# Authentication primitives and their compilation

Martín Abadi[*]
Bell Labs Research
Lucent Technologies

Cédric Fournet
Microsoft Research

Georges Gonthier[†]
INRIA Rocquencourt

**Abstract**

Adopting a programming-language perspective, we study the problem of implementing authentication in a distributed system. We define a process calculus with constructs for authentication and show how this calculus can be translated to a lower-level language using marshaling, multiplexing, and cryptographic protocols. Authentication serves for identity-based security in the source language and enables simplifications in the translation. We reason about correctness relying on the concepts of observational equivalence and full abstraction.

## 1 Authenticity from a programming-language perspective

Establishing the origins and the destinations of messages is a common problem in distributed systems. When security matters, the solutions to the problem rely on sophisticated mechanisms such as authentication protocols, digital signatures, and encryption [25]. From the perspective of programming languages [28], we may rephrase the problem and its solutions in the following general terms:

- First, we have a language with a primitive notion of principal and related operations. A principal may represent a location (e.g., an IP address) or an entity that owns that location (e.g., a user). A typical operation may enable the recipient of a message to determine the origin of the message.
- A compiler implements those primitives by mapping them to security mechanisms. For example, principals of a source program may be associated with cryptographic keys in the corresponding target program.
- The output of the compiler is lower-level code, written in a lower-level vocabulary that permits expressing or at least invoking cryptographic algorithms and protocols. The lower-level code may be harder to understand, but closer to an executable distributed implementation since it does not rely on high-level authentication primitives.
- Attackers may write high-level code and compile it to low-level code, or may write low-level code directly.

[*] Part of this work was done at Compaq's Systems Research Center.
[†] Partly supported by ESPRIT CONFER-2 WG-21836 and by RNRT project MARVEL 98S0347.

To appear in the Proceedings of the 27th ACM Symposium on Principles of Programming Languages (January 2000), pages 302–315. This version includes minor corrections in Figure 7.

This perspective brings to bear ideas and methods from programming languages on security issues. Some of these issues are mundane but important. For example, current descriptions of security mechanisms are often imprecise and confusing; notations from programming languages should help (e.g., [24]). Other issues, like the correctness of authentication protocols, are notoriously subtle and challenging. When we view authentication protocols in context, as part of the output of a compiler, we have an opportunity to shed some new light on their correctness. We demonstrate such advantages of the programming-language perspective in this paper.

We define a distributed process calculus with principals and related operations, show how this calculus can be compiled into another calculus with cryptographic operations, and study the properties of the compilation method. Thus, we build on recent work on the secure implementation of channel abstractions [2, 3]. The main contribution of this paper is the treatment of authentication, which has at least two benefits. First, high-level programs can use principal identities for security, for example in access-control lists. The second benefit is more surprising: the treatment of principal identities allows a lighter, cheaper implementation, as explained below.

### Overview

As source language, we adopt a variant of the join-calculus [15, 14]. In addition to constructs for channel definitions, parallel composition, and conditionals, which are part of the core join-calculus, this variant also includes a syntactic category of principals and the following process forms:

- $x\langle a : v_2, \ldots , v_n\rangle$ is an authenticated message on channel $x$, with origin $a$ and contents $v_2, \ldots , v_n$. (As in secure network objects [33], the first argument of every authenticated message conveys the emitter; we write $a : v_2, \ldots, v_n$ instead of $a, v_2, \ldots, v_n$ in order to stress this convention.)

- $\mathsf{let}\ a = prin(x)\ \mathsf{in}\ P$ determines the identity $a$ of the unique destination of channel $x$, then executes the process $P$.
- $a[P]$ represents the principal $a$ executing the process $P$.

As target language, we adopt another variant of the join-calculus, without principals but with constructs for cryptography. In mapping the source language to the target language, we associate each principal with a public-key pair. Naively, we may try the following implementations:

- $x\langle a : v_2, \dots , v_n\rangle$ corresponds to a message signed with $a$'s signature key.
- $\mathsf{let}\ a = prin(x)\ \mathsf{in}\ P$ obtains $a$ from a certificate that associates $x$ with $a$, signed with $a$'s signature key.
- $a[P]$ corresponds to $P$ plus some layers of communications processing with access to $a$'s keys.

