for tagged protocols. However, their tagging scheme is stronger since it forbids blind copies. A blind copy happens when a protocol participant sends back part of a message he received without looking at what is contained inside this part. On the other hand, they obtain a decidability result, while Blanchet and Podelski, 2005 obtain a termination result for an algorithm which is sound, efficient in practice, but approximate. Arapinis and Duflot, 2007 show that, for a class named "well-formed protocols", it is enough to consider well-typed attacks; this theorem allows them to extend the decidability result of Ramanujam and Suresh, 2003, still forbidding blind copies. Comon-Lundh and Cortier, 2003 show that an algorithm using ordered binary resolution, ordered factorization and splitting terminates on protocols that blindly copy at most one term in each message. In

contrast, the result of Blanchet and Podelski, 2005 puts no limit on the number of blind copies, but requires tagging.

For protocols that are not tagged, heuristics have been designed to adapt the selection function in order to obtain termination more often. We refer the reader to (Blanchet, 2009, §8.2) for more details.

It is also possible to obtain termination in all cases at the cost of additional abstractions. For instance, Goubault-Larrecq, 2005 shows that one can abstract the clauses into clauses in the decidable class $\mathcal{H}_1$, by losing some relational information on the messages.

3.3 Correspondences

Correspondences are properties of the form "if an event has been executed, then other events have been executed." Properties on the ordering of execution of events are also studied in other contexts, for instance in software verification to check that a file is always opened before being closed or that a lock is always acquired before being released. However, in the case of protocols, the events we want to relate are typically in different parallel processes, which complicates the proof.

To model correspondence properties, we introduce an additional construct in our process calculus, namely $\mathsf{event}(M); P$, which executes event M, then process P. The semantics of this construct is simply defined by

$$E, \mathcal{P} \cup \{\, \mathsf{event}(M); P \,\} \to E, \mathcal{P} \cup \{\, P \,\} \qquad \text{(Red Event)}$$

The $\mathcal{N}_{\text{pub}}$-adversaries are restricted to processes that do not contain events (otherwise, no correspondence could be proved). We can then define the fact that a trace executes an event:

Definition 3.4. Let M be a closed term. We say that a trace $Tr = E_0, \mathcal{P}_0 \to^* E', \mathcal{P}'$ *executes event* M if and only if it contains a reduction $E, \mathcal{P} \cup \{\, \mathsf{event}(M); P \,\} \to E, \mathcal{P} \cup \{\, P \,\}$ for some E, $\mathcal{P}$, and P.

The correspondence $\mathsf{event}(M) \rightsquigarrow \bigvee_{j=1}^{m} \bigwedge_{k=1}^{l_j} \mathsf{event}(M_{jk})$ means intuitively that, if event M has been executed, then there exists j such that events $M_{j1}, \ldots, M_{jl_j}$ have been executed. More precisely, for any

value of the variables of M, if event M has been executed, then there exists j and values of the variables of $M_{j1}, \ldots, M_{jl_j}$ that do not appear in M such that events $M_{j1}, \ldots, M_{jl_j}$ have been executed. The formal definition is as follows:

Definition 3.5. The closed process P_0 *satisfies the correspondence*

$$\mathsf{event}(M) \rightsquigarrow \bigvee_{j=1}^{m} \bigwedge_{k=1}^{l_j} \mathsf{event}(M_{jk})$$

against $\mathcal{N}_{\mathrm{pub}}$-adversaries if and only if, for some $\mathcal{N}_{\mathrm{priv}}$ disjoint from $\mathcal{N}_{\mathrm{pub}}$ such that $fn(P_0) \cup fn(M) \cup \bigcup_{j,k} fn(M_{jk}) \subseteq \mathcal{N}_{\mathrm{pub}} \cup \mathcal{N}_{\mathrm{priv}}$, for any $\mathcal{N}_{\mathrm{pub}}$-adversary Q, for any trace $Tr = (\mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}), \{P_0, Q\} \rightarrow^* E', \mathcal{P}'$, for any substitution σ, if Tr executes event σM, then there exist σ' and $j \in \{1, \ldots, m\}$ such that $\sigma' M = \sigma M$ and, for all $k \in \{1, \ldots, l_j\}$, Tr executes event $\sigma' M_{jk}$.

Example 3.3. For example, we can modify the process P_0 of §2.2 by adding events as follows:

$$\begin{aligned}
P_0 = {} & \mathsf{new}\ ssk_A : \mathsf{skey}; \mathsf{new}\ sk_B : \mathsf{skey}; \mathsf{let}\ spk_A = \mathsf{pk}(ssk_A)\ \mathsf{in} \\
& \mathsf{let}\ pk_B = \mathsf{pk}(sk_B)\ \mathsf{in}\ \mathsf{out}(c, spk_A); \mathsf{out}(c, pk_B); \\
& (P_A(spk_A, ssk_A) \mid P_B(pk_B, sk_B, spk_A)) \\
P_A(spk_A, ssk_A) = {} & !\ \mathsf{in}(c, x_{pk_B} : \mathsf{pkey}); \mathsf{new}\ k : \mathsf{key}; \\
& \mathsf{event}(e_A(spk_A, x_{pk_B}, k)); \\
& \mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}(\mathsf{k2b}(k), ssk_A), x_{pk_B})); \\
& \mathsf{in}(c, x : \mathsf{bitstring}); \mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ \mathbf{0} \\
P_B(pk_B, sk_B, spk_A) = {} & !\ \mathsf{in}(c, y : \mathsf{bitstring}); \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in} \\
& \mathsf{let}\ x_k = \mathsf{b2k}(\mathsf{check}(y', spk_A))\ \mathsf{in} \\
& \mathsf{event}(e_B(spk_A, pk_B, x_k)); \mathsf{out}(c, \mathsf{senc}(s, x_k))
\end{aligned}$$

The event $e_A(spk_A, x_{pk_B}, k)$ intuitively means that A started a session of the protocol between the participants with public keys spk_A (that is, A) and x_{pk_B}, with the shared key k. Similarly, the event $e_B(spk_A, pk_B, x_k)$ means that B accepted the shared key k in a session between participants A and B. We can then try to show the correspondence $\mathsf{event}(e_B(x, y, z)) \rightsquigarrow \mathsf{event}(e_A(x, y, z))$, which means that, if

$e_B(x, y, z)$ has been executed, then $e_A(x, y, z)$ has also been executed. In other words, if B thinks he runs a session of the protocol with A and the shared key z, then A thinks she runs a session of the protocol with B and the same key z. This is an authentication property.

As mentioned above, the Horn clause verification method overapproximates the actions that are executed: when $\mathsf{attacker}(p)$ is derivable from the clauses, the adversary *may* have p, that is, if $\mathsf{attacker}(p)$ is not derivable from the clauses, then the adversary cannot have p, but the converse is not true. Let us now consider the proof of a correspondence, for instance $\mathsf{event}(e_1(x)) \rightsquigarrow \mathsf{event}(e_2(x))$, that is, we want to show that, if $e_1(x)$ has been executed, then $e_2(x)$ has been executed. In order to prove this correspondence, we can overapproximate the executions of event e_1: if we prove the correspondence with this overapproximation, it will also hold in the exact semantics. So we can easily extend our analysis for secrecy with an additional predicate event, such that $\mathsf{event}(p)$ means that event p (more formally, event M with corresponding pattern p) may have been executed. We generate clauses $\mathsf{message}(p_1, p_1') \wedge \ldots \wedge \mathsf{message}(p_n, p_n') \Rightarrow \mathsf{event}(p)$ when the process executes event p after receiving messages $p_1', \ldots, p_n'$ on channels $p_1, \ldots, p_n$ respectively. However, such an overapproximation cannot be done for the event e_2: if we prove the correspondence after overapproximating the execution of e_2, we are not really sure that e_2 will be executed, so the correspondence may be wrong in the exact semantics. For instance, assuming c is public, the process

$$\mathsf{in}(c, x); \mathsf{event}(e_1(x)) \mid \mathsf{in}(c, x'); \mathsf{event}(e_2(x'))$$

may execute $e_1(x)$ and $e_2(x)$ for all x that the adversary has, but the correspondence $\mathsf{event}(e_1(x)) \rightsquigarrow \mathsf{event}(e_2(x))$ does not hold for this process: there is a trace in which it executes $e_1(c)$ but does not execute $e_2(c)$. Therefore, we have to use a different method for treating e_2.

We use the following idea: we fix the exact set Ev of allowed events $e_2(p)$ and, in order to prove $\mathsf{event}(e_1(x)) \rightsquigarrow \mathsf{event}(e_2(x))$, we check that only events $e_1(p)$ for p such that $e_2(p) \in Ev$ can be executed. Therefore, if $e_1(M)$ has been executed, then $e_1(p)$ has been executed with p the pattern corresponding to M, so $e_2(p) \in Ev$ has been executed, so $e_2(M)$

has been executed, since a single term M corresponds to a given pattern p thanks to session identifiers, which allow us to distinguish the names created by the same restriction. If we prove this property for any value of Ev, we have proved the desired correspondence. So we introduce a predicate $\mathsf{m\text{-}event}$ (*must event*), such that $\mathsf{m\text{-}event}(p_0)$ is true if and only if $p_0 \in Ev$. We generate clauses $\mathsf{message}(p_1, p_1') \wedge \ldots \wedge \mathsf{message}(p_n, p_n') \wedge \mathsf{m\text{-}event}(p_0) \Rightarrow \mathsf{message}(p, p')$ when the process outputs p' on channel p after executing event p_0 and receiving messages $p_1', \ldots, p_n'$ on channels $p_1, \ldots, p_n$ respectively. In other words, the output of p' on channel p can be executed only when $\mathsf{m\text{-}event}(p_0)$ is true, that is, $p_0 \in Ev$.

More generally, we extend the computation of Horn clauses to events as follows:

$$[\![\mathsf{event}(M); P]\!]\rho s H = [\![P]\!]\rho s(H \wedge \mathsf{m\text{-}event}(\rho(M))) \cup \{H \wedge \mathsf{m\text{-}event}(\rho(M)) \Rightarrow \mathsf{event}(\rho(M))\}$$

We add the hypothesis $\mathsf{m\text{-}event}(\rho(M))$ to H to express that P can be executed only if event M is allowed (which is useful for e_2 in the example above), and we add the clause $H \wedge \mathsf{m\text{-}event}(\rho(M)) \Rightarrow \mathsf{event}(\rho(M))$ to express that the event may be executed if H is true and the event is allowed (which is useful for e_1 in the example above).

In order to determine whether an event can be executed, we determine whether the corresponding fact is derivable from the clauses, by extending the previous resolution-based algorithm. Indeed, the resolution must be performed for an unknown value of Ev. So, basically, we keep $\mathsf{m\text{-}event}$ facts without trying to evaluate them (which we cannot do since Ev is unknown). To do that, we modify the selection function so that it never selects a fact of the form $\mathsf{m\text{-}event}(p)$. We do not know of an off-the-shelf first-order resolution prover that implements this feature, though it would probably not be difficult to add to existing provers. Letting $\mathcal{F}_{\mathrm{me}} = \{\mathsf{m\text{-}event}(p) \mid p \in Ev\}$, Theorem 3.2 becomes:

Theorem 3.3. Let F be a closed fact. The fact F is derivable from $\mathcal{R}_0 \cup \mathcal{F}_{\mathrm{me}}$ if and only if it is derivable from $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\mathrm{me}}$.

As for secrecy, we define $\mathcal{R}_0 = \mathcal{R}_{P_0, \mathcal{N}_{\mathrm{pub}}, \mathcal{N}_{\mathrm{priv}}} \cup \{pred(p_1, \ldots, p_n) \Rightarrow pred'(p_1, \ldots, p_n)\}$. We have that $\sigma pred(p_1, \ldots, p_n)$ is derivable from

$\mathcal{R}_{P_0,\mathcal{N}_{\text{pub}},\mathcal{N}_{\text{priv}}} \cup \mathcal{F}_{\text{me}}$ if and only if there exist a clause $H \Rightarrow pred(p'_1, \ldots, p'_n)$ in $\mathsf{solve}_{P_0,\mathcal{N}_{\text{pub}},\mathcal{N}_{\text{priv}}}(pred(p_1, \ldots, p_n))$ and a substitution σ' such that $\sigma' pred(p'_1, \ldots, p'_n) = \sigma pred(p_1, \ldots, p_n)$ and $\sigma' H$ is derivable from $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\text{me}}$. Since no clause outside $\mathcal{F}_{\text{me}}$ concludes $\mathsf{m\text{-}event}$ facts, the $\mathsf{m\text{-}event}$ facts in $\sigma' H$ must be in $\mathcal{F}_{\text{me}}$, which guarantees that the corresponding events have been executed. We can then show the following theorem:

Theorem 3.4. Let P_0 be a closed process and $\mathcal{N}_{\text{pub}}$ be a set of names. Let M, M_{jk} ($j \in \{1, \ldots, m\}$, $k \in \{1, \ldots, l_j\}$) be terms. Let p, p_{jk} be the patterns obtained by replacing names a with patterns $a[\,]$ in terms M, M_{jk} respectively. Let $\mathcal{N}_{\text{priv}}$ be a set of names disjoint from $\mathcal{N}_{\text{pub}}$ such that $fn(P_0) \cup fn(M) \cup \bigcup_{j,k} fn(M_{jk}) \subseteq \mathcal{N}_{\text{pub}} \cup \mathcal{N}_{\text{priv}}$. Suppose that, for all clauses $R \in \mathsf{solve}_{P_0,\mathcal{N}_{\text{pub}},\mathcal{N}_{\text{priv}}}(\mathsf{event}(p))$, there exist $j \in \{1, \ldots, m\}$, σ' and H such that $R = H \wedge \mathsf{m\text{-}event}(\sigma' p_{j1}) \wedge \ldots \wedge \mathsf{m\text{-}event}(\sigma' p_{jl_j}) \Rightarrow \mathsf{event}(\sigma' p)$. Then P_0 satisfies the correspondence $\mathsf{event}(M) \rightsquigarrow \bigvee_{j=1}^{m} \bigwedge_{k=1}^{l_j} \mathsf{event}(M_{jk})$ against $\mathcal{N}_{\text{pub}}$-adversaries.

This result is a particular case of Blanchet, 2009, Theorem 4. Intuitively, if all clauses are of the form $H \wedge \mathsf{m\text{-}event}(\sigma' p_{j1}) \wedge \ldots \wedge \mathsf{m\text{-}event}(\sigma' p_{jl_j}) \Rightarrow \mathsf{event}(\sigma' p)$, then, in order to derive $\mathsf{event}(\sigma' p)$, the facts $\mathsf{m\text{-}event}(\sigma' p_{j1})$, …, $\mathsf{m\text{-}event}(\sigma' p_{jl_j})$ must be true, so in order to execute event $\sigma' p$, the events $\sigma' p_{j1}$, …, $\sigma' p_{jl_j}$ must have been executed, which proves the desired correspondence.

Example 3.4. For the process P_0 of Example 3.3, the set of clauses $\mathsf{solve}_{P_0,\mathcal{N}_{\text{pub}},\mathcal{N}_{\text{priv}}}(\mathsf{event}(e_B(x, y, z)))$ contains

$$\mathsf{m\text{-}event}(e_A(\mathsf{pk}(ssk_A[\,]), \mathsf{pk}(y'), k[i, \mathsf{pk}(y')])) \wedge \mathsf{attacker}(y') \Rightarrow \mathsf{event}(e_B(\mathsf{pk}(ssk_A[\,]), \mathsf{pk}(sk_B[\,]), k[i, \mathsf{pk}(y')]))\,.$$

This clause prevents us from applying Theorem 3.4 to prove the correspondence $\mathsf{event}(e_B(x, y, z)) \rightsquigarrow \mathsf{event}(e_A(x, y, z))$, because the fact $\mathsf{m\text{-}event}(e_A(\mathsf{pk}(ssk_A[\,]), \mathsf{pk}(y'), k[i, \mathsf{pk}(y')]))$ contains $\mathsf{pk}(y')$ instead of $\mathsf{pk}(sk_B[\,])$. This point corresponds again to the known attack against this protocol: A executes a session with C, which has secret key y' and public key $\mathsf{pk}(y')$, while B thinks he executes a session with A. In

contrast, for the fixed version of the protocol with the public key of B added inside the signature as in Example 2.5,

$$\begin{aligned}&\mathsf{solve}_{P_0,\mathcal{N}_{\mathrm{pub}},\mathcal{N}_{\mathrm{priv}}}(\mathsf{event}(e_B(x,y,z))) = \\ &\quad\{\mathsf{m\text{-}event}(e_A(\mathsf{pk}(ssk_A[\,]),\mathsf{pk}(sk_B[\,]),k[i,\mathsf{pk}(sk_B[\,])])) \Rightarrow \\ &\qquad\mathsf{event}(e_B(\mathsf{pk}(ssk_A[\,]),\mathsf{pk}(sk_B[\,]),k[i,\mathsf{pk}(sk_B[\,])]))\}\,,\end{aligned}$$

so the desired correspondence is proved.

We extended these results to injective correspondences, that is, correspondences that additionally require that each execution of event M corresponds to a *distinct* execution of events M_{jk}. The proof of injectivity relies on session identifiers to distinguish different executions of the same event. We also extended this work to nested correspondences, which can express constraints on the order in which events are executed (Blanchet, 2009, §7.2).

Attack reconstruction was also extended to non-injective and injective correspondences, and it reconstructs the attack of Example 3.4. The extension to injective correspondences presents an additional difficulty: the derivation corresponds to a single execution of event M while, to contradict injectivity, event M must be executed twice and an event M_{jk} must be executed at most once. To solve this problem, we consider two copies of the derivation with variables renamed to distinct variables, and unify some events M_{jk} between these two copies so that they correspond to a single event. After this unification, the two copies of the derivation contain the unified events M_{jk} with the same session identifiers, but still contain event M with different session identifiers, because the derivation does not prove injectivity. We execute the attack reconstruction algorithm with these two copies of the derivation. That allows this algorithm to execute most actions (including event M) twice, but not the unified events M_{jk}, because they occur with the same session identifier in both copies of the derivation. We can then look for a trace that executes event M twice, while still executing the unified events M_{jk} once.

3.4 Equivalences

The notion of process equivalence is the main reasoning tool introduced initially with the spi calculus (Abadi and Gordon, 1999; Abadi and Gordon, 1998) and the applied pi calculus (Abadi and Fournet, 2001; Abadi *et al.*, 2016), with manual proofs. Intuitively, two processes are equivalent when the adversary cannot distinguish them. Proving equivalences is more delicate than proving trace properties, because we need to consider relations between traces or semantic configurations, instead of a single trace. In general, equivalence is undecidable (Hüttel, 2003; Abadi and Cortier, 2006), so automated techniques are incomplete. For a bounded number of sessions, several tools can decide trace equivalence: SPEC (Tiu and Dawson, 2010) for fixed primitives and without else branches, APTE (Cheval *et al.*, 2011) for fixed primitives with else branches and non-determinism, and AKISS (Chadha *et al.*, 2012; Ciobâcă, 2011) for a wide variety of primitives and determinate processes, that is, processes whose execution is entirely determined by the adversary inputs. For an unbounded number of sessions, decision procedures exist for restricted classes of protocols: there is a decision procedure for trace equivalence for symmetric-key, type-compliant, acyclic protocols (Chrétien *et al.*, 2015a), which is too complex for practical implementation, and for ping-pong protocols (Chrétien *et al.*, 2015b), which is implemented in a tool. ProVerif also proves a restricted class of equivalences, explained below.

Next, we give a formal definition of observational equivalence. This definition, inspired by Baudet, 2007 and Arapinis *et al.*, 2014, relies on the semantics with configurations while our previous work (Blanchet, 2004; Blanchet *et al.*, 2008) uses a semantics closer to the applied pi calculus (see Chapter 4).

We consider configurations $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}$ as equal modulo any bijective renaming of names that leaves names in $\mathcal{N}_{\text{pub}}$ unchanged. It is easy to see that such renamings commute with reduction, so all configurations in a trace are just transformed by such a renaming. Hence the semantics is not significantly altered. Allowing such renamings is helpful in order to define the application of a context to a configuration below. However, it would be problematic in our definition of correspondences,

since names appear inside events and these names must not be modified between executions of several events that appear in a correspondence.

A configuration $\mathcal{C} = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}$ *can output on a channel* N, denoted, $\mathcal{C} \downarrow_N$, if there exists $\mathsf{out}(N, M); P \in \mathcal{P}$ with $fn(N) \subseteq \mathcal{N}_{\text{pub}}$, for some term M and process P. For equivalences, the adversary is represented by an *adversarial context*. A context $C[\_]$ is a process with a hole, written $\_$. Let adversarial contexts be contexts of the form $\mathsf{new}\ \widetilde{n}; (\_ \mid Q)$ with $fv(Q) = \emptyset$ and all function symbols in Q are public. When $\mathcal{C} = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}$ and $C[\_] = \mathsf{new}\ \widetilde{n}; (\_ \mid Q)$ is an adversarial context, we define the application of context $C[\_]$ to the configuration $\mathcal{C}$ by $C[\mathcal{C}] = ((\mathcal{N}_{\text{pub}} \cup fn(Q)) \setminus \{\widetilde{n}\}, \mathcal{N}_{\text{priv}} \cup \{\widetilde{n}\}), \mathcal{P} \cup \{Q\}$, after renaming the names in $\mathcal{N}_{\text{priv}}$ so that $\mathcal{N}_{\text{priv}} \cap fn(Q) = \emptyset$. The condition $\mathcal{N}_{\text{priv}} \cap fn(Q) = \emptyset$ avoids clashes between the private names of the configuration $\mathcal{C}$ and the names of the adversary. To compute $C[\mathcal{C}]$, we add the process Q to the multiset $\mathcal{P}$ of processes of the configuration $\mathcal{C}$, so that Q runs in parallel with these processes. The free names of Q are public, so we add them to $\mathcal{N}_{\text{pub}}$. Further, we apply the restriction $\mathsf{new}\ \widetilde{n}$ by moving the names $\widetilde{n}$ from $\mathcal{N}_{\text{pub}}$ to $\mathcal{N}_{\text{priv}}$.

Definition 3.6 (Observational equivalence)**.** *Observational equivalence* between configurations $\approx$ is the largest symmetric relation $\mathcal{R}$ between valid configurations such that $\mathcal{C}\ \mathcal{R}\ \mathcal{C}'$ implies:

1. if $\mathcal{C} \downarrow_a$, then $\mathcal{C}' \rightarrow^* \downarrow_a$, for all names a;
2. if $\mathcal{C} \rightarrow \mathcal{C}_1$, then $\mathcal{C}' \rightarrow^* \mathcal{C}'_1$ and $\mathcal{C}_1\ \mathcal{R}\ \mathcal{C}'_1$, for some $\mathcal{C}'_1$.
3. $C[\mathcal{C}]\ \mathcal{R}\ C[\mathcal{C}']$ for all adversarial contexts $C[\_]$.

We define observational equivalence on semantic configurations. Item 1 guarantees that, if a configuration $\mathcal{C}$ emits on a public channel, then so does $\mathcal{C}'$. Otherwise, an adversary could distinguish them immediately. Item 2 guarantees that this property is preserved by reduction, and Item 3 guarantees that it is preserved in the presence of an adversary. For proving observational equivalence in general, it is not enough to consider traces of one process, we need to relate the configurations of two processes together. For this reason, the proof of observational equivalence is difficult to automate, so ProVerif was designed to prove

classes of equivalences that are easier to handle, but still interesting in practice.

The simplest class of equivalences that ProVerif verifies is strong secrecy (in the case without equational theory) (Blanchet, 2004): strong secrecy means that the adversary cannot distinguish two versions of the protocol that use different values of the secret. This property is defined as follows:

Definition 3.7. The closed process P_0 *preserves the strong secrecy of* s with private names $\mathcal{N}_{\text{priv}}$ if and only if for all closed terms M and N with $(fn(M) \cup fn(N)) \cap \mathcal{N}_{\text{priv}} = \emptyset$, for some $\mathcal{N}_{\text{pub}}$ disjoint from $\mathcal{N}_{\text{priv}}$ such that $fn(P_0) \cup fn(M) \cup fn(N) \subseteq \mathcal{N}_{\text{pub}} \cup \mathcal{N}_{\text{priv}}$, we have $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), P_0\{^M/_s\} \approx (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), P_0\{^N/_s\}$.

To prove strong secrecy, ProVerif relies on a trace property: it shows that no test (destructor application, communication on a channel) that gives a different result for different values of the secret is reachable. This reachability property is encoded by Horn clauses, with an additional predicate used for testing whether a unification succeeds for some values of the secret and not for others.

Example 3.5. ProVerif cannot prove that the protocol of §2.2 preserves the strong secrecy of s. This is not surprising since it does not even preserve the secrecy of s. ProVerif proves that the fixed protocol of Example 2.5 preserves the strong secrecy of s: not only the adversary cannot compute s, but also it cannot distinguish when the value of s changes.

ProVerif can also verify a more general class of equivalences, whose proof is also more difficult: the equivalences between processes P and Q that differ only by the terms they contain (Blanchet *et al.*, 2008). These equivalences are again proved by relying on a trace property, for a process that represents both P and Q. This idea extends the technique of (Pottier and Simonet, 2002; Pottier, 2002) for information flow (without cryptography) to the case of security protocols.

ProVerif represents pairs of processes that differ only by the terms they contain as *biprocesses*. The grammar of biprocesses is an extension

of the grammar of Figure 2.1, with the additional cases $\mathsf{diff}[M, M']$ for terms and $\mathsf{diff}[D, D']$ for expressions. The $\mathcal{N}_{\text{pub}}$-adversaries and the adversarial contexts do not contain diff. Given a biprocess P, we define two processes $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$, as follows: $\mathsf{fst}(P)$ is obtained by replacing all occurrences of $\mathsf{diff}[M, M']$ with M and $\mathsf{diff}[D, D']$ with D in P, and $\mathsf{snd}(P)$ is obtained by replacing $\mathsf{diff}[M, M']$ with M' and $\mathsf{diff}[D, D']$ with D' in P. We define $\mathsf{fst}(M)$, $\mathsf{snd}(M)$, $\mathsf{fst}(D)$, and $\mathsf{snd}(D)$ similarly. We naturally extend fst and snd to multisets of processes, and to configurations by $\mathsf{fst}(E, \mathcal{P}) = E, \mathsf{fst}(\mathcal{P})$ and $\mathsf{snd}(E, \mathcal{P}) = E, \mathsf{snd}(\mathcal{P})$. Our goal is to show that the processes $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$ are observationally equivalent.

Definition 3.8. The closed biprocess P_0 *satisfies observational equivalence* with private names $\mathcal{N}_{\text{priv}}$ if and only if, for some $\mathcal{N}_{\text{pub}}$ disjoint from $\mathcal{N}_{\text{priv}}$ such that $fn(P_0) \subseteq \mathcal{N}_{\text{pub}} \cup \mathcal{N}_{\text{priv}}$, we have that $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \{\mathsf{fst}(P_0)\} \approx (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \{\mathsf{snd}(P_0)\}$.

The semantics of biprocesses is defined as in Figure 2.4, except that the rules (Red I/O), (Red Eval 1), and (Red Eval 2) are as follows:

$$E, \mathcal{P} \cup \{\, \mathsf{out}(N, M); Q, \mathsf{in}(N', x); P \,\} \to E, \mathcal{P} \cup \{\, Q, P\{M/x\} \,\}$$
$$\text{if } \mathsf{fst}(N) = \mathsf{fst}(N') \text{ and } \mathsf{snd}(N) = \mathsf{snd}(N') \qquad \text{(Red I/O)}$$

$$E, \mathcal{P} \cup \{\, \mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \,\} \to E, \mathcal{P} \cup \{\, P\{\mathsf{diff}[M, M']/x\} \,\}$$
$$\text{if } \mathsf{fst}(D) \Downarrow M \text{ and } \mathsf{snd}(D) \Downarrow M' \qquad \text{(Red Eval 1)}$$

$$E, \mathcal{P} \cup \{\, \mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \,\} \to E, \mathcal{P} \cup \{\, Q \,\}$$
$$\text{if } \mathsf{fst}(D) \Downarrow \mathsf{fail} \text{ and } \mathsf{snd}(D) \Downarrow \mathsf{fail} \qquad \text{(Red Eval 2)}$$

and the rules for the conditional are omitted because the conditional $\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$ can be encoded as an expression evaluation $\mathsf{let}\ x = \mathsf{istrue}(M)\ \mathsf{in}\ P\ \mathsf{else}\ Q$, where x is a fresh variable and $\mathsf{istrue}(\mathsf{bool}) : \mathsf{bool}$ is a destructor defined by the rewrite rule $\mathsf{istrue}(\mathsf{true}) \to \mathsf{true}$. A biprocess P reduces by this semantics when its two components $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$ reduce in the same way: a communication is executed when the channel is the same in the input and in the output for both components; an expression evaluation succeeds (resp. fails) when it succeeds (resp. fails) for both components. In particular, if $\mathcal{C} \to \mathcal{C}'$, then $\mathsf{fst}(\mathcal{C}) \to \mathsf{fst}(\mathcal{C}')$ and $\mathsf{snd}(\mathcal{C}) \to \mathsf{snd}(\mathcal{C}')$.

When the two components do not reduce in the same way, we say that the configuration $\mathcal{C}$ *diverges*, and we write $\mathcal{C} \uparrow$ (vocabulary and notation from (Baudet, 2007)):

$$E, \mathcal{P} \cup \{ \mathsf{out}(N, M); Q, \mathsf{in}(N', x); P \} \uparrow \quad \text{if } (\mathsf{fst}(N) = \mathsf{fst}(N')) \not\Leftrightarrow (\mathsf{snd}(N) = \mathsf{snd}(N')) \qquad \text{(Div I/O)}$$

$$E, \mathcal{P} \cup \{ \mathsf{let}\ x = D\ \mathsf{in}\ P\ \mathsf{else}\ Q \} \uparrow \quad \text{if } \mathsf{fst}(D) \Downarrow \mathsf{fail} \not\Leftrightarrow \mathsf{snd}(D) \Downarrow \mathsf{fail} \qquad \text{(Div Eval)}$$

When the configuration $\mathcal{C}$ does not diverge, the two components always reduce in the same way, that is, the biprocess reduces. Formally, if $\mathcal{C}$ does not diverge and $\mathsf{fst}(\mathcal{C}) \to \mathcal{C}_1$, then $\mathcal{C} \to \mathcal{C}'$ and $\mathcal{C}_1 = \mathsf{fst}(\mathcal{C}')$ for some $\mathcal{C}'$, and similarly for snd. If no reachable configuration diverges, then the two components of the considered biprocess P_0 always reduce in the same way. This property is formally defined as follows:

Definition 3.9. Let P_0 be a closed biprocess and $\mathcal{N}_{\text{priv}}$ be a set of names. The biprocess P_0 satisfies *diff-equivalence* with private names $\mathcal{N}_{\text{priv}}$ if, for some $\mathcal{N}_{\text{pub}}$ disjoint from $\mathcal{N}_{\text{priv}}$ such that $fn(P_0) \subseteq \mathcal{N}_{\text{pub}} \cup \mathcal{N}_{\text{priv}}$, for all $\mathcal{N}_{\text{pub}}$-adversaries Q, there exists no configuration $\mathcal{C}$ such that $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \{P_0, Q\} \to^* \mathcal{C} \uparrow$.

Diff-equivalence implies observational equivalence, as shown by the following theorem:

Theorem 3.5. Let P_0 be a closed biprocess and $\mathcal{N}_{\text{priv}}$ be a set of names. If P_0 satisfies diff-equivalence with private names $\mathcal{N}_{\text{priv}}$, then P_0 satisfies observational equivalence with private names $\mathcal{N}_{\text{priv}}$.

Since the formalism is fairly different from the one used in our previous work (Blanchet *et al.*, 2008), we provide a proof of this result in Appendix A. Diff-equivalence is a trace property on biprocesses. By Theorem 3.5, it suffices in order to obtain observational equivalence. It is however not necessary: for instance, if $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), P \approx (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), P'$, then the biprocess $\mathsf{if}\ \mathsf{diff}[\mathsf{true}, \mathsf{false}]\ \mathsf{then}\ P\ \mathsf{else}\ P'$ satisfies observational equivalence with private names $\mathcal{N}_{\text{priv}}$, but Theorem 3.5 does not allow us to prove it (because this biprocess diverges immediately). The property on the traces of biprocesses is encoded into Horn clauses as

before. However, in order to represent the semantics of biprocesses, we use facts $\mathsf{attacker}'(p_1, p_2)$ and $\mathsf{message}'(p_1, p_1', p_2, p_2')$, where components with index 1 correspond to $\mathsf{fst}(P)$ and those with index 2 correspond to $\mathsf{snd}(P)$, instead of $\mathsf{attacker}'(p)$ and $\mathsf{message}'(p, p')$. The fact $\mathsf{attacker}'(p_1, p_2)$ means that, by the same actions, the adversary obtains p_1 by interacting with $\mathsf{fst}(P)$ and p_2 by interacting with $\mathsf{snd}(P)$. The fact $\mathsf{message}'(p_1, p_1', p_2, p_2')$ means that, by the same actions, $\mathsf{fst}(P)$ sends message p_1' on channel p_1 while $\mathsf{snd}(P)$ sends message p_2' on channel p_2. We also use the fact $\mathsf{input}'(p_1, p_2)$ to express that an input on channel p_1 may be executed by $\mathsf{fst}(P)$ while $\mathsf{snd}(P)$ executes an input on p_2. (This situation allows the adversary to test the equality of these channels with those used in an output.) We can then encode the desired trace property on biprocesses into clauses and prove it by resolution, which allows us to apply Theorem 3.5.

Example 3.6. We may consider the property that the secret s exchanged in our running example is indistinguishable from a fresh name, representing intuitively a fresh random value. This notion of secrecy is the symbolic analog of real-or-random secrecy in the computational model (Abdalla *et al.*, 2005). This property can be verified in ProVerif by replacing the final output of the process P_B of §2.2 with

$$\mathsf{new}\ r : \mathsf{bitstring}; \mathsf{out}(c, \mathsf{senc}(\mathsf{diff}[s, r], x_k))$$

The obtained process satisfies observational equivalence with private names $\{s\}$ when the adversary cannot distinguish whether the output message contains the encryption of the secret s or of a fresh name r.

ProVerif finds an attack against diff-equivalence, both for the original version of the protocol (as expected given the attack of Example 3.2), and for the fixed version. Indeed, the fixed version is also subject to an attack against this equivalence: if the message from A to B is replayed, the first component sends the same encrypted message $\mathsf{senc}(s, k)$ twice, while the second component sends two distinct messages $\mathsf{senc}(r_1, k)$ and $\mathsf{senc}(r_2, k)$, with two distinct values of r.

A possible fix against this attack is to use a probabilistic symmetric encryption scheme, defined with a constructor $\mathsf{senc}(\mathsf{bitstring}, \mathsf{pkey}, \mathsf{rand})$, such that the encryption of message M under the key k is represented by

$\mathsf{senc}(M, k, coins)$ where *coins* is a fresh name representing fresh random coins. Decryption is defined by the rewrite rule

$$\mathsf{sdec}(\mathsf{senc}(x, y, z), y) \to x\,.$$

The fixed protocol is obtained by replacing the final output of the process P_B of Example 2.5 with

$$\mathsf{new}\ r : \mathsf{bitstring}; \mathsf{new}\ coins : \mathsf{rand}; \mathsf{out}(c, \mathsf{senc}(\mathsf{diff}[s, r], x_k, coins))$$

The ciphertexts are then different even when they encrypt the same plaintext under the same key, and ProVerif proves that the obtained process satisfies observational equivalence with private names $\{s\}$.

Baudet, 2007 shows the decidability of diff-equivalence for processes without replication, in a framework similar to ours. The protocol verifiers Maude-NPA (Santiago *et al.*, 2014) and Tamarin (Basin *et al.*, 2015) can also prove diff-equivalence.

An important application of diff-equivalence is the study of protocols that rely on weak secrets, such as passwords. These protocols are subject to guessing attacks, in which the adversary guesses the password (for instance by trying all words of a dictionary), and verifies that it has correctly guessed. This verification may be performed either on-line, by interacting with the protocol participants, which can simply be prevented by limiting the number of allowed attempts, or off-line, by computing on intercepted messages without interaction with the other participants.

Off-line guessing attacks can be modeled by combining observational equivalence, phases (§2.5.4), and primitives defined by equations (§2.5.1), since it is often necessary that the failure of decryption cannot be detected in order to protect oneself against such attacks.

In phase 0, the adversary interacts with the protocol, but the weak secret w is considered as unguessable. In phase 1, the adversary guesses a value of the weak secret, and tries to determine whether its guess is correct or not. The absence of off-line guessing attacks is characterized by an equivalence: the adversary cannot distinguish the weak secret w used in phase 0 from a fresh value w', so it cannot determine whether its guess is correct or not.

Definition 3.10. Let P_0 be a closed process without phase prefix and $\mathcal{N}_{\text{priv}}$ a set of names. We say that P_0, with private names $\mathcal{N}_{\text{priv}}$, prevents off-line guessing attacks against w if $\mathsf{new}\ w; (\mathsf{phase}\ 0; P_0 \mid \mathsf{phase}\ 1; \mathsf{new}\ w'; \mathsf{out}(c, \mathsf{diff}[w, w']))$ satisfies observational equivalence with private names $\mathcal{N}_{\text{priv}}$.

ProVerif can prove that the protocols EKE (Bellovin and Merritt, 1992) and Augmented EKE (Bellovin and Merritt, 1993) satisfy this property. This definition is in the line of work by Lowe, 2002; Cohen, 2002; Corin *et al.*, 2003; Corin *et al.*, 2004; Delaune and Jacquemard, 2004; Drielsma *et al.*, 2005. Lowe, 2002 uses the model checker FDR to handle a bounded number of sessions. Delaune and Jacquemard, 2004 give a decision procedure in this case. Corin *et al.*, 2004 give a definition based on equivalence like ours but do not consider the first active phase and analyze a single session.

In order to prove observational equivalence, ProVerif requires that the considered processes differ only by their terms. As suggested by Cheval and Blanchet, 2013, this limitation can be partly overcome by encoding as many constructs as possible into terms. Conditionals can be encoded into terms using the destructor ifthenelse defined in §2.1, as shown in Example 3.7. Expression evaluations such as $\mathsf{let}\ x = D\ \mathsf{in}\ \mathsf{out}(c, M_1)\ \mathsf{else}\ \mathsf{out}(c, M_2)$ can be transformed into $\mathsf{let}\ x = \mathsf{catch\text{-}fail}(D)\ \mathsf{in}\ \mathsf{let}\ m = \mathsf{ifthenelse}(\mathsf{not\text{-}caught\text{-}fail}(x), M_1, M_2)\ \mathsf{in}\ \mathsf{out}(c, m)$, where the destructors catch-fail and not-caught-fail are defined by the rewrite rules

$$\mathsf{catch\text{-}fail}(x) \to x \qquad \mathsf{not\text{-}caught\text{-}fail}(\mathsf{caught\text{-}fail}) \to \mathsf{false}$$
$$\mathsf{catch\text{-}fail}(\mathsf{fail}) \to \mathsf{caught\text{-}fail} \qquad \mathsf{not\text{-}caught\text{-}fail}(x) \to \mathsf{true}$$

and caught-fail is a new constant. After transformation, if D succeeds, then x has the same value as before, and $x \neq \mathsf{caught\text{-}fail}$, so $\mathsf{not\text{-}caught\text{-}fail}(x) = \mathsf{true}$ and $\mathsf{ifthenelse}(\mathsf{not\text{-}caught\text{-}fail}(x), M_1, M_2)$ returns M_1; if D fails, then $x = \mathsf{caught\text{-}fail}$, $\mathsf{not\text{-}caught\text{-}fail}(x) = \mathsf{false}$, and $\mathsf{ifthenelse}(\mathsf{not\text{-}caught\text{-}fail}(x), M_1, M_2)$ returns M_2.

Example 3.7. As in (Cheval and Blanchet, 2013), we consider a simplified version of the private authentication protocol by Abadi and Fournet, 2004. In this protocol, a participant A is willing to reveal its identity

to a participant B, without revealing it to other participants. Using the pattern-matching extension of §2.5.3, the roles of A and B may be written in ProVerif as follows:

$$
\begin{array}{lcl}
A(sk_A, sk_B) & = & \mathsf{new}\ a : \mathsf{rand}; \mathsf{out}(c, \mathsf{aenc}((a, \mathsf{pk}(sk_A)), \mathsf{pk}(sk_B))); \\
 & & \mathsf{in}(c, x : \mathsf{bitstring}) \\
B(sk_B, sk_A) & = & \mathsf{in}(c, y : \mathsf{bitstring}); \mathsf{new}\ b : \mathsf{rand}; \\
 & & \mathsf{let}\ (xa : \mathsf{rand}, xpk_A : \mathsf{pkey}) = \mathsf{adec}(y, sk_B)\ \mathsf{in} \\
(*) & & \quad \mathsf{if}\ xpk_A = \mathsf{pk}(sk_A)\ \mathsf{then} \\
 & & \quad\quad \mathsf{out}(c, \mathsf{aenc}((xa, b, \mathsf{pk}(sk_B)), \mathsf{pk}(sk_A))) \\
 & & \quad \mathsf{else}\ \mathsf{out}(c, \mathsf{aenc}(b, \mathsf{pk}(sk_B))) \\
 & & \mathsf{else}\ \mathsf{out}(c, \mathsf{aenc}(b, \mathsf{pk}(sk_B)))
\end{array}
$$

where sk_A and sk_B are the private keys of A and B, respectively. A first sends to B a nonce a and its own public key $\mathsf{pk}(sk_A)$ encrypted with the public key of B, $\mathsf{pk}(sk_B)$. After receiving this message, B checks that the message is of the correct form and that it contains the public key of A. If so, B replies with the correct message composed of the nonce a he received, a freshly generated nonce b, and his own public key $(\mathsf{pk}(sk_B))$, all this encrypted with the public key of A. Otherwise, B replies with an error message, $\mathsf{aenc}(b, \mathsf{pk}(sk_B))$. From the point of view of the adversary, this error message is indistinguishable from the correct one since the private keys sk_A and sk_B are unknown to the adversary, so the adversary should not be able to tell whether B is willing to talk to A or to another participant. Formally, this anonymity property holds when the process

$$
\begin{array}{l}
\mathsf{new}\ sk_A : \mathsf{skey}; \mathsf{new}\ sk'_A : \mathsf{skey}; \mathsf{new}\ sk_B : \mathsf{skey}; \\
\mathsf{out}(c, (\mathsf{pk}(sk_A), \mathsf{pk}(sk'_A), \mathsf{pk}(sk_B))); B(sk_B, \mathsf{diff}[sk_A, sk'_A])
\end{array}
$$

satisfies observational equivalence with no private names. This process generates private keys, publishes the corresponding public keys, and runs B talking to A in the first component and to A' in the second one. (A and A' need not be explicitly modeled since the role of A uses only public keys, so it can be included in the adversary.)

ProVerif cannot prove this equivalence, because this process does not satisfy diff-equivalence. Suppose the adversary sends $\mathsf{aenc}((a, \mathsf{pk}(sk_A)),$

$\mathsf{pk}(sk_B))$ to B. Then $xpk_A = \mathsf{pk}(sk_A)$, so the then branch of the test $(*)$ is taken in $B(sk_B, sk_A)$, while the else branch is taken in $B(sk_B, sk'_A)$.

The test $(*)$ can be performed in a term, by transforming the process $B(sk_B, sk_A)$ into

$$\begin{aligned} B'(sk_B, sk_A) \quad = \quad & \mathsf{in}(c, y : \mathsf{bitstring}); \mathsf{new}\ b : \mathsf{rand}; \\ & \mathsf{let}\ (xa : \mathsf{rand}, xpk_A : \mathsf{pkey}) = \mathsf{adec}(y, sk_B)\ \mathsf{in} \\ & \quad \mathsf{out}(c, \mathsf{ifthenelse}(\mathsf{equal}(xpk_A, \mathsf{pk}(sk_A)), \\ & \qquad\qquad \mathsf{aenc}((xa, b, \mathsf{pk}(sk_B)), \mathsf{pk}(sk_A)), \\ & \qquad\qquad \mathsf{aenc}(b, \mathsf{pk}(sk_B)))) \\ & \mathsf{else}\ \mathsf{out}(c, \mathsf{aenc}(b, \mathsf{pk}(sk_B))) \end{aligned}$$

ProVerif proves diff-equivalence for the transformed process.

ProVerif includes an automatic process simplification procedure, which transforms conditionals and expression evaluations into terms when possible, as well as a merging procedure, which can merge two processes into a biprocess in order to prove observational equivalence, when the processes are sufficiently similar (Cheval and Blanchet, 2013).

Another transformation useful for proving equivalence consists in swapping data between parallel processes at synchronization points. This transformation is particularly useful for e-voting protocols. Consider a process $V(id_A, v_A)$ that represents a voter id_A voting v_A. The voting protocol satisfies ballot secrecy when the process

$$V(id_A, \mathsf{diff}[v_A, v_B]) \mid V(id_B, \mathsf{diff}[v_B, v_A]) \tag{3.6}$$

satisfies observational equivalence, which means that an adversary cannot distinguish when two voters swap their votes (Delaune *et al.*, 2009). In voting protocols, the voters often synchronize with each other, for instance because the protocol includes several stages, such as registration, voting, and tallying. To prove this equivalence, one often needs to swap the votes between the two voters at a synchronization point. This idea was introduced by Delaune *et al.*, 2008 and developed and formalized by Blanchet and Smyth, 2016. For instance, as in (Blanchet and Smyth, 2016), let us consider a toy protocol in which each voter first sends its identity on an anonymous channel to register, then sends its vote on an anonymous channel:

$$V(id_A, v_A) = \mathsf{out}(c, id_A); \mathsf{out}(c, v_A)\,.$$

This protocol does not satisfy ballot secrecy because $V(id_A, v_A) \mid V(id_B, v_B)$ can output id_A, v_A, id_B, v_B on channel c in that order, while $V(id_A, v_B) \mid V(id_B, v_A)$ cannot. To solve this problem, the voters need to synchronize together after the registration. ProVerif represents the synchronization by the construct $\mathsf{sync}\ n$, so the voting process becomes:

$$V(id_A, v_A) = \mathsf{out}(c, id_A); \mathsf{sync}\ 1; \mathsf{out}(c, v_A)\,. \tag{3.7}$$

The synchronization construct $\mathsf{sync}\ n$ is similar to $\mathsf{phase}\ n$ except that it never drops processes that have not reached the synchronization yet. The number of processes that must synchronize is fixed from the beginning of the execution, and $\mathsf{sync}\ n$ blocks until that number of processes reach the synchronization $\mathsf{sync}\ n$. For instance, consider the biprocess (3.6) where V is defined by (3.7), that is,

$$\begin{aligned} P_0 = {} & \mathsf{out}(c, id_A); \mathsf{sync}\ 1; \mathsf{out}(c, \mathsf{diff}[v_A, v_B]) \\ & \mid \mathsf{out}(c, id_B); \mathsf{sync}\ 1; \mathsf{out}(c, \mathsf{diff}[v_B, v_A])\,. \end{aligned} \tag{3.8}$$

The process P_0 contains two occurrences of $\mathsf{sync}\ 1$, so $\mathsf{sync}\ 1$ blocks until two processes reach $\mathsf{sync}\ 1$. We refer the reader to (Blanchet and Smyth, 2016) for a formal semantics of synchronization.

The protocol P_0 satisfies ballot secrecy, but this property cannot be shown directly using diff-equivalence, because P_0 does not satisfy diff-equivalence with no private names. Intuitively, diff-equivalence requires that the subprocesses of the parallel composition, namely, $\mathsf{out}(c, id_A); \mathsf{sync}\ 1; \mathsf{out}(c, \mathsf{diff}[v_A, v_B])$ and $\mathsf{out}(c, id_B); \mathsf{sync}\ 1; \mathsf{out}(c, \mathsf{diff}[v_B, v_A])$, each satisfy diff-equivalence, which is false, because $\mathsf{out}(c, v_A)$ is not equivalent to $\mathsf{out}(c, v_B)$. (Using the definition of diff-equivalence, we let $\mathcal{N}_{\text{priv}} = \emptyset$, $\mathcal{N}_{\text{pub}} = \{c, id_A, id_B, v_A, v_B\}$, P_0 as in (3.8), and $Q = \mathsf{in}(c, x); \mathsf{in}(c, y); \mathsf{in}(c, x'); \mathsf{in}(c, y'); \mathsf{let}\ x'' = \mathsf{istrue}(x' = v_A)\ \mathsf{in}\ \mathsf{out}(c, v_A)$. The configuration $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \{P_0, Q\}$ reduces into $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \{\mathsf{let}\ x'' = \mathsf{istrue}(\mathsf{diff}[v_A, v_B] = v_A)\ \mathsf{in}\ \mathsf{out}(c, v_A)\}$ by receiving id_A in x and id_B in y, synchronizing, receiving $\mathsf{diff}[v_A, v_B]$ in x' and $\mathsf{diff}[v_B, v_A]$ in y', and removing $\mathbf{0}$ processes. The evaluation of $\mathsf{istrue}(\mathsf{diff}[v_A, v_B] = v_A)$ succeeds in the first component and fails in the second one, hence the obtained configuration diverges and the biprocess P_0 does not satisfy diff-equivalence.) To prove the equivalence, we need to swap the votes

at the synchronization point in the second component of the biprocess, so that it becomes:

$$\begin{aligned}&\mathsf{out}(c, id_A); \mathsf{sync}\ 1; \mathsf{out}(c, \mathsf{diff}[v_A, v_A])\\ |\ &\mathsf{out}(c, id_B); \mathsf{sync}\ 1; \mathsf{out}(c, \mathsf{diff}[v_B, v_B])\end{aligned}$$

which obviously satisfies diff-equivalence. Blanchet and Smyth, 2016 have designed, proved correct, and implemented a compiler that translates the synchronization construct into inputs and outputs and allows this swapping.