With sufficient care, this sketch can be turned into a precise translation. In addition, the translation can be optimized, for example by multiplexing messages of several high-level channels on a unique low-level location-to-location channel.

We pursue this approach using concepts and techniques from programming-language theory. In particular, as in our recent work, our main results are full-abstraction theorems which say that our translations preserve observational equivalence. Intuitively, these results do not rule out the possibility of attacks against the programs in the source language, but they imply that the translations do not enable new attacks [1]. They mean that reasoning about the security of programs in the source language, which is relatively simple and clear, applies also to the corresponding programs in the target language.

Full abstraction can be expensive. In our recent work, the pursuit of full abstraction led us to implement point-to-point messages by a combination of broadcasts, public-key decryption, and filtering, in order to thwart traffic-analysis attacks. Similarly, it led us to the use of independent keys for each channel or even for each message in order to allow the mutual anonymity of emitters and receivers. (In the join-calculus, which was not originally designed with security in mind, an emitter need not know whether two channels have the same destination and a receiver need not know whether two messages have the same origin. These properties must be preserved by any fully abstract implementation.)

In this paper, we avoid many of those costs without giving up full abstraction. We achieve this simplification by a careful choice of the source calculus, by noticing that certain inessential observational equivalences do not hold in the new source calculus because of the presence of identity information, and by the cautious application of standard implementation techniques. For example, the presence of identity information removes the mutual anonymity of emitters and receivers, and enables some reuse of keys. Thus, although our method remains unimplemented and still requires optimization, we take a substantial step toward a realistic and correct implementation of a process calculus with secure communication.

The next section introduces our source calculus. Section 3 introduces our target calculus and a simple network model. Section 4 describes the structure of our translations. This section leaves open the choice of a low-level cryptographic protocol, which is the subject of section 5. Section 6 presents our results. Finally, section 7 discusses some related work and concludes. All proofs and many technical details are omitted.

## 2 A calculus with authentication

In this section we describe the variant of the join-calculus that serves as our high-level language. It has an explicit model of distribution and of principals. An appendix contains a review of a standard join-calculus, which has been presented in several previous papers and which is the point of departure for this section.

### 2.1 Syntax and semantics

The syntax of our source calculus is given in Figure 1.

A distributed computation is modeled as a configuration, that is, as an assembly of processes. Each process corresponds to a principal or location (in the sense of the distributed join-calculus and of other process calculi with locations, e.g., [12, 31, 6]).

We let $\mathcal{A}$ be a countable set of principal names, and $(\mathcal{N}_a)_{a\in\mathcal{A}}$ be a family of countable, disjoint sets of channel names indexed by principals. Hence, every channel name corresponds to one principal. We implicitly rely on a type system (see the appendix), assume that every set $\mathcal{N}_a$ contains infinitely many names for each type, and consider only well-typed processes. We write $\mathsf{Principal}$ for the type of principal names, and $\langle\tau_1, \dots , \tau_n\rangle$ for the type of a channel that carries tuples of values with types $\tau_1, \dots , \tau_n$.

The grammar for processes is fairly standard. Some of its constructs have already been discussed in the overview. In the process $\mathsf{def}_S\ D\ \mathsf{in}\ P$, the subscript $S$ is the subset of the names defined in $D$ that is exported to the context. The scope of the names in $S$ is unrestricted. The scope of the other names defined in $D$ is restricted to the process $P$ and to the guarded processes in $D$. (This use of exported names and the construct $\mathsf{def}_S\ D\ \mathsf{in}\ P$ come from the open join-calculus [11].)

Throughout, we consider only processes $P$ that satisfy the following well-formedness conditions which restrict the use of exported names:

- If a name is exported by a definition $\mathsf{def}_S\ D\ \mathsf{in}\ Q$ inside $P$, it cannot be exported by any other definition in $P$; names defined by $D$ cannot be exported by any definition in $Q$.
- Definitions that appear in subexpressions of $P$ of the forms $J \triangleright Q$, $\mathsf{if}\ u = v\ \mathsf{then}\ Q\ \mathsf{else}\ Q'$, or $\mathsf{let}\ a = prin(x)\ \mathsf{in}\ Q$ do not export any names. (In the terminology of Figure 2, only definitions in evaluation contexts may export names.)