3.5 Usage heuristics

Several heuristics can help obtaining the best performance of ProVerif. Technical details on some of these heuristics can be found in (Blanchet *et al.*, 2016, §6.3.1).

As mentioned in §2.5.1, using destructors when possible yields better performance than equations.

The precision and cost of the analysis can also be tuned by adjusting the arguments of patterns that represent names. By moving restrictions downwards in the syntax tree as far as possible, more inputs occur above restrictions, so the representation of the corresponding fresh names has more arguments. This transformation increases the precision and the cost of the analysis. Conversely, by moving restrictions upwards, the representation of the corresponding fresh names has fewer arguments, and the analysis is faster and less precise. ProVerif also allows the user to annotate restrictions with the variables that should occur in the internal representation of fresh names. (The session identifiers are always present. They are sufficient for soundness.) For the proof of equivalences, false attacks are sometimes due to names that have different arguments in the two components of the biprocess. In this case, making sure that matching names have the same arguments helps avoiding the false attack. This can be done by annotating the restrictions that create these names with the desired arguments.

When ProVerif does not terminate, tuning the selection function of the resolution algorithm may help. In particular, when ProVerif generates an unbounded number of clauses in which the selected fact is

an instance of a fact F, one can tell ProVerif to avoid selecting a fact that matches F, by the declaration nounif F. ProVerif tries to detect some of these cases automatically, and to adjust the selection function accordingly, but sometimes manual declarations are still useful.

To share the work as much as possible, one can group several secrecy and correspondence queries together, in a single query declaration. ProVerif then generates a single set of Horn clauses for these queries, and saturates this set once. This is particularly beneficial when these queries use the same events. However, when queries use different events, it is often better to treat them separately, because the more events appear in a query, the more complex the generated clauses are, which can slow down ProVerif considerably. (Events that do not occur in the query are ignored when ProVerif generates the clauses.)

Finally, for the proof of equivalences, grouping all tests performed on a message in a single test can help avoiding false attacks. As a toy example, consider the biprocess

$$\mathsf{let}\ x = \mathsf{diff}[\mathsf{aenc}(\mathsf{sign}(a, sk_A), \mathsf{pk}(sk_B)), \mathsf{aenc}(a, \mathsf{pk}(sk_A))]\ \mathsf{in}$$
$$\mathsf{let}\ y = \mathsf{adec}(x, sk_A)\ \mathsf{in}\ \mathsf{let}\ z = \mathsf{check}(y, \mathsf{pk}(sk_B))\ \mathsf{in}\ \mathsf{out}(c, z)$$

This biprocess does not satisfy diff-equivalence with private names $\{a, sk_A, sk_B\}$, because the decryption $\mathsf{adec}(x, sk_A)$ fails in the first component (the keys do not match), while it succeeds in the second component. However, this biprocess satisfies observational equivalence, because the signature verification $\mathsf{check}(y, \mathsf{pk}(sk_B))$ fails in the second component, so no component executes the output. ProVerif can prove the equivalence after grouping the decryption and the signature verification in one step:

$$\mathsf{let}\ x = \mathsf{diff}[\mathsf{aenc}(\mathsf{sign}(a, sk_A), \mathsf{pk}(sk_B)), \mathsf{aenc}(a, \mathsf{pk}(sk_A))]\ \mathsf{in}$$
$$\mathsf{let}\ z = \mathsf{check}(\mathsf{adec}(x, sk_A), \mathsf{pk}(sk_B))\ \mathsf{in}\ \mathsf{out}(c, z)$$

Indeed, this biprocess satisfies diff-equivalence because the combined decryption and signature verification fails in both components. In case a proof of equivalence fails, ProVerif performs an automatic process simplification that groups the tests (Cheval and Blanchet, 2013). However, grouping them manually in the initial process avoids the first failed proof attempt and yields better performance.

4

Link with the Applied Pi Calculus

The input language of ProVerif is a dialect of the applied pi calculus introduced by Abadi and Fournet, 2001, and later updated by Abadi *et al.*, 2016. In this chapter, we use the latter version. The applied pi calculus is an extension of the pi calculus with function symbols defined by an equational theory. It can be understood as a minimal core of the input language of ProVerif. This chapter formally relates the two languages.

The syntax of terms M of the applied pi calculus is the same as in ProVerif (Figure 2.1). The syntax of processes P of the applied pi calculus is given in Figure 4.1. Although the precise syntax of restrictions, inputs, and outputs differs, this calculus is very similar to the input language of ProVerif. More precisely, it matches a subset of the ProVerif language with equations (§2.5.1) and enriched terms (§2.5.2), with the following additional conditions: the only destructor is equal defined by (2.5), $\mathsf{equal}(M, N)$ is written $M = N$, the expression in the conditional is always of the form $M = N$, and the destructor equal does not occur elsewhere; all function symbols are public; the expression evaluation construct is removed. The correspondence is formalized by the following encoding function, from ProVerif processes to applied pi

$P, Q, R ::=$	processes (or plain processes)
$\mathbf{0}$	null process
$P \mid Q$	parallel composition
$!P$	replication
$\nu n.P$	name restriction ("new")
$\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
$N(x).P$	message input
$\overline{N}\langle M\rangle.P$	message output

Figure 4.1: Syntax of the applied pi calculus

processes:

$$[\![\mathbf{0}]\!] = \mathbf{0}$$
$$[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$$
$$[\![!P]\!] = ![\![P]\!]$$
$$[\![\mathsf{new}\ n : T; P]\!] = \nu n.[\![P]\!]$$
$$[\![\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] = \mathsf{if}\ M = N\ \mathsf{then}\ [\![P]\!]\ \mathsf{else}\ [\![Q]\!]$$
$$[\![\mathsf{in}(N, x : T); P]\!] = N(x).[\![P]\!]$$
$$[\![\mathsf{out}(N, M); P]\!] = \overline{N}\langle M\rangle.[\![P]\!]$$

The applied pi calculus has a notion of extended processes, which additionally contain the current knowledge of the adversary, represented by substitutions. Extended processes are useful for proving observational equivalence manually, via the notion of labeled bisimilarity (Abadi *et al.*, 2016), but they are not necessary for relating the calculus to ProVerif, so we omit them here. We refer to processes as *plain processes* when we want to stress that they are not extended processes.

The applied pi calculus has a sort system, which is similar to the type system of ProVerif. However, in the applied pi calculus, each variable and name comes with its own sort, so processes are not annotated with sorts. We write $\vdash M : T$ when M is a term of sort T and $\vdash P$ when the process P is well-sorted in the applied pi calculus. If the type of each function symbol in ProVerif is the same as its sort in the applied pi calculus and the type given to each variable and name in the type

environment Γ and in the process P matches its sort in the applied pi calculus, then $\Gamma \vdash M : T$ if and only if $\vdash M : T$, and $\Gamma \vdash P$ if and only if $\vdash [\![P]\!]$. The untyped version of the applied pi calculus can be obtained by giving all terms the sort channel. In this chapter, we assume that either this untyped version of the applied pi calculus is used, or ProVerif is configured to take types into account in the verification of security properties.

In contrast to the semantics of ProVerif, the operational semantics of the applied pi calculus is defined by a reduction relation $\rightarrow_\diamond$ on processes. To prepare processes for reduction, we use a structural equivalence relation $\overset{\diamond}{\equiv}$. Formally, the semantics for plain processes is defined as follows (Abadi *et al.*, 2016, §B.2). As usual, a context is a process with a hole. An *evaluation context* is a context whose hole is not under a replication, a conditional, an input, or an output. Structural equivalence $\overset{\diamond}{\equiv}$ is the smallest equivalence relation on processes closed by application of evaluation contexts such that

Par-**0**	$P \mid \mathbf{0}$	$\overset{\diamond}{\equiv}$	P	
Par-A	$P \mid (Q \mid R)$	$\overset{\diamond}{\equiv}$	$(P \mid Q) \mid R$	
Par-C	$P \mid Q$	$\overset{\diamond}{\equiv}$	$Q \mid P$	
Repl	$!P$	$\overset{\diamond}{\equiv}$	$P \mid !P$	
New-**0**	$\nu n.\mathbf{0}$	$\overset{\diamond}{\equiv}$	$\mathbf{0}$	
New-C	$\nu n.\nu n'.P$	$\overset{\diamond}{\equiv}$	$\nu n'.\nu n.P$	
New-Par	$P \mid \nu n.Q$	$\overset{\diamond}{\equiv}$	$\nu n.(P \mid Q)$	when $n \notin fn(P)$
Rewrite	$P\{^M/_x\}$	$\overset{\diamond}{\equiv}$	$P\{^N/_x\}$	when $M =_\mathcal{E} N$

Rules Par-A, Par-C, and Par-**0** express that parallel composition is associative, commutative, and has **0** as neutral element. Rule Repl allows one to create new copies of replicated processes (or to fold them back by applying Repl from right to left). The scope extrusion rule New-Par allows one to enlarge the scope of a restriction, for instance to make processes P and Q communicate. Rule Rewrite allows one to rewrite terms modulo the equational theory. The reduction relation $\rightarrow_\diamond$ is the smallest relation on processes closed by $\overset{\diamond}{\equiv}$ and by

application of evaluation contexts such that:

$$\begin{array}{lll} \textsc{Comm} & \overline{N}\langle M\rangle.P \mid N(x).Q & \to_\diamond \quad P \mid Q\{^M/_x\} \\ \textsc{Then} & \mathsf{if}\ M = M\ \mathsf{then}\ P\ \mathsf{else}\ Q & \to_\diamond \quad P \\ \textsc{Else} & \mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q & \to_\diamond \quad Q \end{array}$$

for any ground terms M and N such that $M \neq_{\mathcal{E}} N$

Rule Comm performs communication between two processes. Rules Then and Else define the semantics of conditionals. This semantics is superficially very different from the semantics of ProVerif. However, we can still relate the two semantics formally. The encoding can be extended to configurations as follows:

$$[\![(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}]\!] = \nu\widetilde{a}.([\![P_1]\!] \mid \cdots \mid [\![P_n]\!])$$

where $\mathcal{N}_{\text{priv}} = \{\widetilde{a}\}$ and $\mathcal{P} = \{P_1, \ldots, P_n\}$. In particular, $[\![(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}]\!] = \nu\widetilde{a}.\mathbf{0}$ when $\mathcal{P} = \emptyset$. The encoding of configurations is unique modulo $\overset{\diamond}{\equiv}$. More formally, $[\![(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}]\!]$ is a set of processes, containing all processes of the form $\nu\widetilde{a}.([\![P_1]\!] \mid \cdots \mid [\![P_n]\!])$ with any ordering of $\widetilde{a}$ and any parallel composition of $[\![P_1]\!]$, …, $[\![P_n]\!]$ modulo associativity and commutativity. The results below hold for any element of $[\![\mathcal{C}]\!]$.

The next proposition formally relates the two semantics. It is proved in Appendix B, together with the other results of this chapter.

Proposition 4.1. If $\mathcal{C} \to^* \mathcal{C}'$, then $[\![\mathcal{C}]\!] \to_\diamond{}^* \overset{\diamond}{\equiv} [\![\mathcal{C}']\!]$.

Conversely, if $[\![\mathcal{C}]\!] \to_\diamond{}^* P'$, then $\mathcal{C} \to^* \mathcal{C}'$ and $P' \overset{\diamond}{\equiv} [\![\mathcal{C}']\!]$ for some $\mathcal{C}'$.

Next, we define observational equivalence in the applied pi calculus. We write $P \Downarrow_a^\diamond$ when P can send a message on name a, that is, when $P \to_\diamond{}^* \overset{\diamond}{\equiv} C[\overline{a}\langle M\rangle.Q]$ for some evaluation context $C[\_]$ that does not bind a.

Definition 4.1. An *observational bisimulation* is a symmetric relation $\mathcal{R}$ between closed processes such that $P\ \mathcal{R}\ Q$ implies:

1. if $P \Downarrow_a^\diamond$, then $Q \Downarrow_a^\diamond$;
2. if $P \to_\diamond{}^* P'$ and P' is closed, then $Q \to_\diamond{}^* Q'$ and $P'\ \mathcal{R}\ Q'$ for some Q';

3. $C[P]\ \mathcal{R}\ C[Q]$ for all closed evaluation contexts $C[\_]$.

Observational equivalence ($\overset{\diamond}{\approx}$) is the largest such relation.

As shown in §B.3, this definition is equivalent to (Abadi *et al.*, 2016, Definition 4.1) for plain processes.

On the ProVerif side, the definition of $\mathcal{C} \downarrow_N$ needs to be adapted in the presence of equations, because the channel N is modulo the equational theory. We have $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P} \downarrow_N$ if and only if there exists $\mathsf{out}(N', M); P \in \mathcal{P}$ with $N =_{\mathcal{E}} N'$ and $fn(N) \subseteq \mathcal{N}_{\text{pub}}$, for some terms N', M and process P. We can then relate the equivalence used by ProVerif (Definition 3.6) and the equivalence of the applied pi calculus (Definition 4.1).

Proposition 4.2. Let $\mathcal{C}$ and $\mathcal{C}'$ be valid configurations. If $\mathcal{C} \approx \mathcal{C}'$, then $[\![\mathcal{C}]\!] \overset{\diamond}{\approx} [\![\mathcal{C}']\!]$.

This proposition shows that, if ProVerif proves an equivalence between processes in the language subset corresponding to the applied pi calculus, then the corresponding applied pi calculus processes are observationally equivalent.

The converse is less important in practice, since ProVerif proves diff-equivalence, which is stronger than $\approx$. Its proof requires encoding a ProVerif context into the applied pi calculus. In the proof in appendix, we provide this encoding for the core ProVerif language with equations and enriched terms. Pattern-matching and tables can be defined as an encoding in that subset of the ProVerif calculus, so the result extends to them. Phases would be more difficult to encode, though intuitively they do not give more power to the adversary when the considered protocol does not use them. For simplicity, the next proposition considers a ProVerif language without phases.

Proposition 4.3. Let $\mathcal{C}$ and $\mathcal{C}'$ be valid configurations. If $[\![\mathcal{C}]\!] \overset{\diamond}{\approx} [\![\mathcal{C}']\!]$, then $\mathcal{C} \approx \mathcal{C}'$.

5

Applications

This chapter outlines a number of applications and extensions of ProVerif.

5.1 Case studies

ProVerif can successfully prove secrecy and authentication properties for many protocols of the literature (Blanchet, 2009): no false attack is found in the 19 protocols tested in (Blanchet, 2009). It can also handle more substantial case studies:

- Abadi and Blanchet, 2005b verify a certified email protocol (Abadi *et al.*, 2002). In this work, correspondence properties are used to prove that the receiver receives the message if and only if the sender has a receipt for the message, with simple manual arguments to take into account that the reception of sent messages is guaranteed. This protocol relies on a secure channel; one of the tested versions includes the SSH transport layer protocol in order to establish this channel.
- Abadi *et al.*, 2007 study the JFK protocol (*Just Fast Keying*) by Aiello *et al.*, 2004, which was one of the candidates to replace

IKE as the key exchange protocol in IPSec. This work combines manual proofs and ProVerif to prove correspondences and equivalences.

- Blanchet and Chaudhuri, 2008 study the secure filesystem Plutus (Kallahalla *et al.*, 2003), discovering and fixing weaknesses of the initial system.

- ProVerif was also used for verifying a certified email web service (Lux *et al.*, 2005), a certified mailing-list protocol (Khurana and Hahm, 2006), e-voting protocols (Kremer and Ryan, 2005; Backes *et al.*, 2008a; Delaune *et al.*, 2009; Cortier and Wiedling, 2012), e-auction protocols (Dreier *et al.*, 2013), the ad-hoc routing protocol ARAN (*Authenticated Routing for Adhoc Networks*) (Godskesen, 2006), zero-knowledge protocols (Backes *et al.*, 2008b; Backes *et al.*, 2015), the Trusted Platform Module (TPM) (Delaune *et al.*, 2011), and the associated Direct Anonymous Attestation protocol (Backes *et al.*, 2008b; Smyth *et al.*, 2015), for instance.

ProVerif can also be used for verifying protocols in the computational model, via computational soundness results:

- Canetti and Herzog, 2006 show that, for a restricted class of protocols that use only public-key encryption, a proof in the Dolev-Yao model implies security in the computational model, in the universal composability framework.

- Backes *et al.*, 2014 prove computational soundness for the equivalence properties shown by ProVerif. To achieve this result, they take advantage that ProVerif relies on a trace property on biprocesses in order to prove equivalence, so they can leverage computational soundness results for trace properties.

5.2 Extensions

Other authors also design and implement extensions of ProVerif:

- Küsters and Truderung, 2008; Küsters and Truderung, 2009 extend ProVerif with support for exclusive or, and improve support for Diffie-Hellman key agreements. In the same line, Pankova and Laud, 2012 extend ProVerif with support for bilinear pairings (see also §2.5.1).

- StatVerif (Arapinis *et al.*, 2011) is an extension of ProVerif with support for mutable state. It allows global mutable memory cells, which are encoded into Horn clauses by adding their current contents to the arguments of the predicates attacker and message.

- Mödersheim, 2010; Bruni *et al.*, 2015 provide another extension of ProVerif with support for state. They allow sets (such as databases of keys or access rights) where revocation is possible, so that the set of true facts does not increase monotonically. To achieve this result, they rely on a new abstraction of sets into terms in Horn clauses. This approach was first proposed at the Horn clause level (Mödersheim, 2010), then extended by providing a translation from an extension of the pi calculus (Bruni *et al.*, 2015).

- Chothia *et al.*, 2015 provide an extension that avoids false attacks in protocols that first commit to a value, then later reveal it. This extension relies on the insertion of phase instructions, to avoid that ProVerif considers the revealed value as available to the adversary from the beginning of the protocol.

5.3 ProVerif as back-end

ProVerif is also used as a back-end for building other verification tools:

- Bhargavan *et al.*, 2004 use it to build the Web services verification tool TulaFale: Web services are protocols that send XML messages; TulaFale translates them into the input format of ProVerif and uses ProVerif to prove the desired security properties.

- Bhargavan *et al.*, 2006 use ProVerif for verifying implementations of protocols in F# (a functional language of the Microsoft .NET

environment): a subset of F# large enough for expressing security protocols is translated into the input format of ProVerif. The TLS protocol (Bhargavan *et al.*, 2008) and the TPM (Mukhamedov *et al.*, 2013), in particular, were studied using this technique.

- The JavaSPI framework (Avalle *et al.*, 2011) allows modeling security protocols in a subset of Java with annotations. These specifications can be verified with ProVerif via a Java-to-ProVerif converter and compiled into Java protocol implementations.

- Aizatulin *et al.*, 2011 use symbolic execution in order to extract ProVerif models from pre-existing protocol implementations in C. This technique currently analyzes a single execution path of the protocol, so it is limited to protocols without branching. An earlier related approach is that of Goubault-Larrecq and Parrennes, 2005: they also use the Horn clause method for analyzing implementations of protocols written in C. However, they translate protocols into clauses of the $\mathcal{H}_1$ class and use the $\mathcal{H}_1$ prover by Goubault-Larrecq, 2005 rather than ProVerif to prove secrecy properties of the protocol.

- Bansal *et al.*, 2012; Bansal *et al.*, 2013 build the Web-spi library, which allows them to model web security mechanisms and protocols and verify them using ProVerif.

6

Conclusion

The protocol verifier ProVerif takes as input a description of the protocol to verify in an extension of the pi calculus with cryptography, as well as the security properties to prove. It may return three different answers: the property is true, the property is false with an attack (execution trace of the protocol), or "I do not know" with a derivation at the Horn clause level which does not correspond to a trace of the protocol.

Its main strengths are: it is fully automatic; it supports an unbounded number of sessions of the protocol; it allows the specification of many different cryptographic primitives by rewrite rules or by equations; and it can prove a wide variety of security properties, namely, secrecy, correspondences, and some equivalences.

It still has limitations: it may not terminate and it makes abstractions, which may lead to false attacks. These limitations are unavoidable because the verification of security protocols with an unbounded number of sessions is undecidable. In practice, ProVerif is still efficient and precise in many examples. A further limitation is the class of equational theories that it supports; in particular, it does not support associativity. The class of equivalences that it can prove is also limited. Some extensions, outlined in §5.2, tackle these limitations, and other verifiers make

different trade-offs. For instance, Maude-NPA (Meadows, 1996; Escobar *et al.*, 2006) and Tamarin (Schmidt *et al.*, 2012) support more equational theories, at the cost of a more costly verification or by requiring user intervention.

These limitations suggest areas for future work. For instance, we might improve the precision of the analysis to avoid false attacks, by adding more information into the clauses. We might support more equational theories, by implementing unification and resolution modulo associativity and commutativity. We might improve the support for mutable state, possibly by integrating StatVerif (Arapinis *et al.*, 2011) and extending it. The proof of equivalences is also an area in which further research could be done, perhaps by combining manual and automatic proofs. For instance, a proof of observational equivalence could be decomposed into a family of static equivalences (Abadi and Fournet, 2001) (equivalences between messages), proved automatically by ProVerif, and a manual bisimulation proof. On a more technical side, the user interface of ProVerif could be improved, for instance by displaying attacks graphically. The subsumption tests could be parallelized to improve performance on machines with multiple cores.

Finally, ProVerif verifies specifications of protocols in the symbolic model. One may go further in several directions. First, one may verify specifications of protocols in a more concrete model, such as the computational model, used by cryptographers. Several tools exist for verifying cryptographic primitives and protocols in this model, in particular, CertiCrypt (Barthe *et al.*, 2009; Béguelin *et al.*, 2009) and its successor EasyCrypt (Barthe *et al.*, 2011; Barthe *et al.*, 2014b), and CryptoVerif (Blanchet, 2008a). They are generally more delicate to use than symbolic protocol verifiers. We plan to modify ProVerif and CryptoVerif to make their input languages compatible, so that the same input file can be used to verify a protocol in both tools, to find attacks or obtain a symbolic proof in ProVerif, or obtain a computational proof in CryptoVerif. Second, instead of verifying specifications, one may verify runnable implementations of protocols. This verification is important because new attacks may appear in the implementation. Some tools for verifying implementations rely on ProVerif as a back-end (see §5.3).

There exist other approaches, by extracting models from implementations for other symbolic (O'Shea, 2008) or computational (Bhargavan *et al.*, 2007; Aizatulin *et al.*, 2012) verifiers, by generating implementations from models by compilation (Song *et al.*, 2001; Milicia, 2002; Pozza *et al.*, 2004; Pironti and Sisto, 2010; Almeida *et al.*, 2013; Cadé and Blanchet, 2013; Cadé and Blanchet, 2015), or by direct verification of the implementation, for instance by typing (Bengtson *et al.*, 2011; Bhargavan *et al.*, 2010; Swamy *et al.*, 2011; Fournet and Kohlweiss, 2011; Bhargavan *et al.*, 2013), by model-checking (Chaki and Datta, 2009), or by the general-purpose C verifier VCC (Dupressoir *et al.*, 2011). Going even further, one may wish to take into account side-channels attacks, that is, physical attacks, which may rely for instance on timing, power consumption, or fault injection, and are not considered in the computational model. Taking such attacks into account in mechanized verification tools is still at its beginning (Barthe *et al.*, 2014a).

Acknowledgments

This survey borrows material from previous surveys (Blanchet, 2012b; Blanchet, 2014) and from my habilitation thesis (Blanchet, 2008b).

Much research on ProVerif has been done with co-authors: in alphabetical order, Martín Abadi, Xavier Allamigeon, Vincent Cheval, Cédric Fournet, Andreas Podelski, and Ben Smyth. I would like to thank them for their contributions. I also thank Ben Smyth for helpful comments on a draft of this survey.

Appendices

A

Proof of Theorem 3.5

Theorem 3.5. Let P_0 be a closed biprocess and $\mathcal{N}_{\text{priv}}$ be a set of names. If P_0 satisfies diff-equivalence with private names $\mathcal{N}_{\text{priv}}$, then P_0 satisfies observational equivalence with private names $\mathcal{N}_{\text{priv}}$.