We write $\mathsf{rv}(J)$ for the formal parameters in $J$, $\mathsf{dv}(D)$ for the names defined in $D$, $\mathsf{fv}(P)$ for the names free in $P$, and $\mathsf{xv}(P)$ for the names exported by $P$. For instance, we have:

$$\mathsf{xv}(P \mid Q) \overset{\text{def}}{=} \mathsf{xv}(P) \uplus \mathsf{xv}(Q)$$
$$\mathsf{fv}(P \mid Q) \overset{\text{def}}{=} (\mathsf{fv}(P) \setminus \mathsf{xv}(Q)) \cup (\mathsf{fv}(Q) \setminus \mathsf{xv}(P))$$

where $S \uplus S'$ represents the union of $S$ and $S'$ plus the requirement that $S$ and $S'$ be disjoint sets. We omit the other scoping rules.

The sets $\mathsf{fv}(P)$ and $\mathsf{xv}(P)$ are disjoint; they form the interface of $P$. The interface of a process can evolve during

| | | | |
|---|---|---|---|
| $u, v$ | $::=$ | | values |
| | | $x$ | channel |
| | $\mid$ | $a$ | principal |
| $P, Q$ | $::=$ | | processes |
| | | $x\langle a : u_2, \ldots, u_n\rangle$ | authenticated message |
| | $\mid$ | $\mathsf{def}_S\ D\ \mathsf{in}\ P$ | local definition |
| | $\mid$ | $P \mid Q$ | parallel composition |
| | $\mid$ | $\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$ | comparison |
| | $\mid$ | $\mathsf{let}\ a = \mathit{prin}(x)\ \mathsf{in}\ P$ | principal extraction |
| | $\mid$ | $\mathsf{0}$ | null process |
| $D$ | $::=$ | | definitions |
| | | $J \triangleright P$ | reaction rule |
| | $\mid$ | $D \wedge D$ | conjunction |
| $J$ | $::=$ | | join patterns |
| | | $x\langle a : u_2, \ldots, u_n\rangle$ | authenticated message |
| | $\mid$ | $J \mid J$ | synchronization |
| $A, B, C$ | $::=$ | | configurations |
| | | $a[P]$ | running principal |
| | $\mid$ | $C \setminus S$ | restriction |
| | $\mid$ | $C \parallel C'$ | parallel composition |

Figure 1: Grammar of the high-level calculus.

*Structural equivalence* $\equiv$ *for processes* is the smallest equivalence such that:

1. $P \equiv Q$ when $P$ is $\alpha$-convertible to $Q$
2. $\wedge$ and $\mid$ are associate-commutative modulo $\equiv$, with unit $\mathsf{0}$ for $\mid$
3. $\equiv$ is closed by application of evaluation contexts
   $E(\cdot)\ ::=\ (\cdot)\ \mid\ P \mid E(\cdot)\ \mid\ \mathsf{def}_S\ D\ \mathsf{in}\ E(\cdot)$
4. $P \mid \mathsf{def}_S\ D\ \mathsf{in}\ Q\ \equiv\ \mathsf{def}_S\ D\ \mathsf{in}\ P \mid Q$ when $\mathsf{fv}(P) \cap \mathsf{dv}(D) \subseteq S$
5. $\mathsf{def}_S\ D\ \mathsf{in}\ \mathsf{def}_{S'}\ D'\ \mathsf{in}\ P\ \equiv\ \mathsf{def}_{S \uplus S'}\ D \wedge D'\ \mathsf{in}\ P$ when $\mathsf{fv}(D) \cap \mathsf{dv}(D') \subseteq S'$

*Reduction* $\rightarrow$ *for processes* is the smallest relation closed by application of evaluation contexts such that:

JOIN $\mathsf{def}_S\ D\ \mathsf{in}\ J\sigma \mid Q\ \rightarrow\ \mathsf{def}_S\ D\ \mathsf{in}\ P\sigma \mid Q$
when $D = J \triangleright P$ or $D = J \triangleright P \wedge D'$
and $\mathit{dom}(\sigma) = \mathsf{rv}(J)$

TEST $\mathsf{if}\ u = u\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \rightarrow\ P$
$\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \rightarrow\ Q$ when $u \neq v$

PRIN $\mathsf{let}\ a = \mathit{prin}(x)\ \mathsf{in}\ P\ \rightarrow\ P\{^b/_a\}$
when $x \in \mathcal{N}_b$

STRUCT $$\frac{P\ \equiv\ Q \qquad Q\ \rightarrow\ Q' \qquad Q'\ \equiv\ P'}{P\ \rightarrow\ P'}$$

*Structural equivalence* $\equiv$ *for configurations* is the smallest equivalence such that:

1. $A \equiv B$ when $A$ is $\alpha$-convertible to $B$
2. $\parallel$ is associative-commutative modulo $\equiv$
3. $\equiv$ is closed by application of configuration contexts
   $G(\cdot)\ ::=\ (\cdot)\ \mid\ A \parallel G(\cdot)\ \mid\ G(\cdot) \setminus S$
4. $a[\mathsf{def}_S\ D\ \mathsf{in}\ P]\ \equiv\ a[\mathsf{def}_{S \cup S'}\ D\ \mathsf{in}\ P] \setminus S'$ when $S' \subseteq \mathcal{N}_a \cap \mathsf{dv}(D) \setminus S$
5. $C \setminus S \parallel b[Q]\ \equiv\ (C \parallel b[Q]) \setminus S$
6. $(C \setminus S) \setminus S'\ \equiv\ C \setminus S \uplus S'$
7. $\dfrac{P\ \equiv\ Q}{a[P]\ \equiv\ a[Q]}$

*Reduction* $\rightarrow$ *for configurations* is the smallest relation closed by application of configuration contexts such that:

LOCAL $$\frac{P\ \rightarrow\ P'}{a[P]\ \rightarrow\ a[P']}$$

COMM $a[P \mid x\langle c : \widetilde{v}\rangle] \parallel b[Q]\ \rightarrow\ a[P] \parallel b[Q \mid x\langle a : \widetilde{v}\rangle]$
when $x \in \mathcal{N}_b$

STRUCT $$\frac{A\ \equiv\ A' \qquad A'\ \rightarrow\ B' \qquad B'\ \equiv\ B}{A\ \rightarrow\ B}$$

Figure 2: Operational semantics for processes and configurations in the high-level calculus.

$$s \left[ \mathsf{def}_{\{entry,query\}} \begin{array}{rll} & entry\langle a : x\rangle \mid open\langle s :\rangle & \triangleright\ won\langle s : a, x\rangle \\ \wedge & query\langle a : y\rangle \mid won\langle s : b, x\rangle & \triangleright\ y\langle s : b, x\rangle \mid won\langle s : b, x\rangle \end{array} \ \mathsf{in}\ open\langle s :\rangle \right]$$

$$\|\quad p\,[\ \mathsf{let}\ s' = prin(entry)\ \mathsf{in\ if}\ s = s'\ \mathsf{then}\ entry\langle p : v\rangle\ \mathsf{else}\ 0\ ]$$

Figure 3: An example.

computation: names in $\mathsf{xv}(P)$ can be used for receiving messages, which may contain additional free names; conversely, names in $\mathsf{fv}(P)$ can be used for sending messages, which may export additional defined names.

The grammar for configurations describes the parallel composition of principals. It also has a restriction operator for names exported by these principals: in the configuration $C \setminus S$, the scope of the names in $S$ is exactly $C$.

We also adopt well-formedness conditions on configurations:

- Each principal is defined at most once.
- Principal names are never restricted.
- A definition in a principal $a$ may define or export a name only if the name is in $\mathcal{N}_a$.
- In a configuration that defines a principal $a$, all the names in $\mathcal{N}_a$ that appear elsewhere in the configuration must be exported by $a$.
- In a configuration that contains a restricted configuration $C \setminus S$, all the names in $S$ must be exported by $C$ and can appear only in $C$.