Proof. Let us define the relation $\mathcal{R}$ by $\mathcal{C}\ \mathcal{R}\ \mathcal{C}'$ if and only if there exists a configuration $\mathcal{C}''$ such that $\mathcal{C} = \mathsf{fst}(\mathcal{C}'')$, $\mathcal{C}' = \mathsf{snd}(\mathcal{C}'')$, and for all adversarial contexts $C_1[\_], \ldots, C_k[\_]$, there exists no $\mathcal{C}_{\text{bad}}$ such that $C_1[\ldots C_k[\mathcal{C}'']] \to^* \mathcal{C}_{\text{bad}} \uparrow$. The relation $\mathcal{R}$ is symmetric and satisfies the three conditions of Definition 3.6:

1. Suppose $\mathcal{C} = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P} \downarrow_a$ and $\mathcal{C}\ \mathcal{R}\ \mathcal{C}'$. For some term M and process P, we have $\mathsf{out}(a, M); P \in \mathcal{P}$ with $a \in \mathcal{N}_{\text{pub}}$. Moreover, there exists a configuration $\mathcal{C}''$ such that $\mathcal{C} = \mathsf{fst}(\mathcal{C}'')$, $\mathcal{C}' = \mathsf{snd}(\mathcal{C}'')$, and for all adversarial contexts $C_1[\_], \ldots, C_k[\_]$, there exists no $\mathcal{C}_{\text{bad}}$ such that $C_1[\ldots C_k[\mathcal{C}'']] \to^* \mathcal{C}_{\text{bad}} \uparrow$. Hence $\mathcal{C}'' = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}_0$ with $\mathsf{out}(N_0, M_0); P_0 \in \mathcal{P}_0$ and $\mathsf{fst}(N_0) = a$. Let $C[\_] = \_ \mid \mathsf{in}(a, x); \mathbf{0}$. We have $C[\mathcal{C}''] = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}_0 \cup \{\mathsf{in}(a, x); \mathbf{0}\}$ since $a \in \mathcal{N}_{\text{pub}}$. Since $\mathcal{C}\ \mathcal{R}\ \mathcal{C}'$, $C[\mathcal{C}'']$ does not diverge, so $\mathsf{fst}(N_0) = \mathsf{fst}(a)$ if and only if $\mathsf{snd}(N_0) = \mathsf{snd}(a)$. Therefore, $\mathsf{snd}(N_0) = a$, so $\mathcal{C}' = \mathsf{snd}(\mathcal{C}'') = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathsf{snd}(\mathcal{P}_0)$ with $\mathsf{out}(a, \mathsf{snd}(M_0)); \mathsf{snd}(P_0) \in \mathsf{snd}(\mathcal{P}_0)$, so $\mathcal{C}' \downarrow_a$.

2. Suppose $\mathcal{C} \mathrel{\mathcal{R}} \mathcal{C}'$ and $\mathcal{C} \to \mathcal{C}_1$. Then there exists a configuration $\mathcal{C}''$ such that $\mathcal{C} = \mathsf{fst}(\mathcal{C}'')$, $\mathcal{C}' = \mathsf{snd}(\mathcal{C}'')$, and for all adversarial contexts $C_1[\_], \ldots, C_k[\_]$, there exists no $\mathcal{C}_{\text{bad}}$ such that $C_1[\ldots C_k[\mathcal{C}'']] \to^* \mathcal{C}_{\text{bad}} \uparrow$. Since $\mathcal{C}''$ does not diverge, we have $\mathcal{C}'' \to \mathcal{C}_1''$ and $\mathsf{fst}(\mathcal{C}_1'') = \mathcal{C}_1$ for some $\mathcal{C}_1''$, by cases on the reductions. Then $\mathcal{C}' = \mathsf{snd}(\mathcal{C}'') \to \mathsf{snd}(\mathcal{C}_1'')$. Let $\mathcal{C}_1' = \mathsf{snd}(\mathcal{C}_1'')$. We have $\mathcal{C}' \to \mathcal{C}_1'$ and $\mathcal{C}_1 \mathrel{\mathcal{R}} \mathcal{C}_1'$ because $\mathcal{C}_1 = \mathsf{fst}(\mathcal{C}_1'')$, $\mathcal{C}_1' = \mathsf{snd}(\mathcal{C}_1'')$, and for all adversarial contexts $C_1[\_], \ldots, C_k[\_]$, there exists no $\mathcal{C}_{\text{bad}}$ such that $C_1[\ldots C_k[\mathcal{C}_1'']] \to^* \mathcal{C}_{\text{bad}} \uparrow$, since that would imply $C_1[\ldots C_k[\mathcal{C}'']] \to C_1[\ldots C_k[\mathcal{C}_1'']] \to^* \mathcal{C}_{\text{bad}} \uparrow$.

3. Let $C[\_]$ be an adversarial context. Suppose $\mathcal{C} \mathrel{\mathcal{R}} \mathcal{C}'$. There exists a configuration $\mathcal{C}''$ such that $\mathcal{C} = \mathsf{fst}(\mathcal{C}'')$, $\mathcal{C}' = \mathsf{snd}(\mathcal{C}'')$, and for all adversarial contexts $C_1[\_], \ldots, C_k[\_]$, there exists no $\mathcal{C}_{\text{bad}}$ such that $C_1[\ldots C_k[\mathcal{C}'']] \to^* \mathcal{C}_{\text{bad}} \uparrow$. Then $C[\mathcal{C}] = \mathsf{fst}(C[\mathcal{C}''])$, $C[\mathcal{C}'] = \mathsf{snd}(C[\mathcal{C}''])$, and for all adversarial contexts $C_1[\_], \ldots, C_k[\_]$, there exists no $\mathcal{C}_{\text{bad}}$ such that $C_1[\ldots C_k[C[\mathcal{C}'']]] \to^* \mathcal{C}_{\text{bad}} \uparrow$, so $C[\mathcal{C}] \mathrel{\mathcal{R}} C[\mathcal{C}']$, using $C[\mathcal{C}'']$ instead of $\mathcal{C}''$.

Therefore, $\mathcal{R} \subseteq \approx$. Let $\mathcal{C}'' = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \{P_0\}$. To show that P_0 satisfies observational equivalence with private names $\mathcal{N}_{\text{priv}}$, it suffices to show that $\mathsf{fst}(\mathcal{C}'') \approx \mathsf{snd}(\mathcal{C}'')$, so it suffices to show that $\mathsf{fst}(\mathcal{C}'') \mathrel{\mathcal{R}} \mathsf{snd}(\mathcal{C}'')$. To prove this property, we suppose that there exist adversarial contexts $C_1[\_], \ldots, C_k[\_]$ such that $C_1[\ldots C_k[\mathcal{C}'']] \to^* \uparrow$, and we derive a contradiction. To reach this contradiction, we build an $\mathcal{N}_{\text{pub}}$-adversary Q from the context $C_1[\ldots C_k[\_]]$, such that $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \{P_0, Q\} \to^* \uparrow$, which contradicts the hypothesis of the theorem.

Let $C_i[\_] = \mathsf{new}\ \widetilde{n}_i; (\_ \mid Q_i)$ for all $i \in \{1, \ldots, k\}$. We have $C_k[\mathcal{C}''] = ((\mathcal{N}_{\text{pub}} \cup \mathit{fn}(Q_k)) \setminus \{\widetilde{n}_k\}, \alpha_k \mathcal{N}_{\text{priv}} \cup \{\widetilde{n}_k\}), \alpha_k \mathcal{P} \cup \{Q_k\}$ where α_k is a bijective renaming of names in $\mathcal{N}_{\text{priv}}$ to names not in $\mathit{fn}(Q_k)$ that leaves names in $\mathcal{N}_{\text{pub}}$ unchanged. Let $\mathcal{N}_{\text{pub}_{k+1}} = \mathcal{N}_{\text{pub}}$ and for all $i \in \{1, \ldots, k\}$, $\mathcal{N}_{\text{pub}_i} = (\mathcal{N}_{\text{pub}_{i+1}} \cup \mathit{fn}(Q_i)) \setminus \{\widetilde{n}_i\}$. So we have

$$\begin{aligned} C_1[\ldots C_k[\mathcal{C}'']] &= (\mathcal{N}_{\text{pub}_1}, \alpha_1(\ldots(\alpha_k \mathcal{N}_{\text{priv}} \cup \{\widetilde{n}_k)\ldots) \cup \widetilde{n}_1), \\ &\qquad \{\alpha_1 \ldots \alpha_k P_0, \alpha_1 \ldots \alpha_{k-1} Q_k, \ldots, Q_1\} \\ &= (\mathcal{N}_{\text{pub}_1}, \alpha_1 \ldots \alpha_k \mathcal{N}_{\text{priv}} \cup \textstyle\bigcup_{i=1}^{k} \alpha_1 \ldots \alpha_{i-1} \widetilde{n}_i), \end{aligned}$$

$$\{\alpha_1 \dots \alpha_k P_0\} \cup \{\alpha_1 \dots \alpha_{i-1} Q_i \mid i \in \{1, \dots, k\}\},$$

where α_i is a bijective renaming of the names in $\alpha_{i+1} \dots \alpha_k \mathcal{N}_{\text{priv}} \cup \bigcup_{j=i+1}^{k} \alpha_{i+1} \dots \alpha_{j-1} \widetilde{n}_j$ to names not in $fn(Q_i)$ that leaves names in $\mathcal{N}_{\text{pub}_{i+1}}$ unchanged. So

$$\begin{aligned}(\alpha_1 \dots \alpha_k)^{-1} C_1[\dots C_k[\mathcal{C}'']] = \\ ((\alpha_1 \dots \alpha_k)^{-1} \mathcal{N}_{\text{pub}_1}, \mathcal{N}_{\text{priv}} \cup \textstyle\bigcup_{i=1}^{k} (\alpha_i \dots \alpha_k)^{-1} \widetilde{n}_i), \\ \{P_0\} \cup \{(\alpha_i \dots \alpha_k)^{-1} Q_i \mid i \in \{1, \dots, k\}\}.\end{aligned}$$

We write $\prod_{i=1}^{k} P_i$ for $P_1 \mid \dots \mid P_k$. Let $Q' = \prod_{i=1}^{k} (\alpha_i \dots \alpha_k)^{-1} Q_i$, $\{\widetilde{n}\} = fn(Q') \setminus \mathcal{N}_{\text{pub}}$ and $Q = \mathsf{new}\ \widetilde{n}; Q'$. The process Q is an $\mathcal{N}_{\text{pub}}$-adversary. Moreover, let us show that $\{\widetilde{n}\} \cap \mathcal{N}_{\text{priv}} = \emptyset$. Suppose that $a \in \{\widetilde{n}\} \cap \mathcal{N}_{\text{priv}}$. Since $\mathcal{N}_{\text{pub}} \cap \mathcal{N}_{\text{priv}} = \emptyset$, for some $i \in \{1, \dots, n\}$, $a \in fn((\alpha_i \dots \alpha_k)^{-1} Q_i) \cap \mathcal{N}_{\text{priv}}$, so $\alpha_i \dots \alpha_k a \in fn(Q_i) \cap \alpha_i \dots \alpha_k \mathcal{N}_{\text{priv}}$. Since $\alpha_i \dots \alpha_k a \in fn(Q_i)$, $\alpha_{i+1} \dots \alpha_k a \notin \alpha_{i+1} \dots \alpha_k \mathcal{N}_{\text{priv}}$, since α_i renames names in $\alpha_{i+1} \dots \alpha_k \mathcal{N}_{\text{priv}}$ to names not in $fn(Q_i)$. Therefore, $\alpha_i \dots \alpha_k a \notin \alpha_i \dots \alpha_k \mathcal{N}_{\text{priv}}$. Contradiction. Hence we have proved $\{\widetilde{n}\} \cap \mathcal{N}_{\text{priv}} = \emptyset$.

Moreover, we have the following property: if $(\mathcal{N}_{\text{pub}}', \mathcal{N}_{\text{priv}}'), \mathcal{P} \rightarrow^* \uparrow$, then $(\mathcal{N}_{\text{pub}}'', \mathcal{N}_{\text{priv}}''), \alpha\mathcal{P} \rightarrow^* \uparrow$, where $\mathcal{N}_{\text{pub}}'' \cap \mathcal{N}_{\text{priv}}'' = \emptyset$, α is a bijective renaming, and $fn(\alpha\mathcal{P}) \subseteq \mathcal{N}_{\text{pub}}'' \cup \mathcal{N}_{\text{priv}}''$. This property is easy to prove by induction on the length of the trace. From this property and $C_1[\dots C_k[\mathcal{C}'']] \rightarrow^* \uparrow$, we derive

$$(\alpha_1 \dots \alpha_k)^{-1} C_1[\dots C_k[\mathcal{C}'']] \rightarrow^* \uparrow,$$

so

$$\begin{aligned}&(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \{P_0, Q\} \rightarrow^* \\ &\quad (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}} \cup \{\widetilde{n}\}), \{P_0\} \cup \{(\alpha_i \dots \alpha_k)^{-1} Q_i \mid i \in \{1, \dots, k\}\} \rightarrow^* \uparrow .\end{aligned}$$

This property contradicts the hypothesis of theorem. This contradiction allows us to conclude the proof. □

B

Proofs for Chapter 4

These proofs use the notations and definitions of (Abadi *et al.*, 2016). In particular, they use the notion of extended processes A (Abadi *et al.*, 2016, §2.1), with associated domain $dom(A)$ (Abadi *et al.*, 2016, §2.1), structural equivalence $\equiv$ and reduction relation $\rightarrow$ (Abadi *et al.*, 2016, §2.2); simple contexts (Abadi *et al.*, 2016, §A); partial normal forms, with associated structural equivalence $\overset{\circ}{\equiv}$ and reduction relation $\rightarrow_\circ$ (Abadi *et al.*, 2016, §B.2); and the labeled semantics $\xrightarrow{\alpha}_\diamond$ of (Abadi *et al.*, 2016, §B.2).

B.1 Proof of Proposition 4.1

Lemma B.1. If $E, \mathcal{P} \rightarrow E', \mathcal{P}'$, then $[\![E, \mathcal{P}]\!] \overset{\diamond}{\equiv} [\![E', \mathcal{P}']\!]$ or $[\![E, \mathcal{P}]\!] \rightarrow_\diamond [\![E', \mathcal{P}']\!]$.

Proof. By cases on the applied reduction rule.

- Case (Red Nil). We have $\mathcal{P} = \mathcal{P}' \cup \{\mathbf{0}\}$. When $\mathcal{P}' \neq \emptyset$, we have $[\![E, \mathcal{P}]\!] \overset{\diamond}{\equiv} [\![E', \mathcal{P}']\!]$ by Par-$\mathbf{0}$. When $\mathcal{P}' = \emptyset$, we have $[\![E, \mathcal{P}]\!] = [\![E', \mathcal{P}']\!]$, so we also have $[\![E, \mathcal{P}]\!] \overset{\diamond}{\equiv} [\![E', \mathcal{P}']\!]$.
- Case (Red Par). We have $[\![E, \mathcal{P}]\!] \overset{\diamond}{\equiv} [\![E', \mathcal{P}']\!]$ by Par-A and Par-C.

- Case (Red Repl). We have $[\![E, \mathcal{P}]\!] \overset{\diamond}{\equiv} [\![E', \mathcal{P}']\!]$ by Repl.
- Case (Red Res). We have $[\![E, \mathcal{P}]\!] \overset{\diamond}{\equiv} [\![E', \mathcal{P}']\!]$ by New-Par.
- Case (Red I/O″). Recall that the channels and output message do not contain destructors, so we have

$$E, \mathcal{P} = E, \mathcal{P}_0 \cup \{\, \mathsf{out}(N, M); Q, \mathsf{in}(N', x); P \,\} \to \\ E, \mathcal{P}_0 \cup \{\, Q, P\{^{M}/_{x}\} \,\} = E', \mathcal{P}'$$

  where $N =_{\mathcal{E}} N'$. By Rewrite, we can replace N' with N in the input channel of the encoded process, so we have $[\![E, \mathcal{P}]\!] \to_{\diamond} [\![E', \mathcal{P}']\!]$ by Comm.
- Case (Red Cond 1″). We have

$$E, \mathcal{P} = E, \mathcal{P}_0 \cup \{\mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q\} \to E, \mathcal{P}_0 \cup \{P\} = E', \mathcal{P}'$$

  where $D \Downarrow M'$ and $M' =_{\mathcal{E}} \mathsf{true}$. Recall that all conditions are of the form $D = (M = N)$. Since $D \Downarrow M'$ and $M' =_{\mathcal{E}} \mathsf{true}$, we have $M =_{\mathcal{E}} N$. By Rewrite, we can replace N with M in the condition of the encoded process, so we have $[\![E, \mathcal{P}]\!] \to_{\diamond} [\![E', \mathcal{P}']\!]$ by Then.
- Case (Red Cond 2″). We have

$$E, \mathcal{P} = E, \mathcal{P}_0 \cup \{\mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q\} \to E, \mathcal{P}_0 \cup \{Q\} = E', \mathcal{P}'$$

  where $D \Downarrow M$ and $M \neq_{\mathcal{E}} \mathsf{true}$. Recall that all conditions are of the form $D = (M = N)$. Since $D \Downarrow M'$ and $M' \neq_{\mathcal{E}} \mathsf{true}$, we have $M \neq_{\mathcal{E}} N$. The terms M and N are ground since configurations contain closed processes, so we have $[\![E, \mathcal{P}]\!] \to_{\diamond} [\![E', \mathcal{P}']\!]$ by Else.

All cases may use Par-A and Par-C to reorganize parallel compositions and New-C to reorder restrictions. □

We write $\prod_{i=1}^{n} P_i$ for any parallel composition $P_1 \mid \cdots \mid P_n$ modulo associativity and commutativity. We define the property $\mathit{Prop}(E, \mathcal{P}, \mathcal{P}_{\mathrm{red}}, P')$ by $\mathit{Prop}(E, \mathcal{P}, \mathcal{P}_{\mathrm{red}}, P')$ if and only if $E = (\mathcal{N}_{\mathrm{pub}}, \{\widetilde{a}\})$, $\mathcal{P} = \{P_1, \ldots, P_n\}$, and

1. $\mathcal{P}_{\mathrm{red}} = \{P_i\}$, $[\![P_i]\!] \to_\diamond P''$ and $P' \equiv \nu\widetilde{a}.(P'' \mid \prod_{j\neq i}[\![P_j]\!])$ for some $i \in \{1,\ldots,n\}$ and P'', or

2. $\mathcal{P}_{\mathrm{red}} = \{P_i, P_j\}$, $[\![P_i]\!] \xrightarrow{N(x)}_\diamond P''$, $[\![P_j]\!] \xrightarrow{\nu x.\overline{N}\langle x\rangle}_\diamond A''$, and $P' \equiv \nu\widetilde{a}, x.((P'' \mid A'') \mid \prod_{k\neq i,j}[\![P_k]\!])$ for some $i, j \in \{1,\ldots,n\}$, P'', A'', x, and ground term N.

Intuitively, this property means that $[\![E, \mathcal{P}]\!] \to_\diamond P'$ by reducing the processes in $\mathcal{P}_{\mathrm{red}}$.

Lemma B.2. If $[\![E, \mathcal{P}]\!] \to_\diamond P$, then $\mathit{Prop}(E, \mathcal{P}, \mathcal{P}_{\mathrm{red}}, P)$ for some $\mathcal{P}_{\mathrm{red}}$.

Proof. Let $E = (\mathcal{N}_{\mathrm{pub}}, \{\widetilde{a}\})$ and $\mathcal{P} = \{P_1,\ldots,P_n\}$. We have $\nu\widetilde{a}.([\![P_1]\!] \mid \cdots \mid [\![P_n]\!]) \to_\diamond P$. By (Abadi *et al.*, 2016, Lemma B.21(2)), $[\![P_1]\!] \mid \cdots \mid [\![P_n]\!] \to_\diamond P'$ and $P \equiv \nu\widetilde{a}.P'$ for some P'.

We show the following property by induction on the cardinal of I, using (Abadi *et al.*, 2016, Lemma B.18(1)):

P1. if $\prod_{i\in I} P_i$ is closed, α is $\nu x.\overline{N}\langle x\rangle$ or $N(M)$ for some ground N and $\prod_{i\in I} P_i \xrightarrow{\alpha}_\diamond P'$, then there exist $i \in I$ and P'' such that $P_i \xrightarrow{\alpha}_\diamond P''$ and $P' \equiv P'' \mid \prod_{j\in I\setminus\{i\}} P_j$

Next, we show the following property:

P2. if $\prod_{i\in I} P_i$ is closed and $\prod_{i\in I} P_i \to_\diamond P'$, then either there exist $i \in I$ and P'' such that $P_i \to_\diamond P''$ and $P' \equiv P'' \mid \prod_{j\in I\setminus\{i\}}[\![P_j]\!]$, or there exist $i, j \in I$, P'', A'', x and a ground term N such that $P_i \xrightarrow{N(x)}_\diamond P''$, $P_j \xrightarrow{\nu x.\overline{N}\langle x\rangle}_\diamond A''$, and $P' \equiv \nu x.((P'' \mid A'') \mid \prod_{k\in I\setminus\{i,j\}} P_k)$.

by induction on the cardinal of I. If $|I| = 1$, then the first case obviously holds. If $|I| > 1$, then $\prod_{i\in I} P_i = (\prod_{i\in I_1} P_i) \mid (\prod_{i\in I_2} P_i)$ for two disjoint non-empty sets I_1 and I_2 such that $I = I_1 \cup I_2$. By (Abadi *et al.*, 2016, Lemma B.21(1)), one of the following four cases holds:

1. $\prod_{i\in I_1} P_i \to_\diamond P''$ and $P' \equiv P'' \mid (\prod_{i\in I_2} P_i)$ for some P'';

2. $\prod_{i\in I_2} P_i \to_\diamond P''$ and $P' \equiv P'' \mid (\prod_{i\in I_1} P_i)$ for some P'';

3. $\prod_{i\in I_1} P_i \xrightarrow{N(x)}_\diamond P''$, $\prod_{i\in I_2} P_i \xrightarrow{\nu x.\overline{N}\langle x\rangle}_\diamond A''$, and $P' \equiv \nu x.(P'' \mid A'')$ for some P'', A'', x, and ground term N;

4. $\prod_{i\in I_1} P_i \xrightarrow{\nu x.\overline{N}\langle x\rangle}_\diamond A''$, $\prod_{i\in I_2} P_i \xrightarrow{N(x)}_\diamond P''$, and $P' \equiv \nu x.(P'' \mid A'')$ for some P'', A'', x, and ground term N.

In the first two cases, we conclude by induction hypothesis. In the last two cases, we conclude by P1.

By applying P2 to $\prod_{i=1}^n [\![P_i]\!] \to_\diamond P'$, either there exist $i \in \{1,\ldots,n\}$ and P'' such that $[\![P_i]\!] \to_\diamond P''$ and $P' \equiv P'' \mid \prod_{j\neq i} [\![P_j]\!]$, or there exist $i,j \in \{1,\ldots,n\}$, P'', A'', x, and a ground term N such that $[\![P_i]\!] \xrightarrow{N(x)}_\diamond P''$, $[\![P_j]\!] \xrightarrow{\nu x.\overline{N}\langle x\rangle}_\diamond A''$, and $P' \equiv \nu x.((P'' \mid A'') \mid \prod_{k\neq i,j} [\![P_k]\!])$. In the first case, we obtain $\mathit{Prop}(E, \mathcal{P}, \mathcal{P}_{\mathrm{red}}, P)$ with $\mathcal{P}_{\mathrm{red}} = \{P_i\}$, and in the second case, we obtain $\mathit{Prop}(E, \mathcal{P}, \mathcal{P}_{\mathrm{red}}, P)$ with $\mathcal{P}_{\mathrm{red}} = \{P_i, P_j\}$. □

As in (Abadi *et al.*, 2016), we define the size of processes by induction on the syntax, such that $\mathit{size}(!P) = 1 + 2 \times \mathit{size}(P)$ and, when P is not a replication, $\mathit{size}(P)$ is one plus the size of the immediate subprocesses of P. When $\mathcal{P} = \{P_1,\ldots,P_n\}$ is a multiset of processes, we define $\mathit{size}(\mathcal{P}) = \mathit{size}(P_1) + \cdots + \mathit{size}(P_n)$.

Lemma B.3. If $[\![E, \mathcal{P}]\!] \to_\diamond P'$, then $E, \mathcal{P} \to^* E', \mathcal{P}'$ and $P' \overset{\diamond}{\equiv} [\![E', \mathcal{P}']\!]$ for some $E', \mathcal{P}'$.

Proof. Let $E = (\mathcal{N}_{\mathrm{pub}}, \{\widetilde{a}\})$ and $\mathcal{P} = \{P_1,\ldots,P_n\}$. By Lemma B.2, there exists $\mathcal{P}_{\mathrm{red}}$ such that $\mathit{Prop}(E, \mathcal{P}, \mathcal{P}_{\mathrm{red}}, P')$. We proceed by induction on $\mathit{size}(\mathcal{P}_{\mathrm{red}})$.

- Case $\mathcal{P}_{\mathrm{red}} = \{P_i\}$, $[\![P_i]\!] \to_\diamond P''$, and $P' \equiv \nu\widetilde{a}.(P'' \mid \prod_{j\neq i} [\![P_j]\!])$ for some $i \in \{1,\ldots,n\}$ and P''. By (Abadi *et al.*, 2016, Lemma B.21), we have the following cases:

  1. $[\![P_i]\!] = Q_1 \mid Q_2$ for some Q_1 and Q_2, and one of the following cases holds:
     (a) $Q_1 \to_\diamond Q_1'$ and $P'' \equiv Q_1' \mid Q_2$ for some Q_1',
     (b) $Q_1 \xrightarrow{N(x)}_\diamond A$, $Q_2 \xrightarrow{\nu x.\overline{N}\langle x\rangle}_\diamond B$, and $P'' \equiv \nu x.(A \mid B)$ for some A, B, x, and ground term N,

     and two symmetric cases obtained by swapping Q_1 and Q_2.

By definition of $[\![\cdot]\!]$, we have $P_i = Q_1'' \mid Q_2''$, $Q_1 = [\![Q_1'']\!]$, and $Q_2 = [\![Q_2'']\!]$ for some Q_1'' and Q_2''. By (Red Par), $E, \mathcal{P} \to E, \mathcal{P}'$ where $\mathcal{P}' = \mathcal{P} \backslash \{P_i\} \cup \{Q_1'', Q_2''\}$. We have $Prop(E, \mathcal{P}', \mathcal{P}'_{\text{red}}, P')$ with $\mathcal{P}'_{\text{red}} = \{Q_1''\}$ (case a), $\mathcal{P}'_{\text{red}} = \{Q_2''\}$ (symmetric of case a), or $\mathcal{P}'_{\text{red}} = \{Q_1'', Q_2''\}$ (case b and its symmetric), and $size(\mathcal{P}'_{\text{red}}) < size(\mathcal{P}_{\text{red}})$ in all cases, so we conclude by induction hypothesis.

2. $[\![P_i]\!] = \nu n.Q$, $Q \to_\diamond Q'$, and $P'' \equiv \nu n.Q'$ for some n, Q, and Q'.

   We have $P_i = \mathsf{new}\ n; Q''$ and $Q = [\![Q'']\!]$ for some Q''. By (Red Res), $E, \mathcal{P} \to E', \mathcal{P}'$ where $E' = (\mathcal{N}_{\text{pub}}, \{\tilde{a}\} \cup \{n'\})$, $\mathcal{P}' = \mathcal{P} \setminus \{P_i\} \cup \left\{Q''\{^{n'}/_{n}\}\right\}$, and $n' \notin \mathcal{N}_{\text{pub}} \cup \{\tilde{a}\}$. We have $[\![Q''\{^{n'}/_{n}\}]\!] = Q\{^{n'}/_{n}\} \to_\diamond Q'\{^{n'}/_{n}\}$. (We show by induction on the derivation of $Q \to_\diamond Q'$ that, if $Q \to_\diamond Q'$, then $\sigma Q \to_\diamond \sigma Q'$ where σ is a bijective renaming.) So we have $Prop(E', \mathcal{P}', \mathcal{P}'_{\text{red}}, P')$ with $\mathcal{P}'_{\text{red}} = \left\{Q''\{^{n'}/_{n}\}\right\}$ and $size(\mathcal{P}'_{\text{red}}) < size(\mathcal{P}_{\text{red}})$, so we conclude by induction hypothesis.

3. $[\![P_i]\!] = !Q$, $Q \mid Q \to_\diamond Q'$, and $P'' \equiv Q' \mid !Q$ for some Q and Q'.

   We have $P_i = !Q''$ and $Q = [\![Q'']\!]$ for some Q''. By (Red Repl) twice, $E, \mathcal{P} \to^* E, \mathcal{P}'$ where $\mathcal{P}' = \mathcal{P} \cup \{Q'', Q''\}$. Since $Q \mid Q \to_\diamond Q'$, by (Abadi *et al.*, 2016, Lemma B.21), one of the following cases holds:

   (a) $Q \to_\diamond Q_1'$ and $Q' \equiv Q_1' \mid Q$ for some Q_1',

   (b) $Q \xrightarrow{N(x)}_\diamond A$, $Q \xrightarrow{\nu x.\overline{N}\langle x\rangle}_\diamond B$, and $Q' \equiv \nu x.(A \mid B)$ for some A, B, x, and ground term N,

   so we have $Prop(E, \mathcal{P}', \mathcal{P}'_{\text{red}}, P')$ with $\mathcal{P}'_{\text{red}} = \{Q''\}$ (case a) or $\mathcal{P}'_{\text{red}} = \{Q'', Q''\}$ (case b), and $size(\mathcal{P}'_{\text{red}}) < size(\mathcal{P}_{\text{red}})$ in all cases, so we conclude by induction hypothesis.

4. $[\![P_i]\!] = \mathsf{if}\ M = N\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2$ and either $M =_{\mathcal{E}} N$ and $P'' \equiv Q_1$, or $M \neq_{\mathcal{E}} N$ and $P'' \equiv Q_2$, for some M, N, Q_1, and Q_2.

We have $P_i = \mathsf{if}\ M = N\ \mathsf{then}\ Q'_1\ \mathsf{else}\ Q'_2$, $Q_1 = [\![Q'_1]\!]$, and $Q_2 = [\![Q'_2]\!]$ for some Q'_1 and Q'_2.

If $M =_{\mathcal{E}} N$, then by (Red Cond 1″), $E, \mathcal{P} \to E, \mathcal{P}'$ where $\mathcal{P}' = \mathcal{P} \setminus \{P_i\} \cup \{Q'_1\}$. We have $[\![E, \mathcal{P}']\!] = \mathsf{new}\ \tilde{a}; ([\![Q'_1]\!] \mid \prod_{j \neq i} [\![P_j]\!]) = \mathsf{new}\ \tilde{a}; (Q_1 \mid \prod_{j \neq i} [\![P_j]\!])$ and $P' \equiv \nu \tilde{a}.(P'' \mid \prod_{j \neq i} [\![P_j]\!]) \equiv [\![E, \mathcal{P}']\!]$.

If $M \neq_{\mathcal{E}} N$, then by (Red Cond 2″), $E, \mathcal{P} \to E, \mathcal{P}'$ where $\mathcal{P}' = \mathcal{P} \setminus \{P_i\} \cup \{Q'_2\}$. We have $[\![E, \mathcal{P}']\!] = \mathsf{new}\ \tilde{a}; ([\![Q'_2]\!] \mid \prod_{j \neq i} [\![P_j]\!]) = \mathsf{new}\ \tilde{a}; (Q_2 \mid \prod_{j \neq i} [\![P_j]\!])$ and $P' \equiv \nu \tilde{a}.(P'' \mid \prod_{j \neq i} [\![P_j]\!]) \equiv [\![E, \mathcal{P}']\!]$.

- Case $\mathcal{P}_{\text{red}} = \{P_i, P_j\}$, $[\![P_i]\!] \xrightarrow{N(x)}_{\diamond} P''$, $[\![P_j]\!] \xrightarrow{\nu x.\overline{N}\langle x \rangle}_{\diamond} A''$, and $P' \equiv \nu \tilde{a}, x.((P'' \mid A'') \mid \prod_{k \neq i,j} [\![P_k]\!])$ for some $i, j \in \{1, \ldots, n\}$, P'', A'', x, and ground term N. By (Abadi *et al.*, 2016, Lemma B.18) applied to the transition $[\![P_i]\!] \xrightarrow{N(x)}_{\diamond} P''$, we have the following cases:

  1. $[\![P_i]\!] = Q_1 \mid Q_2$ and either $Q_1 \xrightarrow{N(x)}_{\diamond} A'$ and $P'' \equiv A' \mid Q_2$, or $Q_2 \xrightarrow{N(x)}_{\diamond} A'$ and $A \equiv Q_1 \mid A'$, for some Q_1, Q_2, and A'.

     We have $P_i = Q'_1 \mid Q'_2$, $Q_1 = [\![Q'_1]\!]$, and $Q_2 = [\![Q'_2]\!]$ for some Q'_1 and Q'_2. By (Red Par), $E, \mathcal{P} \to E, \mathcal{P}'$ where $\mathcal{P}' = \mathcal{P} \setminus \{P_i\} \cup \{Q'_1, Q'_2\}$. We have $Prop(E, \mathcal{P}', \mathcal{P}'_{\text{red}}, P')$ with $\mathcal{P}'_{\text{red}} = \{Q'_1, P_j\}$ or $\mathcal{P}'_{\text{red}} = \{Q'_2, P_j\}$ and $size(\mathcal{P}'_{\text{red}}) < size(\mathcal{P}_{\text{red}})$, so we conclude by induction hypothesis.

  2. $[\![P_i]\!] = \nu n.Q$, $Q \xrightarrow{N(x)}_{\diamond} A'$, and $P'' \equiv \nu n.A'$ for some Q, A', and n that does not occur in $N(x)$.

     We have $P_i = \mathsf{new}\ n; Q'$ and $Q = [\![Q']\!]$ for some Q'. By (Red Res), $E, \mathcal{P} \to E', \mathcal{P}'$ where $E' = (\mathcal{N}_{\text{pub}}, \{\tilde{a}\} \cup \{n'\})$, $\mathcal{P}' = \mathcal{P} \setminus \{P_i\} \cup \left\{Q'\{^{n'}/_{n}\}\right\}$, and $n' \notin \mathcal{N}_{\text{pub}} \cup \{\tilde{a}\}$. We have $[\![Q'\{^{n'}/_{n}\}]\!] = Q\{^{n'}/_{n}\} \xrightarrow{N(x)}_{\diamond} A'\{^{n'}/_{n}\}$. (We show by induction on the derivation of $Q \xrightarrow{N(x)}_{\diamond} A'$ that, if $Q \xrightarrow{N(x)}_{\diamond} A'$, then $\sigma Q \xrightarrow{\sigma N(x)}_{\diamond} \sigma A'$ where σ is a bijective renaming.) So we have $Prop(E', \mathcal{P}', \mathcal{P}'_{\text{red}}, P')$ with $\mathcal{P}'_{\text{red}} = \left\{Q'\{^{n'}/_{n}\}, P_j\right\}$

and $size(\mathcal{P}'_{\text{red}}) < size(\mathcal{P}_{\text{red}})$, so we conclude by induction hypothesis.

3. $[\![P_i]\!] = !Q$, $Q \xrightarrow{N(x)}_{\diamond} A'$, and $P'' \equiv A' \mid !Q$ for some Q and A'.
We have $P_i = !Q'$ and $Q = [\![Q']\!]$ for some Q'. By (Red Repl), $E, \mathcal{P} \to E, \mathcal{P}'$ where $\mathcal{P}' = \mathcal{P} \cup \{Q'\}$. We have $Prop(E', \mathcal{P}', \mathcal{P}'_{\text{red}}, P')$ with $\mathcal{P}'_{\text{red}} = \{Q', P_j\}$ and $size(\mathcal{P}'_{\text{red}}) < size(\mathcal{P}_{\text{red}})$, so we conclude by induction hypothesis.

4. $[\![P_i]\!] = N'(x').Q_1$, $N =_{\mathcal{E}} N'$, and $P'' \equiv Q_1\{^{x}/_{x'}\}$ for some Q_1, N', and x'.
By (Abadi *et al.*, 2016, Lemma B.18) applied to the transition $[\![P_j]\!] \xrightarrow{\nu x.\overline{N}\langle x\rangle}_{\diamond} A''$, we distinguish cases on $[\![P_j]\!]$. The first three cases are similar to those for $[\![P_i]\!]$. The last case is $[\![P_j]\!] = \overline{N''}\langle M''\rangle.Q_2$, $N =_{\mathcal{E}} N''$, $x \notin fv([\![P_j]\!])$, and $A'' \equiv Q_2 \mid \{^{M''}/_{x}\}$ for some N'', M'', and Q_2.
We have $P_i = \mathsf{in}(N', x'); Q'_1$ and $Q_1 = [\![Q'_1]\!]$ for some Q'_1. We also have $P_j = \mathsf{out}(N'', M''); Q'_2$ and $Q_2 = [\![Q'_2]\!]$ for some Q'_2. By (Red I/O″), $E, \mathcal{P} \to E, \mathcal{P}'$ where $\mathcal{P}' = \mathcal{P} \setminus \{P_i, P_j\} \cup \{Q'_1\{^{M''}/_{x'}\}, Q'_2\}$ since $N' =_{\mathcal{E}} N''$. Therefore,

$$\begin{aligned}[\![E, \mathcal{P}']\!] &= \nu\tilde{a}.([\![Q'_1\{^{M''}/_{x'}\}]\!] \mid [\![Q'_2]\!] \mid \prod_{k\neq i,j} [\![P_k]\!]) \\ &= \nu\tilde{a}.(Q_1\{^{M''}/_{x'}\} \mid Q_2 \mid \prod_{k\neq i,j} [\![P_k]\!])\,.\end{aligned}$$

Moreover,

$$\begin{aligned}P' &\equiv \nu\tilde{a}, x.((P'' \mid A'') \mid \prod_{k\neq i,j} [\![P_k]\!]) \\ &\equiv \nu\tilde{a}, x.(Q_1\{^{x}/_{x'}\} \mid Q_2 \mid \{^{M''}/_{x}\} \mid \prod_{k\neq i,j} [\![P_k]\!]) \\ P' &\equiv \nu\tilde{a}.(Q_1\{^{M''}/_{x'}\} \mid Q_2 \mid \prod_{k\neq i,j} [\![P_k]\!])\end{aligned}$$

since x does not occur free in Q_2 and P_k for $k \neq i, j$ since these are closed processes. So $P' \equiv [\![E, \mathcal{P}']\!]$. □

Proposition 4.1. If $\mathcal{C} \to^* \mathcal{C}'$, then $[\![\mathcal{C}]\!] \to_{\diamond}{}^* \overset{\diamond}{\equiv} [\![\mathcal{C}']\!]$.
Conversely, if $[\![\mathcal{C}]\!] \to_{\diamond}{}^* P'$, then $\mathcal{C} \to^* \mathcal{C}'$ and $P' \overset{\diamond}{\equiv} [\![\mathcal{C}']\!]$ for some $\mathcal{C}'$.

Proof. The first point is proved by induction on the number of steps of $\mathcal{C} \to^* \mathcal{C}'$, using Lemma B.1. The second point is proved by induction on the number of steps of $[\![\mathcal{C}]\!] \to_\diamond^* P'$, using Lemma B.3. □

B.2 Proof of Propositions 4.2 and 4.3

Let *replication contexts* $C_!{[\_]}$ be plain contexts generated by the following grammar:

$C_!{[\_]} ::=$	replication contexts
$\_$	hole
$P \mid C_!{[\_]}$	parallel composition
$C_!{[\_]} \mid P$	parallel composition
$\nu a.C_!{[\_]}$	restriction
$!C_!{[\_]}$	replication

We use replication contexts to characterize $P \Downarrow_a^\diamond$ without using structural equivalence, by the following lemma:

Lemma B.4. We have $P \stackrel{\diamond}{\equiv} C[\overline{a}\langle M\rangle.Q]$ for some M, Q, and evaluation context $C[\_]$ that does not bind a if and only if $P = C'_![\overline{N'}\langle M'\rangle.Q']$ and $N' =_\mathcal{E} a$ for some N', M', Q', and replication context $C'_![\_]$ that does not bind a.

Proof. The implication from left to right is an immediate consequence of the following property: If $P \stackrel{\diamond}{\equiv} C_![\overline{N}\langle M\rangle.Q]$ and $N =_\mathcal{E} a$ for some N, M, Q, and replication context $C_![\_]$ that does not bind a, then $P = C'_![\overline{N'}\langle M'\rangle.Q']$ and $N' =_\mathcal{E} a$ for some N', M', Q', and replication context $C'_![\_]$ that does not bind a. This property is proved by induction on the derivation of $P \stackrel{\diamond}{\equiv} C_![\overline{a}\langle M\rangle.Q]$.

The converse is proved by unfolding replications in $C'_!$ by Repl and by replacing N' with a by Rewrite. □

Lemma B.5. Let $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}$ be a valid configuration and $a \in \mathcal{N}_{\text{pub}}$. If $C_![\overline{N}\langle M\rangle.Q] = [\![P]\!]$, and $N =_\mathcal{E} a$ for some $P \in \mathcal{P}$, N, M, Q, and replication context $C_!$ that does not bind a, then $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P} \to^* (\mathcal{N}_{\text{pub}}, \mathcal{N}'_{\text{priv}}), \mathcal{P}'$ with $\overline{N'}\langle M'\rangle.Q' = [\![P']\!]$ and $N' =_\mathcal{E} a$ for some $\mathcal{N}'_{\text{priv}}$, $\mathcal{P}'$, $P' \in \mathcal{P}'$, N', M', Q'.

Proof. This lemma is proved by reducing the context $C_!$ by (Red Par), (Red Res), and (Red Repl). More formally, the proof proceeds by induction on the size of the replication context $C_![\_]$.

- If $C_![\_] = \_$, the result is obvious.
- If $C_![\_] = P' \mid C'_![\_]$, then $P = P'_1 \mid P_1$ with $[\![P'_1]\!] = P'$ and $[\![P_1]\!] = C'_![\overline{N}\langle M\rangle.Q]$. We have $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P} = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}_0 \cup \{P'_1 \mid P_1\} \to (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}_0 \cup \{P'_1, P_1\}$ by (Red Par). We conclude by induction hypothesis with P_1 instead of P.
- The case $C_![\_] = C'_![\_] \mid P'$ is similar.
- If $C_![\_] = \nu b.C'_![\_]$, then $P = \mathsf{new}\ b; P_1$ with $[\![P_1]\!] = C'_![\overline{N}\langle M\rangle.Q]$. We have

$$\begin{aligned}(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P} =& (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}_0 \cup \{\mathsf{new}\ b; P_1\} \\ \to& (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}} \cup \{b'\}), \mathcal{P}_0 \cup \left\{P_1\{^{b'}/_b\}\right\}\end{aligned}$$

  for some $b' \notin \mathcal{N}_{\text{pub}} \cup \mathcal{N}_{\text{priv}}$ by (Red Res). We have $[\![P_1\{^{b'}/_b\}]\!] = C'_![\overline{N\{^{b'}/_b\}}\langle M\{^{b'}/_b\}\rangle.Q\{^{b'}/_b\}]$ and $N\{^{b'}/_b\} =_{\mathcal{E}} a$ since $a \neq b$ because $C_!$ does not bind a. We conclude by induction hypothesis with $P_1\{^{b'}/_b\}$ instead of P.
- If $C_![\_] = !C'_![\_]$, then $P = !P_1$ with $[\![P_1]\!] = C'_![\overline{N}\langle M\rangle.Q]$. We have $(\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P} = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}_0 \cup \{!P_1\} \to (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}}), \mathcal{P}_0 \cup \{!P_1, P_1\}$ by (Red Repl). We conclude by induction hypothesis with P_1 instead of P. □

Lemma B.6. Let $E, \mathcal{P}$ be a valid configuration. We have $[\![E, \mathcal{P}]\!] \Downarrow^{\diamond}_a$ if and only if $E, \mathcal{P} \to^* \downarrow_a$.

Proof. Suppose $[\![E, \mathcal{P}]\!] \Downarrow^{\diamond}_a$. We have $[\![E, \mathcal{P}]\!] \to_{\diamond}{}^* \overset{\diamond}{\equiv} C[\overline{a}\langle M\rangle.P]$ for some M, P and evaluation context $C[\_]$ that does not bind a. By Proposition 4.1, $E, \mathcal{P} \to^* E', \mathcal{P}'$ and $[\![E', \mathcal{P}']\!] \overset{\diamond}{\equiv} C[\overline{a}\langle M\rangle.P]$ for some $E', \mathcal{P}'$. By Lemma B.4, $[\![E', \mathcal{P}']\!] = C'_![\overline{N'}\langle M'\rangle.P']$ and $N' =_{\mathcal{E}} a$ for some N', M', P', and replication context $C'_!$ that does not bind a. Let $E' = (\mathcal{N}_{\text{pub}}, \{\widetilde{a}\})$ and $\mathcal{P}' = \{P_1, \ldots, P_n\}$. We have $[\![E', \mathcal{P}']\!] = \nu\widetilde{a}.([\![P_1]\!] \mid \cdots \mid [\![P_n]\!])$. Then $[\![P_i]\!] = C''_![\overline{N'}\langle M'\rangle.P']$ for some replication context $C''_!$ that does not

bind a and $P_i \in \mathcal{P}'$, and $a \notin \tilde{a}$. Since $N' =_{\mathcal{E}} a$, we have $a \in fn(N')$. (Otherwise, we would have $N'\{^{N''}/_a\} =_{\mathcal{E}} a\{^{N''}/_a\}$ for all N'', since equality is preserved by substitution of terms for names because the equations do not contain names, so $N' =_{\mathcal{E}} N''$, which contradicts the assumption that the equational theory is non trivial.) So $a \in fn(P_i)$, so $a \in fn(\mathcal{P}')$. Since $fn(\mathcal{P}') \subseteq \mathcal{N}_{\text{pub}} \cup \{\tilde{a}\}$ and $a \notin \tilde{a}$, we have $a \in \mathcal{N}_{\text{pub}}$. By Lemma B.5, $E', \mathcal{P}' \to^* E'', \mathcal{P}''$, $\overline{N''}\langle M''\rangle.P'' = [\![Q]\!]$, and $N'' =_{\mathcal{E}} a$ for some $E'' = (\mathcal{N}_{\text{pub}}, \mathcal{N}'_{\text{priv}})$, $\mathcal{P}''$, $Q \in \mathcal{P}''$, N'', M'', and P''. Therefore $Q = \mathsf{out}(N'', M''); Q' \in \mathcal{P}''$ for some Q', $N'' =_{\mathcal{E}} a$, and $a \in \mathcal{N}_{\text{pub}}$, so $E'', \mathcal{P}'' \downarrow_a$, so $E, \mathcal{P} \to^* \downarrow_a$.

Conversely, suppose $E, \mathcal{P} \to^* \downarrow_a$. We have $E, \mathcal{P} \to^* E', \mathcal{P}'$ and $E', \mathcal{P}' \downarrow_a$ for some $E', \mathcal{P}'$. So we have $E' = (\mathcal{N}_{\text{pub}}, \{\tilde{a}\})$, $a \in \mathcal{N}_{\text{pub}}$, $\mathsf{out}(N, M); Q \in \mathcal{P}'$, and $N =_{\mathcal{E}} a$ for some $\mathcal{N}_{\text{pub}}$, $\tilde{a}$, N, M, and Q. We have $[\![E', \mathcal{P}']\!] = \nu\tilde{a}.([\![P_1]\!] \mid \cdots \mid [\![P_n]\!])$ where $\mathcal{P}' = \{P_1, \ldots, P_n\}$, and $[\![\mathsf{out}(N, M); Q]\!] = \overline{N}\langle M\rangle.[\![Q]\!]$, so $[\![E', \mathcal{P}']\!] = C[\overline{N}\langle M\rangle.[\![Q]\!]]$ where $C[\_]$ is an evaluation context that does not bind a, since $a \notin \tilde{a}$. So $[\![E', \mathcal{P}']\!] \overset{\diamond}{\equiv} C[\overline{a}\langle M\rangle.[\![Q]\!]]$ by Rewrite. By Proposition 4.1, $[\![E, \mathcal{P}]\!] \to_\diamond{}^* \overset{\diamond}{\equiv} [\![E', \mathcal{P}']\!] \overset{\diamond}{\equiv} C[\overline{a}\langle M\rangle.[\![Q]\!]]$, so $[\![E, \mathcal{P}]\!] \Downarrow^\diamond_a$. □

We define the encoding of contexts as the encoding of processes, with additionally $[\![\_]\!] = \_$.

Lemma B.7. Let $C[\_]$ be an adversarial context. We have $[\![C[\mathcal{C}]]\!] \overset{\diamond}{\equiv} [\![C]\!][[\![\mathcal{C}]\!]]$.

Proof. Let $C[\_] = \mathsf{new}\ \tilde{n}; (\_ \mid Q)$ with $fv(Q) = \emptyset$ and $\mathcal{C} = (\mathcal{N}_{\text{pub}}, \mathcal{N}_{\text{priv}})$, $\mathcal{P}$. We rename the names in $\mathcal{N}_{\text{priv}}$ so that $\mathcal{N}_{\text{priv}} \cap fn(Q) = \emptyset$. Let $\mathcal{N}_{\text{priv}} = \{\tilde{a}\}$ and $\mathcal{P} = \{P_1, \ldots, P_n\}$. We have $C[\mathcal{C}] = ((\mathcal{N}_{\text{pub}} \cup fn(Q)) \setminus \{\tilde{n}\}, \mathcal{N}_{\text{priv}} \cup \{\tilde{n}\}), \mathcal{P} \cup \{Q\}$, so

$$\begin{aligned}[\![C[\mathcal{C}]]\!] &= \nu\tilde{a}, \tilde{n}.([\![P_1]\!] \mid \cdots \mid [\![P_n]\!] \mid [\![Q]\!]) \\ &\overset{\diamond}{\equiv} \nu\tilde{n}.(\nu\tilde{a}.([\![P_1]\!] \mid \cdots \mid [\![P_n]\!]) \mid [\![Q]\!]) = [\![C]\!][[\![\mathcal{C}]\!]]\,.\end{aligned}$$

□

Proposition 4.2. Let $\mathcal{C}$ and $\mathcal{C}'$ be valid configurations. If $\mathcal{C} \approx \mathcal{C}'$, then $[\![\mathcal{C}]\!] \overset{\diamond}{\approx} [\![\mathcal{C}']\!]$.

Proof. We define a relation $\mathcal{R}$ between closed processes by $P \mathcal{R} Q$ if and only if $P \overset{\diamond}{\equiv} [\![\mathcal{C}]\!]$, $Q \overset{\diamond}{\equiv} [\![\mathcal{C}']\!]$, and $\mathcal{C} \approx \mathcal{C}'$ for some $\mathcal{C}$ and $\mathcal{C}'$. The relation $\mathcal{R}$ is symmetric. Let us show that it satisfies the three conditions of Definition 4.1:

1. If $P \mathcal{R} Q$ and $P \Downarrow_a^{\diamond}$, then $P \overset{\diamond}{\equiv} [\![\mathcal{C}]\!]$, $Q \overset{\diamond}{\equiv} [\![\mathcal{C}']\!]$, and $\mathcal{C} \approx \mathcal{C}'$ for some $\mathcal{C}$ and $\mathcal{C}'$, so $[\![\mathcal{C}]\!] \Downarrow_a^{\diamond}$, so by Lemma B.6, $\mathcal{C} \to^* \downarrow_a$. Since $\mathcal{C} \approx \mathcal{C}'$, we have $\mathcal{C}' \to^* \downarrow_a$, so by Lemma B.6, $[\![\mathcal{C}']\!] \Downarrow_a^{\diamond}$, so $Q \Downarrow_a^{\diamond}$.

2. Suppose $P \mathcal{R} Q$, $P \to_{\diamond}{}^* P'$, and P' is closed. We have $P \overset{\diamond}{\equiv} [\![\mathcal{C}]\!]$, $Q \overset{\diamond}{\equiv} [\![\mathcal{C}']\!]$, and $\mathcal{C} \approx \mathcal{C}'$ for some $\mathcal{C}$ and $\mathcal{C}'$. If $P = P'$, then $Q \to_{\diamond}{}^* Q'$ and $P' \mathcal{R} Q'$ with $Q' = Q$. Otherwise, $[\![\mathcal{C}]\!] \to_{\diamond}{}^* P'$, so by Proposition 4.1, $\mathcal{C} \to^* \mathcal{C}_1$ and $P' \overset{\diamond}{\equiv} [\![\mathcal{C}_1]\!]$ for some $\mathcal{C}_1$. Hence $\mathcal{C}' \to^* \mathcal{C}_1'$ and $\mathcal{C}_1 \approx \mathcal{C}_1'$ for some $\mathcal{C}_1'$. By Proposition 4.1, $[\![\mathcal{C}']\!] \to_{\diamond}{}^* \overset{\diamond}{\equiv} [\![\mathcal{C}_1']\!]$.

   If there is no reduction $\to_{\diamond}$ in this trace, we define $Q' = Q$. We have $Q \to_{\diamond}{}^* Q'$. Moreover, $P' \overset{\diamond}{\equiv} [\![\mathcal{C}_1]\!]$, $\mathcal{C}_1 \approx \mathcal{C}_1'$, and $Q' = Q \overset{\diamond}{\equiv} [\![\mathcal{C}']\!] \overset{\diamond}{\equiv} [\![\mathcal{C}_1']\!]$, so $P' \mathcal{R} Q'$.

   If there is at least one reduction $\to_{\diamond}$ in this trace, we define $Q' = [\![\mathcal{C}_1']\!]$. We have $Q \overset{\diamond}{\equiv} [\![\mathcal{C}']\!] \to_{\diamond}{}^* \overset{\diamond}{\equiv} [\![\mathcal{C}_1']\!] = Q'$, so $Q \to_{\diamond}{}^* Q'$. Moreover, $P' \overset{\diamond}{\equiv} [\![\mathcal{C}_1]\!]$, $\mathcal{C}_1 \approx \mathcal{C}_1'$ and $Q' = [\![\mathcal{C}_1']\!]$, so $P' \mathcal{R} Q'$.

3. Let us finally show that, if $P \mathcal{R} Q$ and $C[\_]$ is a closed evaluation context, then $C[P] \mathcal{R} C[Q]$. Up to structural equivalence $\overset{\diamond}{\equiv}$, the context $C[\_]$ can be decomposed into several closed contexts of the form $\nu\tilde{n}.(\_ \mid Q)$, so it is enough to show the desired property for such contexts. For such a context $C[\_]$, there exists an adversarial context $C'[\_]$ such that $[\![C'[\_]]\!] = C[\_]$. Since $P \mathcal{R} Q$, we have $P \overset{\diamond}{\equiv} [\![\mathcal{C}]\!]$, $Q \overset{\diamond}{\equiv} [\![\mathcal{C}']\!]$, and $\mathcal{C} \approx \mathcal{C}'$ for some $\mathcal{C}$ and $\mathcal{C}'$. Then $C[P] \overset{\diamond}{\equiv} [\![C'][\![\mathcal{C}]\!]] \overset{\diamond}{\equiv} [\![C'[\mathcal{C}]]\!]$ and $C[Q] \overset{\diamond}{\equiv} [\![C'][\![\mathcal{C}']\!]] \overset{\diamond}{\equiv} [\![C'[\mathcal{C}']]\!]$ by Lemma B.7. Moreover, $C[\mathcal{C}] \approx C'[\mathcal{C}']$, so $C[P] \mathcal{R} C[Q]$.

Since $\overset{\diamond}{\approx}$ is the largest such relation, we have $\mathcal{R} \subseteq \overset{\diamond}{\approx}$. If $\mathcal{C} \approx \mathcal{C}'$, then $[\![\mathcal{C}]\!] \mathcal{R} [\![\mathcal{C}']\!]$, so $[\![\mathcal{C}]\!] \overset{\diamond}{\approx} [\![\mathcal{C}']\!]$. □

Lemma B.8. If P and Q are closed processes and $P \overset{\diamond}{\equiv} \overset{\diamond}{\approx} \overset{\diamond}{\equiv} Q$, then $P \overset{\diamond}{\approx} Q$.

Proof. We define the relation $\mathcal{R}$ between closed processes by $P\ \mathcal{R}\ Q$ if and only if $P \overset{\diamond}{\equiv}\overset{\diamond}{\approx}\overset{\diamond}{\equiv} Q$. The relation $\mathcal{R}$ is symmetric and satisfies the three conditions of Definition 4.1, so $\mathcal{R} \subseteq \overset{\diamond}{\approx}$. This property proves the lemma. □

Proposition 4.3. Let $\mathcal{C}$ and $\mathcal{C}'$ be valid configurations. If $[\![\mathcal{C}]\!] \overset{\diamond}{\approx} [\![\mathcal{C}']\!]$, then $\mathcal{C} \approx \mathcal{C}'$.

Proof. Since the contexts in Definition 3.6 may be any ProVerif adversarial contexts, we need to encode all ProVerif constructs into the subset already encoded by $[\![\cdot]\!]$. In this proof, we consider the core ProVerif language with equations and enriched terms. Pattern-matching and tables can be defined as an encoding into that subset, so the result extends to them. We do not consider phases. We define this encoding as follows: $[P]_{\mathsf{PV}}$ is the process obtained from P by performing the following replacements:

- If fail occurs in D or D' (in the output case), then replace $\mathsf{out}(D, D'); P$, $\mathsf{in}(D, x : T); P$, and $\mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q$ with $\mathbf{0}$, and replace $\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ with Q.
- If D is not of the form $D_1 = D_2$, then replace $\mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q$ with $\mathsf{if}\ D = \mathsf{true}\ \mathsf{then}\ P\ \mathsf{else}\ Q$.
- If $P_1 = P_0\{^{D_1=D_2}/_x\}$, P_0 is $\mathsf{out}(D, D'); P$, $\mathsf{in}(D, y : T); P$, $\mathsf{if}\ D = D'\ \mathsf{then}\ P\ \mathsf{else}\ Q$, or $\mathsf{let}\ y : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$, x occurs only in D or D' and exactly once, and fail does not occur in any of D, D', D_1, D_2, then replace P_1 with $\mathsf{if}\ D_1 = D_2\ \mathsf{then}\ P_0\{^{\mathsf{true}}/_x\}\ \mathsf{else}\ P_0\{^{\mathsf{false}}/_x\}$.
- Replace $\mathsf{let}\ x : T = M\ \mathsf{in}\ P\ \mathsf{else}\ Q$ with $P\{^M/_x\}$.

We define $[C]_{\mathsf{PV}}$ similarly, and $[E, \mathcal{P}]_{\mathsf{PV}} = E, \{[P]_{\mathsf{PV}} \mid P \in \mathcal{P}\}$.

We show that the encoding preserves observational equivalence. In the next properties, C denotes any context (any process with a hole, not only an adversarial context). We define $\mathsf{gen}(P, P') = \{((E, \{P_1, \ldots, P_n\}), (E, \{P'_1, \ldots, P'_n\})) \mid (E, \{P_1, \ldots, P_n\})$ and $(E, \{P'_1, \ldots, P'_n\})$ are valid configurations, for all $i \leq n$, $P_i = C_i[\sigma_i P]$ and $P'_i = C_i[\sigma_i P']$ for

some context C_i and substitution σ_i, or $P_i = P'_i\}$. If $(\mathcal{C}, \mathcal{C}') \in \mathsf{gen}(P, P')$ and $\mathcal{C} \to \mathcal{C}_1$ by a reduction that does not reduce processes P_i with an empty context C_i, then for some $\mathcal{C}'_1$, $\mathcal{C}' \to \mathcal{C}'_1$ by the same reduction rule and $(\mathcal{C}_1, \mathcal{C}'_1) \in \mathsf{gen}(P, P')$. The symmetric property also holds, and $\mathsf{gen}(P, P')$ is closed under application of adversarial contexts. Because of these properties, $\mathsf{gen}(P, P')$ is a good starting point for building relations used for proving observational equivalence.

We prove the following properties:

P1. If fail occurs in D, then $D \Downarrow \mathsf{fail}$. If fail does not occur in D, then $D \Downarrow M$ for some term M. These properties are proved by induction on D. They come from the fact that equality is the only considered destructor, and it evaluates to fail if and only if one of its arguments evaluates to fail.

P2. If P_0 is $\mathsf{out}(D, D'); P$, $\mathsf{in}(D, x : T); P$, or $\mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q$ and fail occurs in D or D', then $E, \mathcal{P} \cup \{C[P_0]\} \approx E, \mathcal{P} \cup \{C[\mathbf{0}]\}$. This property is proved by defining a relation $\mathcal{R}$ between valid configurations by $E, \mathcal{P} \cup \{\sigma_1 P_0, \ldots, \sigma_n P_0\}\ \mathcal{R}\ E, \mathcal{P}'$ when $((E, \mathcal{P}), (E, \mathcal{P}')) \in \mathsf{gen}(P_0, \mathbf{0})$, and showing that $\mathcal{R} \cup \mathcal{R}^{-1}$ is symmetric and satisfies the three conditions of Definition 3.6. The main argument is that P_0 never reduces.

P3. If fail occurs in D, then $E, \mathcal{P} \cup \{C[\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q]\} \approx E, \mathcal{P} \cup \{C[Q]\}$. This property is proved by defining a relation $\mathcal{R} = \mathsf{gen}(\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q, Q)$, and showing that $\mathcal{R} \cup \mathcal{R}^{-1}$ is symmetric and satisfies the three conditions of Definition 3.6. The main argument is that $\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ reduces to Q since $D \Downarrow \mathsf{fail}$ by P1.

P4. We have $E, \mathcal{P} \cup \{C[\mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q]\} \approx E, \mathcal{P} \cup \{C[\mathsf{if}\ D = \mathsf{true}\ \mathsf{then}\ P\ \mathsf{else}\ Q]\}$. This property is proved by defining a relation $\mathcal{R} = \mathsf{gen}(\mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q, \mathsf{if}\ D = \mathsf{true}\ \mathsf{then}\ P\ \mathsf{else}\ Q)$, and showing that $\mathcal{R} \cup \mathcal{R}^{-1}$ is symmetric and satisfies the three conditions of Definition 3.6. The main argument is that $\mathsf{if}\ D\ \mathsf{then}\ P\ \mathsf{else}\ Q$ and $\mathsf{if}\ D = \mathsf{true}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ either both reduce to P or both reduce to Q, because $D \Downarrow M$ with $M =_{\mathcal{E}} \mathsf{true}$ if and only if $(D = \mathsf{true}) \Downarrow M'$ with $M' =_{\mathcal{E}} \mathsf{true}$.

P5. If P_0 is $\mathsf{out}(D, D'); P$, $\mathsf{in}(D, y : T); P$, $\mathsf{if}\ D = D'\ \mathsf{then}\ P\ \mathsf{else}\ Q$, or $\mathsf{let}\ y : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$, x occurs only in D or D' and exactly once, and fail does not occur in any of D, D', D_1, D_2, then $E, \mathcal{P} \cup \{C[P_0\{^{D_1=D_2}/_x\}]\} \approx E, \mathcal{P} \cup \{C[\mathsf{if}\ D_1 = D_2\ \mathsf{then}\ P_0\{^{\mathsf{true}}/_x\}\ \mathsf{else}\ P_0\{^{\mathsf{false}}/_x\}]\}$. This property is proved by defining a relation $\mathcal{R}$ between valid configurations by $E, \mathcal{P} \cup \{\sigma_1 P_0\{^{D_1=D_2}/_x\}, \ldots, \sigma_n P_0\{^{D_1=D_2}/_x\}\}\ \mathcal{R}\ E, \mathcal{P}' \cup \{\sigma_1 P_0\{^{M_1}/_x\}, \ldots, \sigma_n P_0\{^{M_n}/_x\}\}$ when $((E, \mathcal{P}), (E, \mathcal{P}')) \in \mathsf{gen}(P_0\{^{D_1=D_2}/_x\}, \mathsf{if}\ D_1 = D_2\ \mathsf{then}\ P_0\{^{\mathsf{true}}/_x\}\ \mathsf{else}\ P_0\{^{\mathsf{false}}/_x\})$ and for all $i \leq n$, $\sigma_i(D_1 = D_2) \Downarrow M_i$, and showing that $\mathcal{R} \cup \mathcal{R}^{-1}$ is symmetric and satisfies the three conditions of Definition 3.6. The main argument is that $\sigma_i(\mathsf{if}\ D_1 = D_2\ \mathsf{then}\ P_0\{^{\mathsf{true}}/_x\}\ \mathsf{else}\ P_0\{^{\mathsf{false}}/_x\})$ reduces to $\sigma_i P_0\{^{M_i}/_x\}$ when $\sigma_i(D_1 = D_2) \Downarrow M_i$, and that this process behaves like $\sigma_i P_0\{^{D_1=D_2}/_x\}$.

P6. We have $E, \mathcal{P} \cup \{C[\mathsf{let}\ x : T = M\ \mathsf{in}\ P\ \mathsf{else}\ Q]\} \approx E, \mathcal{P} \cup \{C[P\{^M/_x\}]\}$. This property is proved by defining a relation $\mathcal{R} = \mathsf{gen}(\mathsf{let}\ x : T = M\ \mathsf{in}\ P\ \mathsf{else}\ Q, P\{^M/_x\})$ and showing that $\mathcal{R} \cup \mathcal{R}^{-1}$ is symmetric and satisfies the three conditions of Definition 3.6. The main argument is that $\mathsf{let}\ x : T = M\ \mathsf{in}\ P\ \mathsf{else}\ Q$ reduces to $P\{^M/_x\}$.

P7. We have $\mathcal{C} \approx [\mathcal{C}]_{\mathsf{PV}}$ by P2-P6.

We define a relation $\mathcal{R}$ between valid configurations by $\mathcal{C}\ \mathcal{R}\ \mathcal{C}'$ if and only if $\mathcal{C} \approx \mathcal{C}_\mathsf{e}$, $[\![\mathcal{C}_\mathsf{e}]\!] \mathrel{\overset{\diamond}{\approx}} [\![\mathcal{C}'_\mathsf{e}]\!]$, and $\mathcal{C}'_\mathsf{e} \approx \mathcal{C}'$ for some $\mathcal{C}_\mathsf{e}$ and $\mathcal{C}'_\mathsf{e}$. The relation $\mathcal{R}$ is symmetric. Let us show that it satisfies the three conditions of Definition 3.6:

1. If $\mathcal{C}\ \mathcal{R}\ \mathcal{C}'$ and $\mathcal{C} \downarrow_a$, then $\mathcal{C}_\mathsf{e} \to^* \downarrow_a$, so by Lemma B.6, $[\![\mathcal{C}_\mathsf{e}]\!] \Downarrow^\diamond_a$, so $[\![\mathcal{C}'_\mathsf{e}]\!] \Downarrow^\diamond_a$, so by Lemma B.6, $\mathcal{C}'_\mathsf{e} \to^* \downarrow_a$, so $\mathcal{C}' \to^* \downarrow_a$.

2. If $\mathcal{C}\ \mathcal{R}\ \mathcal{C}'$ and $\mathcal{C} \to \mathcal{C}_1$, then $\mathcal{C}_\mathsf{e} \to^* \mathcal{C}_{1\mathsf{e}}$ and $\mathcal{C}_1 \approx \mathcal{C}_{1\mathsf{e}}$ for some $\mathcal{C}_{1\mathsf{e}}$. By Proposition 4.1, $[\![\mathcal{C}_\mathsf{e}]\!] \to_\diamond{}^* \mathrel{\overset{\diamond}{\equiv}} [\![\mathcal{C}_{1\mathsf{e}}]\!]$.

   If no reduction $\to_\diamond$ is performed, then $[\![\mathcal{C}_{1\mathsf{e}}]\!] \mathrel{\overset{\diamond}{\equiv}} [\![\mathcal{C}_\mathsf{e}]\!] \mathrel{\overset{\diamond}{\approx}} [\![\mathcal{C}'_\mathsf{e}]\!]$ so by Lemma B.8, $[\![\mathcal{C}_{1\mathsf{e}}]\!] \mathrel{\overset{\diamond}{\approx}} [\![\mathcal{C}'_\mathsf{e}]\!]$, so $\mathcal{C}_1\ \mathcal{R}\ \mathcal{C}'_1$ and $\mathcal{C}' \to^* \mathcal{C}'_1$ with $\mathcal{C}'_1 = \mathcal{C}'$.

   If at least one reduction $\to_\diamond$ is performed, then $[\![\mathcal{C}_\mathsf{e}]\!] \to_\diamond{}^* [\![\mathcal{C}_{1\mathsf{e}}]\!]$, so $[\![\mathcal{C}'_\mathsf{e}]\!] \to_\diamond{}^* Q'$ and $[\![\mathcal{C}_{1\mathsf{e}}]\!] \mathrel{\overset{\diamond}{\approx}} Q'$ for some Q'. By Proposition 4.1,

$\mathcal{C}'_{\mathsf{e}} \to^* \mathcal{C}'_{1\mathsf{e}}$ and $Q' \overset{\diamond}{\equiv} [\![\mathcal{C}'_{1\mathsf{e}}]\!]$ for some $\mathcal{C}'_{1\mathsf{e}}$, so by Lemma B.8, $[\![\mathcal{C}_{1\mathsf{e}}]\!] \overset{\diamond}{\approx} [\![\mathcal{C}'_{1\mathsf{e}}]\!]$. Moreover, since $\mathcal{C}'_{\mathsf{e}} \to^* \mathcal{C}'_{1\mathsf{e}}$ and $\mathcal{C}'_{\mathsf{e}} \approx \mathcal{C}'$, we have $\mathcal{C}' \to^* \mathcal{C}'_1$ and $\mathcal{C}'_{1\mathsf{e}} \approx \mathcal{C}'_1$ for some $\mathcal{C}'_1$. So $\mathcal{C}_1 \mathcal{R} \mathcal{C}'_1$.

3. If $\mathcal{C} \mathcal{R} \mathcal{C}'$ and C is an adversarial context, then by P7, $C[\mathcal{C}] \approx C[\mathcal{C}_{\mathsf{e}}] \approx C'[\mathcal{C}_{\mathsf{e}}]$ and $C[\mathcal{C}'] \approx C[\mathcal{C}'_{\mathsf{e}}] \approx C'[\mathcal{C}'_{\mathsf{e}}]$, where $C' = [C]_{\mathsf{PV}}$. By Lemma B.7 and since $[\![C']\!]$ is a closed evaluation context, $[\![C'[\mathcal{C}_{\mathsf{e}}]]\!] \overset{\diamond}{\equiv} [\![C']\!][[\![\mathcal{C}_{\mathsf{e}}]]\!] \overset{\diamond}{\approx} [\![C']\!][[\![\mathcal{C}'_{\mathsf{e}}]]\!] \overset{\diamond}{\equiv} [\![C'[\mathcal{C}'_{\mathsf{e}}]]\!]$ so by Lemma B.8, $[\![C'[\mathcal{C}_{\mathsf{e}}]]\!] \overset{\diamond}{\approx} [\![C'[\mathcal{C}'_{\mathsf{e}}]]\!]$, so $C[\mathcal{C}] \mathcal{R} C[\mathcal{C}']$.

Since $\approx$ is the largest such relation, we have $\mathcal{R} \subseteq \approx$. If $[\![\mathcal{C}]\!] \overset{\diamond}{\approx} [\![\mathcal{C}']\!]$, then $\mathcal{C} \mathcal{R} \mathcal{C}'$, so $\mathcal{C} \approx \mathcal{C}'$. □

B.3 Relating definitions of observational equivalence

Let $\approx_\pi$ be observational equivalence as defined in (Abadi *et al.*, 2016, Definition 4.1). Let us show that, for plain processes, $\approx_\pi$ is equal to $\overset{\diamond}{\approx}$, defined in Definition 4.1.

Lemma B.9. If $A \overset{\circ}{\equiv} B$ and $dom(A) = \emptyset$, then $dom(B) = \emptyset$ and $A \overset{\diamond}{\equiv} B$.
If $A \to_\circ B$ and $dom(A) = \emptyset$, then $dom(B) = \emptyset$ and $A \to_\diamond B$.

Proof. When $dom(\sigma) = \emptyset$, that is, $\sigma = \mathbf{0}$, the rules of $\overset{\circ}{\equiv}$ reduce to:

Plain″	$\nu\tilde{n}.(\mathbf{0} \mid P)$	$\overset{\circ}{\equiv}$	$\nu\tilde{n}.(\mathbf{0} \mid P')$	when $P \overset{\diamond}{\equiv} P'$
New-C″	$\nu\tilde{n}.(\mathbf{0} \mid P)$	$\overset{\circ}{\equiv}$	$\nu\tilde{n}'.(\mathbf{0} \mid P)$	when $\tilde{n}'$ is a reordering of $\tilde{n}$
New-Par″	$\nu\tilde{n}.(\mathbf{0} \mid \nu n'.P)$	$\overset{\circ}{\equiv}$	$\nu\tilde{n}, n'.(\mathbf{0} \mid P)$	
Rewrite″	$\nu\tilde{n}.(\mathbf{0} \mid P)$	$\overset{\circ}{\equiv}$	$\nu\tilde{n}.(\mathbf{0} \mid P)$	

so in all cases, if $A \overset{\circ}{\equiv} B$ and $dom(A) = \emptyset$, then $dom(B) = \emptyset$ and $A \overset{\diamond}{\equiv} B$. Hence we have the first property.

Suppose $A \to_\circ B$ and $dom(A) = \emptyset$. We have $A \overset{\circ}{\equiv} \nu\tilde{n}.(\sigma \mid P)$, $P \to_\diamond P'$ and $\nu\tilde{n}.(\sigma \mid P') \overset{\circ}{\equiv} B$. By the first property, $\sigma = \mathbf{0}$ and $A \overset{\diamond}{\equiv} \nu\tilde{n}.(\mathbf{0} \mid P) \to_\diamond \nu\tilde{n}.(\mathbf{0} \mid P') \overset{\diamond}{\equiv} B$, so $A \to_\diamond B$. □

Lemma B.10. Let P be a plain process. If $P \equiv A$, then $\mathrm{pnf}(A)$ is a plain process and $P \overset{\diamond}{\equiv} \mathrm{pnf}(A)$. If $P \to A$, then $\mathrm{pnf}(A)$ is a plain process and $P \to_\diamond \mathrm{pnf}(A)$.

Proof. Suppose $P \equiv A$. By (Abadi *et al.*, 2016, Lemma B.5), $\mathrm{pnf}(P) \overset{\circ}{\equiv} \mathrm{pnf}(A)$. We have $\mathrm{pnf}(P) = \mathbf{0} \mid P \overset{\diamond}{\equiv} P$ and $dom(\mathrm{pnf}(P)) = \emptyset$. By Lemma B.9, we conclude that $dom(\mathrm{pnf}(A)) = \emptyset$ and $\mathrm{pnf}(P) \overset{\diamond}{\equiv} \mathrm{pnf}(A)$, so $\mathrm{pnf}(A)$ is a plain process and $P \overset{\diamond}{\equiv} \mathrm{pnf}(A)$.

Suppose $P \to A$. By (Abadi *et al.*, 2016, Lemma B.8), $\mathrm{pnf}(P) \to_\circ \mathrm{pnf}(A)$. We have $\mathrm{pnf}(P) = \mathbf{0} \mid P \overset{\diamond}{\equiv} P$ and $dom(\mathrm{pnf}(P)) = \emptyset$. By Lemma B.9, we conclude that $dom(\mathrm{pnf}(A)) = \emptyset$ and $\mathrm{pnf}(P) \to_\diamond \mathrm{pnf}(A)$, so $\mathrm{pnf}(A)$ is a plain process and $P \to_\diamond \mathrm{pnf}(A)$. □

Lemma B.11. Let P and Q be closed processes and σ be a bijective renaming. If $P \overset{\diamond}{\equiv} Q$, then $\sigma P \overset{\diamond}{\equiv} \sigma Q$. If $P \to_\diamond Q$, then $\sigma P \to_\diamond \sigma Q$.

Proof. By induction on the derivations. □

Lemma B.12. Let P and Q be closed processes and σ be a bijective renaming. If $P \overset{\diamond}{\approx} Q$, then $\sigma P \overset{\diamond}{\approx} \sigma Q$.

Proof. We define a relation $\mathcal{R}$ by $P \mathcal{R} Q$ if and only if $\sigma^{-1} P \overset{\diamond}{\approx} \sigma^{-1} Q$. The relation $\mathcal{R}$ is symmetric. Let us show that it satisfies the three properties of Definition 4.1:

1. If $P \mathcal{R} Q$ and $P \Downarrow_a^\diamond$, then by Lemma B.11, $\sigma^{-1} P \Downarrow_{\sigma^{-1} a}^\diamond$, so we have $\sigma^{-1} Q \Downarrow_{\sigma^{-1} a}^\diamond$, so by Lemma B.11, $Q \Downarrow_a^\diamond$.

2. If $P \mathcal{R} Q$, $P \to_\diamond^* P'$, and P' is closed, then by Lemma B.11, $\sigma^{-1} P \to_\diamond^* \sigma^{-1} P'$ and $\sigma^{-1} P'$ is closed, so $\sigma^{-1} Q \to_\diamond^* Q_1$ and $\sigma^{-1} P' \overset{\diamond}{\approx} Q_1$ for some Q_1. Let $Q' = \sigma Q_1$. By Lemma B.11, we have $Q \to_\diamond^* \sigma Q_1 = Q'$ and since $\sigma^{-1} P' \overset{\diamond}{\approx} \sigma^{-1} Q'$, we have $P' \mathcal{R} Q'$.

3. If $P \mathcal{R} Q$ and $C[\_]$ is a closed evaluation context, then $\sigma^{-1} P \overset{\diamond}{\approx} \sigma^{-1} Q$ so $\sigma^{-1}(C[P]) = \sigma^{-1} C[\sigma^{-1} P] \overset{\diamond}{\approx} \sigma^{-1} C[\sigma^{-1} Q] = \sigma^{-1}(C[Q])$, so $C[P] \mathcal{R} C[Q]$. (The application of σ^{-1} to $C[\_]$ is defined by renaming both free and bound names of $C[\_]$ by σ^{-1}.)

Hence $\mathcal{R} \subseteq \overset{\diamond}{\approx}$. If $P \overset{\diamond}{\approx} Q$, then $\sigma P \mathcal{R} \sigma Q$, so $\sigma P \overset{\diamond}{\approx} \sigma Q$. □

Proposition B.1. Let P and Q be plain processes. We have $P \overset{\diamond}{\approx} Q$ if and only if $P \approx_\pi Q$.

Proof. The relation $\approx_\pi$ restricted to closed plain processes is symmetric. Let us show that it satisfies the three conditions of Definition 4.1:

1. If $P \approx_\pi Q$ and $P \Downarrow_a^\diamond$, then by (Abadi *et al.*, 2016, Lemmas B.9 and B.7), $P \Downarrow_a$, so $Q \Downarrow_a$, that is, $Q \to^* \equiv C[\overline{a}\langle M\rangle.Q']$ for some M, Q', and evaluation context $C[\_]$ that does not bind a, so by Lemma B.10, $Q \to_\diamond{}^* \overset{\diamond}{\equiv} \mathrm{pnf}(C[\overline{a}\langle M\rangle.Q']) = C'[\overline{a}\langle M'\rangle.Q'']$ for some M', Q'', and plain evaluation context $C'[\_]$ that does not bind a, so $Q \Downarrow_a^\diamond$.

2. If $P \approx_\pi Q$, $P \to_\diamond{}^* P'$, and P' is closed, then by (Abadi *et al.*, 2016, Lemma B.9), $P \to^* P'$, so $Q \to^* B'$ and $P' \approx_\pi B'$ for some B'. By Lemma B.10, $\mathrm{pnf}(B')$ is a plain process and $Q \to_\diamond{}^* \mathrm{pnf}(B')$. By (Abadi *et al.*, 2016, Lemma B.2), $\mathrm{pnf}(B') \equiv B'$, so $P' \approx_\pi \mathrm{pnf}(B')$. So we have $Q \to_\diamond{}^* Q'$ and $P' \approx_\pi Q'$ for $Q' = \mathrm{pnf}(B')$.

3. If $P \approx_\pi Q$, then $C[P] \approx_\pi C[Q]$ for all plain closed evaluation contexts $C[\_]$, since these contexts are closing for P and Q.

Since $\overset{\diamond}{\approx}$ is the largest such relation, we conclude that, if $P \approx_\pi Q$, then $P \overset{\diamond}{\approx} Q$.

Let us define the relation $\mathcal{R}$ by $A \mathcal{R} B$ if and only if $A \equiv \nu\widetilde{n}.(\sigma \mid P)$, $B \equiv \nu\widetilde{n}.(\sigma \mid Q)$, and $P \overset{\diamond}{\approx} Q$, for some $\widetilde{n}$, closed substitution σ, and closed processes P and Q. The relation $\mathcal{R}$ is symmetric. Let us show that it satisfies the three conditions of (Abadi *et al.*, 2016, Definition 4.1):

1. If $A \mathcal{R} B$ and $A \Downarrow_a$, then $\nu\widetilde{n}.(\sigma \mid P) \to^* \equiv C[\overline{a}\langle M\rangle.P']$ for some M, P', and evaluation context $C[\_]$ that does not bind a. By (Abadi *et al.*, 2016, Lemma B.8), $\nu\widetilde{n}.(\sigma \mid P) \to_\circ{}^* \equiv C[\overline{a}\langle M\rangle.P']$, and by (Abadi *et al.*, 2016, Lemma B.23), $P \to_\diamond{}^* P_1$ and $C[\overline{a}\langle M\rangle.P'] \equiv \nu\widetilde{n}.(\sigma \mid P_1)$ for some P_1. Let $dom(\sigma) = \{\widetilde{x}\}$. We have $\nu\widetilde{n}.P_1 \equiv \nu\widetilde{x},\widetilde{n}.(\sigma \mid P_1) \equiv \nu\widetilde{x}.C[\overline{a}\langle M\rangle.P']$, so by Lemma B.10, $\nu\widetilde{n}.P_1 \overset{\diamond}{\equiv} \mathrm{pnf}(\nu\widetilde{x}.C[\overline{a}\langle M\rangle.P']) = C'[\overline{a}\langle M'\rangle.P'']$ for some M', P'', and plain evaluation context $C'[\_]$ that does not bind a. We rename $\widetilde{n}$ so that $a \notin \{\widetilde{n}\}$. By Lemma B.4, $\nu\widetilde{n}.P_1 = C_![\overline{N''}\langle M''\rangle.P''']$ and $N'' =_\mathcal{E} a$ for some N'', M'', P''', and replication context $C_![\_]$ that does not bind a. So $P_1 = C'_![\overline{N''}\langle M''\rangle.P''']$ for some replication context $C'_![\_]$ that does not bind a, so $P_1 \overset{\diamond}{\equiv} C''[\overline{a}\langle M''\rangle.P''']$ for

some plain evaluation context $C''[\_]$ that does not bind a, by Lemma B.4. Hence $P \Downarrow_a^\diamond$. Since $P \overset{\diamond}{\approx} Q$, we have $Q \Downarrow_a^\diamond$, that is, $Q \to_\diamond^* \overset{\diamond}{\equiv} C_1[\overline{a}\langle M_1\rangle.Q_1]$ for some M_1, Q_1, and plain evaluation context $C_1[\_]$ that does not bind a. Hence $B \equiv \nu\widetilde{n}.(\sigma \mid Q) \to^* \equiv \nu\widetilde{n}.(\sigma | C_1[\overline{a}\langle M_1\rangle.Q_1])$. Since $a \notin \{\widetilde{n}\}$, $\nu\widetilde{n}.(\sigma | C_1[\_])$ is an evaluation context that does not bind a, so $B \Downarrow_a$.

2. If $A \mathcal{R} B$, $A \to^* A'$, and A' is closed, then $\nu\widetilde{n}.(\sigma \mid P) \equiv \to^* A'$. If $A' = A$, then we have $A' \mathcal{R} B'$ and $B \to^* B'$ for $B' = B$. Otherwise, $\nu\widetilde{n}.(\sigma \mid P) \to^+ A'$, so by (Abadi *et al.*, 2016, Lemma B.8), $\nu\widetilde{n}.(\sigma \mid P) \to_\circ{}^+ \mathrm{pnf}(A')$, and by (Abadi *et al.*, 2016, Lemma B.23), $P \to_\diamond{}^+ P'$ and $\mathrm{pnf}(A') \equiv \nu\widetilde{n}.(\sigma \mid P')$ for some P'. Since $P \overset{\diamond}{\approx} Q$, we have $Q \to_\diamond{}^* Q'$ and $P' \overset{\diamond}{\approx} Q'$ for some Q'. Let $B' = \nu\widetilde{n}.(\sigma \mid Q')$. We have $B \to^* B'$. Moreover, $A' \equiv \mathrm{pnf}(A') \equiv \nu\widetilde{n}.(\sigma \mid P')$, $B' = \nu\widetilde{n}.(\sigma \mid Q')$, and $P' \overset{\diamond}{\approx} Q'$, so $A' \mathcal{R} B'$.

3. Suppose $A \mathcal{R} B$ and $C[\_]$ is a closing evaluation context for A and B. We want to show that $C[A] \mathcal{R} C[B]$. We first rename the free names and variables of $C[\_]$, so that the obtained context is simple. By (Abadi *et al.*, 2016, Lemma C.11), it is sufficient to show $C[A] \mathcal{R} C[B]$ for the renamed context. By (Abadi *et al.*, 2016, Lemma A.1), there exists a context $C'[\_]$ of the form $\nu\widetilde{a}, \widetilde{x}.(\_ \mid A')$ such that $C[\_] \equiv C'[\_]$ and all subcontexts of $C'[\_]$ are simple for A. Let $\mathrm{pnf}(A') = \nu\widetilde{n}'.(\sigma' \mid P')$. We have $C[A] \equiv C'[A] \equiv \nu\widetilde{a}, \widetilde{x}.(A \mid A') \equiv \nu\widetilde{a}, \widetilde{x}.(\nu\widetilde{n}.(\sigma \mid P) \mid \nu\widetilde{n}'.(\sigma' \mid P'))$. We rename $\widetilde{n}$ and $\widetilde{n}'$ so that $\{\widetilde{n}\} \cap \{\widetilde{n}'\} = \emptyset$, the names in $\widetilde{n}$ do not occur in σ' and P', and the names in $\widetilde{n}'$ do not occur in σ and P. By Lemma B.12, the property $P \overset{\diamond}{\approx} Q$ is preserved by the renaming of $\widetilde{n}$. We obtain $C[A] \equiv \nu\widetilde{a}, \widetilde{n}, \widetilde{n}'.((\sigma \mid \sigma\sigma')_{dom(\sigma)\cup dom(\sigma')\setminus\{\widetilde{x}\}} \mid P \mid \sigma P')$, where $(\sigma \mid \sigma\sigma')_{dom(\sigma)\cup dom(\sigma')\setminus\{\widetilde{x}\}}$ and $\sigma P'$ are closed. (The substitution σ' does not affect σ nor P because they are closed.) Similarly, $C[B] \equiv \nu\widetilde{a}, \widetilde{n}, \widetilde{n}'.((\sigma \mid \sigma\sigma')_{dom(\sigma)\cup dom(\sigma')\setminus\{\widetilde{x}\}} \mid Q \mid \sigma P')$. Since $P \overset{\diamond}{\approx} Q$ and $\_ | \sigma P'$ is a closed evaluation context, we have $P | \sigma P' \overset{\diamond}{\approx} Q | \sigma P'$. Therefore, $C[A] \mathcal{R} C[B]$.

Since $\approx_\pi$ is the largest such relation, $\mathcal{R} \subseteq \approx_\pi$. In particular, if $P \overset{\diamond}{\approx} Q$, then $P \mathcal{R} Q$, so $P \approx_\pi Q$. □

References

Abadi, M. 1999. "Secrecy by Typing in Security Protocols". *Journal of the ACM*. 46(5): 749–786.

Abadi, M. and B. Blanchet. 2005a. "Analyzing Security Protocols with Secrecy Types and Logic Programs". *Journal of the ACM*. 52(1): 102–146.

Abadi, M. and B. Blanchet. 2005b. "Computer-Assisted Verification of a Protocol for Certified Email". *Science of Computer Programming*. 58(1–2): 3–27. Special issue SAS'03.

Abadi, M., B. Blanchet, and C. Fournet. 2007. "Just Fast Keying in the Pi Calculus". *ACM Transactions on Information and System Security (TISSEC)*. 10(3): 1–59.

Abadi, M., B. Blanchet, and C. Fournet. 2016. "The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication". Report arXiv:1609.03003v1. Available at http://arxiv.org/abs/1609.03003v1.

Abadi, M. and V. Cortier. 2006. "Deciding Knowledge in Security Protocols under Equational Theories". *Theoretical Computer Science*. 367(1–2): 2–32.

Abadi, M. and C. Fournet. 2001. "Mobile Values, New Names, and Secure Communication". In: *28th ACM Symposium on Principles of Programming Languages (POPL'01)*. London, UK: ACM. 104–115.

Abadi, M. and C. Fournet. 2004. "Private authentication". *Theoretical Computer Science*. 322(3): 427–476.

Abadi, M., N. Glew, B. Horne, and B. Pinkas. 2002. "Certified Email with a Light On-line Trusted Third Party: Design and Implementation". In: *11th International World Wide Web Conference*. Honolulu, Hawaii: ACM. 387–395.

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

Abadi, M. and A. D. Gordon. 1999. "A Calculus for Cryptographic Protocols: The Spi Calculus". *Information and Computation*. 148(1): 1–70. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.

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

Abadi, M. and P. Rogaway. 2002. "Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)". *Journal of Cryptology*. 15(2): 103–127.

Abdalla, M., P.-A. Fouque, and D. Pointcheval. 2005. "Password-Based Authenticated Key Exchange in the Three-Party Setting". In: *2005 International Workshop on Practice and Theory in Public Key Cryptography (PKC'05)*. Ed. by S. Vaudenay. Vol. 3386. *Lecture Notes in Computer Science*. Les Diablerets, Switzerland: Springer. 65–84.

Adrian, D., K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. 2015. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice". In: *22nd ACM Conference on Computer and Communications Security*.

Aiello, W., S. M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, K. Keromytis, and O. Reingold. 2004. "Just Fast Keying: Key Agreement in a Hostile Internet". *ACM Transactions on Information and System Security*. 7(2): 242–273.

Aizatulin, M., A. D. Gordon, and J. Jürjens. 2011. "Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution". In: *18th ACM Conference on Computer and Communications Security (CCS'11)*. Chicago, IL, USA: ACM. 331–340.

Aizatulin, M., A. D. Gordon, and J. Jürjens. 2012. "Computational Verification of C Protocol Implementations by Symbolic Execution". In: *19th ACM Conference on Computer and Communications Security (CCS'12)*. Raleigh, NC, USA: ACM. 712–723.

Allamigeon, X. and B. Blanchet. 2005. "Reconstruction of Attacks against Cryptographic Protocols". In: *18th IEEE Computer Security Foundations Workshop (CSFW-18)*. Aix-en-Provence, France: IEEE. 140–154.

Almeida, J. B., M. Barbosa, G. Barthe, and F. Dupressoir. 2013. "Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations". In: *ACM Conference on Computer and Communications Security (CCS'13)*. Berlin, Germany: ACM. 1217–1230.

Arapinis, M. and M. Duflot. 2007. "Bounding Messages for Free in Security Protocols". In: *27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*. Ed. by V. Arvind and S. Prasad. Vol. 4855. *Lecture Notes in Computer Science*. New Delhi, India: Springer. 376–387.

Arapinis, M., J. Liu, E. Ritter, and M. Ryan. 2014. "Stateful Applied Pi Calculus". In: *Principles of Security and Trust—Third International Conference*. Ed. by M. Abadi and S. Kremer. Vol. 8414. *Lecture Notes in Computer Science*. Springer. 22–41.

Arapinis, M., E. Ritter, and M. D. Ryan. 2011. "StatVerif: Verification of stateful processes". In: *24th Computer Security Foundations Symposium (CSF'11)*. IEEE. Cernay-la-Ville, France. 33–47.

Armando, A., D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganó, and L. Vigneron. 2005. "The AVISPA tool for Automated Validation of Internet Security Protocols and Applications". In: *Computer Aided Verification, 17th International Conference, CAV 2005*. Ed. by K. Etessami and S. K. Rajamani. Vol. 3576. *Lecture Notes in Computer Science*. Edinburgh, Scotland: Springer. 281–285.

Armando, A., R. Carbone, and L. Compagna. 2014. "SATMC: a SAT-based Model Checker for Security-critical Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems, 20th International Conference, TACAS 2014*. Ed. by E. Ábrahám and K. Havelund. Vol. 8413. *Lecture Notes in Computer Science*. Grenoble, France: Springer. 31–45. DOI: 10.1007/978-3-642-54862-8_3.

Avalle, M., A. Pironti, R. Sisto, and D. Pozza. 2011. "The JavaSPI Framework for Security Protocol Implementation". In: *International Conference on Availability, Reliability and Security (ARES'11)*. IEEE. 746–751.

Bachmair, L. and H. Ganzinger. 2001. "Resolution Theorem Proving". In: *Handbook of Automated Reasoning*. Ed. by A. Robinson and A. Voronkov. Vol. 1. North Holland. Chap. 2. 19–100.

Backes, M., F. Bendun, M. Maffei, E. Mohammadi, and K. Pecina. 2015. "Symbolic Malleable Zero-Knowledge Proofs". In: *28th IEEE Computer Security Foundations Symposium (CSF'15)*. Verona, Italy: IEEE. 412–480.

Backes, M., C. Hritcu, and M. Maffei. 2008a. "Automated Verification of Remote Electronic Voting Protocols in the Applied Pi-calculus". In: *21st IEEE Computer Security Foundations Symposium (CSF'08)*. Pittsburgh, PA: IEEE Computer Society. 195–209.

Backes, M., M. Maffei, and D. Unruh. 2008b. "Zero-Knowledge in the Applied Pi-calculus and Automated Verification of the Direct Anonymous Attestation Protocol". In: *29th IEEE Symposium on Security and Privacy*. Technical report version available at http://eprint.iacr.org/2007/289. IEEE. Oakland, CA. 202–215.

Backes, M., E. Mohammadi, and T. Ruffing. 2014. "Computational Soundness Results for ProVerif: Bridging the Gap from Trace Properties to Uniformity". In: *Principles of Security and Trust (POST'14)*. Ed. by M. Abadi and S. Kremer. Vol. 8414. *Lecture Notes in Computer Science*. Grenoble, France: Springer. 42–62.

Bansal, C., K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. 2013. "Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage". In: *Principles of Security and Trust (POST 2013)*. Ed. by D. Basin and J. Mitchell. Vol. 7796. *Lecture Notes in Computer Science*. Rome, Italy: Springer. 126–146.

Bansal, C., K. Bhargavan, and S. Maffeis. 2012. "Discovering Concrete Attacks on Website Authorization by Formal Analysis". In: *25th IEEE Computer Security Foundations Symposium (CSF'12)*. IEEE. Cambridge, MA, USA. 247–262.

Barthe, G., F. Dupressoir, P.-A. Fouque, B. Grégoire, M. Tibouchi, and J.-C. Zapalowicz. 2014a. "Making RSA-PSS Provably Secure against Non-random Faults". In: *Cryptographic Hardware and Embedded Systems (CHES'14)*. Ed. by L. Batina and M. Robshaw. Vol. 8731. *Lecture Notes in Computer Science*. Busan, South Korea: Springer. 206–222.

Barthe, G., F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub. 2014b. "EasyCrypt: A Tutorial". In: *Foundations of Security Analysis and Design VII*. Ed. by A. Aldini, J. Lopez, and F. Martinelli. Vol. 8604. *Lecture Notes in Computer Science*. Springer. 146–166.

Barthe, G., B. Grégoire, S. Heraud, and S. Z. Béguelin. 2011. "Computer-Aided Security Proofs for the Working Cryptographer". In: *Advances in Cryptology – CRYPTO 2011*. Ed. by P. Rogaway. Vol. 6841. *Lecture Notes in Computer Science*. Santa Barbara, CA, USA: Springer. 71–90.

Barthe, G., B. Grégoire, and S. Zanella. 2009. "Formal Certification of Code-Based Cryptographic Proofs". In: *36th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'09)*. Savannah, Georgia: ACM. 90–101.

Basin, D., J. Dreier, and R. Casse. 2015. "Automated Symbolic Proofs of Observational Equivalence". In: *22nd ACM Conference on Computer and Communications Security (CCS'15)*. Denver, CO: ACM. 1144–1155.

Basin, D., S. Mödersheim, and L. Viganò. 2005. "OFMC: A symbolic model checker for security protocols". *International Journal of Information Security*. 4(3): 181–208.

Baudet, M. 2007. "Sécurité des protocoles cryptographiques: aspects logiques et calculatoires". *PhD thesis*. Ecole Normale Supérieure de Cachan.

Béguelin, S. Z., B. Grégoire, G. Barthe, and F. Olmedo. 2009. "Formally Certifying the Security of Digital Signature Schemes". In: *30th IEEE Symposium on Security and Privacy, S&P 2009*. Oakland, CA: IEEE. 237–250.

Bellovin, S. M. and M. Merritt. 1992. "Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks". In: *1992 IEEE Computer Society Symposium on Research in Security and Privacy*. 72–84.

Bellovin, S. M. and M. Merritt. 1993. "Augmented Encrypted Key Exchange: a Password-Based Protocol Secure Against Dictionary Attacks and Password File Compromise". In: *First ACM Conference on Computer and Communications Security*. 244–250.

Bengtson, J., K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. 2011. "Refinement Types for Secure Implementations". *ACM Transactions on Programming Languages and Systems*. 33(2).

Beurdouche, B., K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. 2015. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *IEEE Symposium on Security & Privacy 2015 (Oakland'15)*. IEEE.

Bhargavan, K., R. Corin, and C. Fournet. 2007. "Crypto-Verifying Protocol Implementations in ML". http://doc.utwente.nl/64107/1/fs2cv.pdf.

Bhargavan, K., R. Corin, C. Fournet, and E. Zălinescu. 2008. "Cryptographically Verified Implementations for TLS". In: *15th ACM Conference on Computer and Communications Security (CCS'08)*. ACM. 459–468.

Bhargavan, K., C. Fournet, and A. Gordon. 2004. "Verifying Policy-Based Security for Web Services". In: *ACM Conference on Computer and Communications Security (CCS'04)*. Washington DC: ACM. 268–277.

Bhargavan, K., C. Fournet, and A. Gordon. 2010. "Modular Verification of Security Protocol Code by Typing". In: *ACM Symposium on Principles of Programming Languages (POPL'10)*. Madrid, Spain: ACM. 445–456.

Bhargavan, K., C. Fournet, A. Gordon, and S. Tse. 2006. "Verified interoperable implementations of security protocols". In: *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. Venice, Italy: IEEE Computer Society. 139–152.

Bhargavan, K., C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. 2013. "Implementing TLS with Verified Cryptographic Security". In: *IEEE Symposium on Security & Privacy*. 445–462.

Blanchet, B. 2004. "Automatic Proof of Strong Secrecy for Security Protocols". In: *IEEE Symposium on Security and Privacy*. Oakland, California. 86–100.

Blanchet, B. 2008a. "A Computationally Sound Mechanized Prover for Security Protocols". *IEEE Transactions on Dependable and Secure Computing*. 5(4): 193–207.

Blanchet, B. 2008b. "Vérification automatique de protocoles cryptographiques : modèle formel et modèle calculatoire". *Mémoire d'habilitation à diriger des recherches*. Université Paris-Dauphine.

Blanchet, B. 2009. "Automatic Verification of Correspondences for Security Protocols". *Journal of Computer Security*. 17(4): 363–434.

Blanchet, B. 2011. "Using Horn Clauses for Analyzing Security Protocols". In: *Formal Models and Techniques for Analyzing Security Protocols*. Ed. by V. Cortier and S. Kremer. Vol. 5. *Cryptology and Information Security Series*. IOS Press. 86–111.

Blanchet, B. 2012a. "Mechanizing Game-Based Proofs of Security Protocols". In: *Software Safety and Security - Tools for Analysis and Verification*. Ed. by T. Nipkow, O. Grumberg, and B. Hauptmann. Vol. 33. *NATO Science for Peace and Security Series – D: Information and Communication Security*. Proceedings of the 2011 MOD summer school. IOS Press. 1–25.

Blanchet, B. 2012b. "Security Protocol Verification: Symbolic and Computational Models". In: *First Conference on Principles of Security and Trust (POST'12)*. Ed. by P. Degano and J. Guttman. Vol. 7215. *Lecture Notes in Computer Science*. Tallinn, Estonia: Springer. 3–29.

Blanchet, B. 2014. "Automatic Verification of Security Protocols in the Symbolic Model: the Verifier ProVerif". In: *Foundations of Security Analysis and Design VII, FOSAD Tutorial Lectures*. Ed. by A. Aldini, J. Lopez, and F. Martinelli. Vol. 8604. *Lecture Notes in Computer Science*. Springer. 54–87.

Blanchet, B., M. Abadi, and C. Fournet. 2008. "Automated Verification of Selected Equivalences for Security Protocols". *Journal of Logic and Algebraic Programming*. 75(1): 3–51.

Blanchet, B. and A. Chaudhuri. 2008. "Automated Formal Analysis of a Protocol for Secure File Sharing on Untrusted Storage". In: *IEEE Symposium on Security and Privacy*. IEEE. Oakland, CA. 417–431.

Blanchet, B. and A. Podelski. 2005. "Verification of Cryptographic Protocols: Tagging Enforces Termination". *Theoretical Computer Science*. 333(1-2): 67–90. Special issue FoSSaCS'03.

Blanchet, B. and B. Smyth. 2016. "Automated reasoning for equivalences in the applied pi calculus with barriers". In: *29th IEEE Computer Security Foundations Symposium (CSF'16)*. Lisboa, Portugal: IEEE. 310–324.

Blanchet, B., B. Smyth, and V. Cheval. 2016. "ProVerif 1.94pl1: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial". Available at http://proverif.inria.fr/manual.pdf.

Boichut, Y., N. Kosmatov, and L. Vigneron. 2006. "Validation of Prouvé protocols using the automatic tool TA4SP". In: *Third Taiwanese-French Conference on Information Technology (TFIT 2006)*. Nancy, France. 467–480.

Bruni, A., S. Mödersheim, F. Nielson, and H. R. Nielson. 2015. "Set-Pi: Set Membership Pi-Calculus". In: *28th IEEE Computer Security Foundations Symposium (CSF'15)*. Verona, Italy: IEEE. 185–198.

Cadé, D. and B. Blanchet. 2013. "From Computationally-Proved Protocol Specifications to Implementations and Application to SSH". *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*. 4(1): 4–31.

Cadé, D. and B. Blanchet. 2015. "Proved Generation of Implementations from Computationally Secure Protocol Specifications". *Journal of Computer Security*. 23(3): 331–402.

Canetti, R. and J. Herzog. 2006. "Universally Composable Symbolic Analysis of Mutual Authentication and Key Exchange Protocols". In: *Proceedings, Theory of Cryptography Conference (TCC'06)*. Ed. by S. Halevi and T. Rabin. Vol. 3876. *Lecture Notes in Computer Science*. Extended version available at http://eprint.iacr.org/2004/334. New York, NY: Springer. 380–403.

Chadha, R., S. Ciobâca, and S. Kremer. 2012. "Automated Verification of Equivalence Properties of Cryptographic Protocols". In: *21st European Symposium on Programming (ESOP'12)*. Vol. 7211. *Lecture Notes in Computer Science*. Springer. 108–127.

Chaki, S. and A. Datta. 2009. "ASPIER: An Automated Framework for Verifying Security Protocol Implementations". In: *22nd IEEE Computer Security Foundations Symposium (CSF'09)*. Port Jefferson, NY, USA. 172–185.

Cheval, V. and B. Blanchet. 2013. "Proving More Observational Equivalences with ProVerif". In: *2nd Conference on Principles of Security and Trust (POST 2013)*. Ed. by D. Basin and J. Mitchell. Vol. 7796. *Lecture Notes in Computer Science*. Rome, Italy: Springer. 226–246.

Cheval, V., H. Comon-Lundh, and S. Delaune. 2011. "Trace Equivalence Decision: Negative Tests and Non-determinism". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. Chicago, Illinois, USA: ACM. 321–330.

Chevalier, Y., R. Küsters, M. Rusinowitch, and M. Turuani. 2003. "Deciding the Security of Protocols with Diffie-Hellman Exponentiation and Products in Exponents". In: *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd Conference*. Ed. by P. K. Pandya and J. Radhakrishnan. Vol. 2914. *Lecture Notes in Computer Science*. Mumbai, India: Springer. 124–135.

Chevalier, Y., R. Küsters, M. Rusinowitch, and M. Turuani. 2005. "An NP decision procedure for protocol insecurity with XOR". *Theoretical Computer Science*. 338(1–3): 247–274.

Chothia, T., B. Smyth, and C. Staite. 2015. "Automatically Checking Commitment Protocols in ProVerif without False Attacks". In: *Principles of Security and Trust, 4th International Conference, POST 2015*. Ed. by R. Focardi and A. Myers. Vol. 9036. *Lecture Notes in Computer Science*. London, UK: Springer. 137–155.

Chrétien, R., V. Cortier, and S. Delaune. 2015a. "Decidability of trace equivalence for protocols with nonces". In: *28th IEEE Computer Security Foundations Symposium (CSF'15)*. Verona, Italy: IEEE Computer Society. 170–184. DOI: 10.1109/CSF.2015.19.

Chrétien, R., V. Cortier, and S. Delaune. 2015b. "From security protocols to pushdown automata". *ACM Transactions on Computational Logic*. 17(1:3). DOI: 10.1145/2811262.

Ciobâcă, Ş. 2011. "Automated Verification of Security Protocols with Applications to Electronic Voting". *PhD thesis*. ENS Cachan.

Cohen, E. 2002. "Proving Protocols Safe from Guessing". In: *Foundations of Computer Security*. Copenhagen, Denmark.

Comon-Lundh, H. and V. Cortier. 2003. "New Decidability Results for Fragments of First-Order Logic and Application to Cryptographic Protocols". In: *14th Int. Conf. Rewriting Techniques and Applications (RTA'2003)*. Ed. by R. Nieuwenhuis. Vol. 2706. *Lecture Notes in Computer Science*. Valencia, Spain: Springer. 148–164.

Comon-Lundh, H. and S. Delaune. 2005. "The finite variant property: How to get rid of some algebraic properties". In: *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA'05)*. Ed. by J. Giesl. Vol. 3467. *Lecture Notes in Computer Science*. Nara, Japan: Springer. 294–307.

Comon-Lundh, H. and V. Shmatikov. 2003. "Intruder deductions, constraint solving and insecurity decision in presence of exclusive or". In: *Symposium on Logic in Computer Science (LICS'03)*. Ottawa, Canada: IEEE Computer Society. 271–280.

Corin, R., J. M. Doumen, and S. Etalle. 2004. "Analysing Password Protocol Security Against Off-line Dictionary Attacks". In: *2nd Int. Workshop on Security Issues with Petri Nets and other Computational Models (WISP). Electronic Notes in Theoretical Computer Science*.

Corin, R., S. Malladi, J. Alves-Foss, and S. Etalle. 2003. "Guess What? Here is a New Tool that Finds some New Guessing Attacks". In: *Workshop on Issues in the Theory of Security (WITS'03)*. Ed. by R. Gorrieri. Warsaw, Poland.

Cortier, V., H. Hördegen, and B. Warinschi. 2006. "Explicit Randomness is not Necessary when Modeling Probabilistic Encryption". In: *Workshop on Information and Computer Security (ICS 2006)*. Ed. by C. Dima, M. Minea, and F. Tiplea. Vol. 186. *Electronic Notes in Theoretical Computer Science*. Timisoara, Romania: Elsevier. 49–65.

Cortier, V., S. Kremer, and B. Warinschi. 2011. "A Survey of Symbolic Methods in Computational Analysis of Cryptographic Systems". *Journal of Automated Reasoning*. 46(3-4): 225–259.

Cortier, V., M. Rusinowitch, and E. Zălinescu. 2007. "Relating two standard notions of secrecy". *Logical Methods in Computer Science*. 3(3).

Cortier, V. and C. Wiedling. 2012. "A formal analysis of the Norwegian E-voting protocol". In: *Proceedings of the 1st International Conference on Principles of Security and Trust (POST'12)*. Ed. by P. Degano and J. D. Guttman. Vol. 7215. *Lecture Notes in Computer Science*. Tallinn, Estonia: Springer. 109–128.

Cremers, C. J. 2008. "Unbounded verification, falsification, and characterization of security protocols by pattern refinement". In: *15th ACM conference on Computer and Communications Security (CCS'08)*. Alexandria, Virginia, USA: ACM. 119–128.

Delaune, S. and F. Jacquemard. 2004. "A Theory of Dictionary Attacks and its Complexity". In: *17th IEEE Computer Security Foundations Workshop*. Pacific Grove, CA: IEEE. 2–15.

Delaune, S., S. Kremer, and M. D. Ryan. 2009. "Verifying Privacy-type Properties of Electronic Voting Protocols". *Journal of Computer Security*. 17(4): 435–487.

Delaune, S., S. Kremer, M. D. Ryan, and G. Steel. 2011. "Formal analysis of protocols based on TPM state registers". In: *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*. Cernay-la-Ville, France: IEEE Computer Society. 66–82.

Delaune, S., M. Ryan, and B. Smyth. 2008. "Automatic verification of privacy properties in the applied pi calculus". In: *Second Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM'08)*. Ed. by Y. Karabulut, J. Mitchell, P. Herrmann, and C. D. Jensen. Vol. 263. *IFIP Advances in Information and Communication Technology*. Trondheim, Norway: Springer. 263–278.

Denning, D. E. and G. M. Sacco. 1981. "Timestamps in Key Distribution Protocols". *Communications of the ACM*. 24(8): 533–536.

Diffie, W. and M. Hellman. 1976. "New Directions in Cryptography". *IEEE Transactions on Information Theory*. IT-22(6): 644–654.

Dolev, D. and A. C. Yao. 1983. "On the Security of Public Key Protocols". *IEEE Transactions on Information Theory*. IT-29(12): 198–208.

Dreier, J., P. Lafourcade, and Y. Lakhnech. 2013. "Formal Verification of e-Auction Protocols". In: *Principles of Security and Trust (POST'13)*. Ed. by D. Basin and J. Mitchell. Vol. 7796. *Lecture Notes in Computer Science*. Rome, Italy: Springer. 247–266.

Drielsma, P. H., S. Mödersheim, and L. Viganò. 2005. "A Formalization of Off-line Guessing for Security Protocol Analysis". In: *Logic for Programming, Artificial Intelligence, and Reasoning: 11th International Conference, LPAR 2004*. Ed. by F. Baader and A. Voronkov. Vol. 3452. *Lecture Notes in Computer Science*. Montevideo, Uruguay: Springer. 363–379.

Dupressoir, F., A. D. Gordon, J. Jürjens, and D. A. Naumann. 2011. "Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols". In: *24th IEEE Symposium on Computer Security Foundations (CSF'11)*. Paris, France: IEEE Computer Society. 3–17.

Durgin, N., P. Lincoln, J. C. Mitchell, and A. Scedrov. 2004. "Multiset Rewriting and the Complexity of Bounded Security Protocols". *Journal of Computer Security*. 12(2): 247–311.

Escobar, S., J. Hendrix, C. Meadows, and J. Meseguer. 2007. "Diffie-Hellman cryptographic reasoning in the Maude-NRL protocol analyzer". In: *Proc. 2nd International Workshop on Security and Rewriting Techniques (SecReT 2007)*.

Escobar, S., D. Kapur, C. Lynch, C. Meadows, J. Meseguer, P. Narendran, and R. Sasse. 2011. "Protocol analysis in Maude-NPA using unification modulo homomorphic encryption". In: *13th international ACM SIGPLAN symposium on Principles and practices of declarative programming (PPDP'11)*. Odense, Denmark: ACM. 65–76.

Escobar, S., C. Meadows, and J. Meseguer. 2006. "A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties". *Theoretical Computer Science*. 367(1-2): 162–202.

Fournet, C. and M. Kohlweiss. 2011. "Modular Cryptographic Verification by Typing". In: *7th Workshop on Formal and Computational Cryptography (FCC'11)*. Paris, France.

Godskesen, J. C. 2006. "Formal Verification of the ARAN Protocol Using the Applied Pi-calculus". In: *Sixth International IFIP WG 1.7 Workshop on Issues in the Theory of Security (WITS'06)*. Vienna, Austria. 99–113.

Gordon, A. and A. Jeffrey. 2004. "Types and Effects for Asymmetric Cryptographic Protocols". *Journal of Computer Security*. 12(3/4): 435–484.

Goubault-Larrecq, J. 2005. "Deciding $\mathcal{H}_1$ by resolution". *Information Processing Letters*. 95(3): 401–408.

Goubault-Larrecq, J. and F. Parrennes. 2005. "Cryptographic Protocol Analysis on Real C Code". In: *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*. Ed. by R. Cousot. Vol. 3385. *Lecture Notes in Computer Science*. Paris, France: Springer. 363–379.

Heather, J., G. Lowe, and S. Schneider. 2000. "How to Prevent Type Flaw Attacks on Security Protocols". In: *13th IEEE Computer Security Foundations Workshop (CSFW-13)*. Cambridge, England. 255–268.

Hüttel, H. 2003. "Deciding Framed Bisimilarity". *Electronic Notes in Theoretical Computer Science*. 68(6): 1–20. Special issue Infinity'02: 4th International Workshop on Verification of Infinite-State Systems.

Kallahalla, M., E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. 2003. "Plutus: Scalable secure file sharing on untrusted storage". In: *2nd Conference on File and Storage Technologies (FAST'03)*. San Francisco, CA: Usenix. 29–42.

Khurana, H. and H.-S. Hahm. 2006. "Certified Mailing Lists". In: *ACM Symposium on Communication, Information, Computer and Communication Security (ASIACCS'06)*. Taipei, Taiwan: ACM. 46–58.

Kowalski, R. 1974. "Predicate Logic as Programming Language". In: *Proceedings IFIP Congress*. Stockholm: North Holland. 569–574.

Kremer, S. and R. Künnemann. 2014. "Automated Analysis of Security Protocols with Global State". In: *35th IEEE Symposium on Security and Privacy (S&P'14)*. San Jose, CA, USA: IEEE Computer Society.

Kremer, S. and M. D. Ryan. 2005. "Analysis of an Electronic Voting Protocol in the Applied Pi Calculus". In: *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005*. Ed. by M. Sagiv. Vol. 3444. *Lecture Notes in Computer Science*. Edimbourg, UK: Springer. 186–200.

Küsters, R. and T. Truderung. 2008. "Reducing protocol analysis with XOR to the XOR-free case in the Horn theory based approach". In: *15th ACM conference on Computer and communications security (CCS'08)*. Alexandria, Virginia, USA: ACM. 129–138.

Küsters, R. and T. Truderung. 2009. "Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation". In: *22nd IEEE Computer Security Foundations Symposium (CSF'09)*. Port Jefferson, New York, USA: IEEE. 157–171.

Lowe, G. 1996. "Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 1055. *Lecture Notes in Computer Science*. Springer. 147–166.

Lowe, G. 1997. "A Hierarchy of Authentication Specifications". In: *10th Computer Security Foundations Workshop (CSFW '97)*. IEEE Computer Society. Rockport, Massachusetts. 31–43.

Lowe, G. 2002. "Analyzing Protocols Subject to Guessing Attacks". In: *Workshop on Issues in the Theory of Security (WITS'02)*. Portland, Oregon.

Lux, K. D., M. J. May, N. L. Bhattad, and C. A. Gunter. 2005. "WSEmail: Secure Internet Messaging Based on Web Services". In: *International Conference on Web Services (ICWS'05)*. Orlando, Florida: IEEE Computer Society. 75–82.

Meadows, C. A. 1996. "The NRL Protocol Analyzer: An Overview". *Journal of Logic Programming*. 26(2): 113–131.

Meadows, C. and P. Narendran. 2002. "A Unification Algorithm for the Group Diffie-Hellman Protocol". In: *Workshop on Issues in the Theory of Security (WITS'02)*. Portland, Oregon.

Meier, S., C. Cremers, and D. Basin. 2010. "Strong Invariants for the Efficient Construction of Machine-Checked Protocol Security Proofs". In: *23rd IEEE Computer Security Foundations Symposium (CSF'10)*. Edinburgh, UK: IEEE. 231–245.

Milicia, G. 2002. "χ-Spaces: Programming Security Protocols". In: *14th Nordic Workshop on Programming Theory (NWPT'02)*. Tallinn, Estonia.

Millen, J. 1999. "A Necessarily Parallel Attack". In: *Workshop on Formal Methods and Security Protocols (FMSP'99)*. Trento, Italy.

Milner, R., J. Parrow, and D. Walker. 1992. "A Calculus of Mobile Processes, parts I and II". *Information and Computation*. 100(Sept.): 1–40 and 41–77.

Mödersheim, S. 2010. "Abstraction by Set-Membership: Verifying Security Protocols and Web Services with Databases". In: *17th ACM Conference on Computer and Communications Security (CCS 2010)*. ACM. Chicago, IL, USA. 351–360.

Mödersheim, S. and L. Viganò. 2009. "The Open-source Fixed-point Model Checker for Symbolic Analysis of Security Protocols". In: *Foundations of Security Analysis and Design V, FOSAD 2007 / 2008 / 2009 Tutorial Lectures*. Ed. by A. Aldini, G. Barthe, and R. Gorrieri. Vol. 5705. *Lecture Notes in Computer Science*. Springer. 166–194.

Monniaux, D. 2003. "Abstracting Cryptographic Protocols with Tree Automata". *Science of Computer Programming*. 47(2–3): 177–202.

Mukhamedov, A., A. D. Gordon, and M. Ryan. 2013. "Towards a Verified Reference Implementation of a Trusted Platform Module". In: *Security Protocols XVII*. Ed. by B. Christianson, J. A. Malcolm, V. Matyáš, and M. Roe. Vol. 7028. *Lecture Notes in Computer Science*. Springer. 69–81.

Needham, R. M. and M. D. Schroeder. 1978. "Using Encryption for Authentication in Large Networks of Computers". *Communications of the ACM*. 21(12): 993–999.

O'Shea, N. 2008. "Using Elyjah to Analyse Java Implementations of Cryptographic Protocols". In: *Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08)*. Pittsburgh, PA, USA.

Pankova, A. and P. Laud. 2012. "Symbolic Analysis of Cryptographic Protocols Containing Bilinear Pairings". In: *25th IEEE Computer Security Foundations Symposium (CSF'12)*. Cambridge, MA: IEEE. 63–77.

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

Pironti, A. and R. Sisto. 2010. "Provably Correct Java Implementations of Spi Calculus Security Protocols Specifications". *Computers and Security*. 29(3): 302–314.

Pottier, F. 2002. “A Simple View of Type-Secure Information Flow in the π-Calculus”. In: *15th IEEE Computer Security Foundations Workshop*. Cape Breton, Nova Scotia. 320–330.

Pottier, F. and V. Simonet. 2002. “Information Flow Inference for ML”. In: *29th ACM Symposium on Principles of Programming Languages (POPL’02)*. Portland, Oregon. 319–330.

Pozza, D., R. Sisto, and L. Durante. 2004. “Spi2Java: Automatic cryptographic protocol Java code generation from spi calculus”. In: *18th International Conference on Advanced Information Networking and Applications (AINA’04)*. Vol. 1. Fukuoka, Japan: IEEE Computer Society. 400–405.

Ramanujam, R. and S. Suresh. 2003. “Tagging Makes Secrecy Decidable with Unbounded Nonces as Well”. In: *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science*. Ed. by P. Pandya and J. Radhakrishnan. Vol. 2914. *Lecture Notes in Computer Science*. Mumbai, India: Springer. 363–374.

Rusinowitch, M. and M. Turuani. 2003. “Protocol Insecurity with Finite Number of Sessions is NP-complete”. *Theoretical Computer Science*. 299(1–3): 451–475.

Santiago, S., S. Escobar, C. Meadows, and J. Meseguer. 2014. “A Formal Definition of Protocol Indistinguishability and Its Verification Using Maude-NPA”. In: *Security and Trust Management (STM’14)*. Ed. by S. Mauw and C. D. Jensen. Vol. 8743. *Lecture Notes in Computer Science*. Wroclaw, Poland: Springer. 162–177.

Schmidt, B., S. Meier, C. Cremers, and D. Basin. 2012. “Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties”. In: *25th IEEE Computer Security Foundations Symposium (CSF’12)*. Cambridge, MA, USA: IEEE Computer Society. 78–94.

Schmidt, B., R. Sasse, C. Cremers, and D. Basin. 2014. “Automated Verification of Group Key Agreement Protocols”. In: *2014 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE. 179–194.

Smyth, B., M. D. Ryan, and L. Chen. 2015. “Formal analysis of privacy in Direct Anonymous Attestation schemes”. *Science of Computer Programming*. 111(2): 300–317.

Song, D., A. Perrig, and D. Phan. 2001. "AGVI—Automatic Generation, Verification, and Implementation of Security Protocols". In: *Computer Aided Verification (CAV'01)*. Ed. by G. Berry, H. Comon, and A. Finkel. Vol. 2102. *Lecture Notes in Computer Science*. Paris, France: Springer. 241–245.

Swamy, N., J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. 2011. "Secure Distributed Programming with Value-dependent Types". In: *16th International Conference on Functional Programming (ICFP 2011)*. Tokyo, Japan: ACM. 266–278.

Tiu, A. and J. Dawson. 2010. "Automating Open Bisimulation Checking for the Spi Calculus". In: *23rd IEEE Computer Security Foundations Symposium (CSF'10)*. Edinburgh, UK: IEEE. 307–321.

Turuani, M. 2006. "The CL-Atse Protocol Analyser". In: *Term Rewriting and Applications, 17th International Conference, RTA 2006*. Ed. by F. Pfenning. Vol. 4098. *Lecture Notes in Computer Science*. Seattle, WA: Springer. 277–286.

Weidenbach, C. 1999. "Towards an Automatic Analysis of Security Protocols in First-Order Logic". In: *16th International Conference on Automated Deduction (CADE-16)*. Ed. by H. Ganzinger. Vol. 1632. *Lecture Notes in Artificial Intelligence*. Trento, Italy: Springer. 314–328.

Woo, T. Y. C. and S. S. Lam. 1993. "A Semantic Model for Authentication Protocols". In: *IEEE Symposium on Research in Security and Privacy*. Oakland, California. 178–194.