These conditions can be lifted to well-formedness conditions for configuration contexts; when we apply a context we implicitly assume that the resulting configuration is well-formed.

The operational semantics of the source calculus is given in Figure 2, first for processes within a principal, then for configurations of principals. The rule Local models local computation steps; these steps may involve the synchronization of several messages, but they concern a single principal. The rule Comm models the transmission of a message. In it, we write $\widetilde{v}$ for a tuple $v_2, \ldots, v_n$. Comm overwrites the first argument of the transported message $c : \widetilde{v}$ with the true name of the sending principal $a$. Thus, it offers built-in, reliable authentication of the message. In addition, the rule guarantees that the message goes only to the principal $b$ that exports the channel $x$ (rather than to an attacker), thus helping with the secrecy of the message. Complementarily, the rule Prin serves to identify the destination of a channel.

Thus, the syntax and semantics of the source calculus contain substantial constraints on the context of a process and what this context can do. When we view the context as potentially hostile, these constraints amount to built-in security properties. In this paper, we are interested in translations that enforce those properties.

## 2.2 Examples

As a small example, we show a variant of the contest of a previous paper [3] with authentication. The contest relies on a server that creates a channel and exports its name; the winner is simply the participant whose entry arrives first on this channel. Using authentication primitives, the participants and the server can establish each other's identity.

The first running principal of Figure 3 defines the server at location $s$. The second one is a simple participant at location $p$. The names *open* and *won* are local to the server. The message $open\langle s :\rangle$ represents that the contest is open. The first message on *entry* causes it to be closed, with the sender as winner. Subsequent messages on *query* return the identity of the winner and the winning entry on a continuation channel $y$.

Other principals may run in parallel with these. Some of them may be legitimate participants in the contest. Others may be attackers; they may act as participants some of the time, but they may also attempt to intercept or falsify messages. We are interested in the properties of our system in an arbitrary (implicit) context that includes such principals.

Informally, we have the following properties:

- Each entry is accompanied by the identity of its sender, so the server knows which participant is responsible for the first entry and should get credit for it.
- The participants can determine the destination of the channels *entry* and *query*, so they can know where their entries and queries go.
- Each result of a query is accompanied by the identity of the server, so the participant that receives the result knows that it is authentic.

It is easy to see that a participant is communicating with the server if it uses, literally, the names *entry* and *query* and the names of fresh continuation channels. These properties apply equally in more intricate situations where a participant uses other channel names (which could be dynamically instantiated to *entry* and *query*) for communicating with the server.

It is also possible to recast bigger programming examples in this setting. For instance, imitating the client-server examples of secure network objects [33], we could define a file server that identifies its client when it receives a request to open a file, and returns a handle for a file object that performs no further identity checks; or a terminal server and a client that identify each other before starting a shell. Of course, as the examples become more elaborate, they call for the use of a full programming language with libraries rather than a small process calculus.

## 3 A lower-level calculus with cryptography

In this section, we present a slight variant of the sjoin-calculus [2], which we use as a low-level implementation language; it includes primitives for public-key and shared-key cryptography. As a low-level counterpart of configurations, we describe a (trivial) public-key infrastructure. We also introduce a low-level network model.

| | | | |
|---|---|---|---|
| $u, v$ | $::=$ | | values |
| | | $a, b, a^+, a^-, x, y$ | names |
| | $\mid$ | $\{u\}_v$ | encryption : $\mathsf{BitString}$ |
| | $\mid$ | $(u)_v$ | signature : $\mathsf{BitString}$ |
| | $\mid$ | $\tau$ | type representation : $\mathsf{BitString}$ |
| | $\mid$ | $0, u+1$ | integer representations : $\mathsf{BitString}$ |
| | $\mid$ | $u.v$ | concatenation : $\mathsf{BitString}$ |
| $P, Q$ | $::=$ | | processes |
| | | $x\langle u_1, \ldots, u_n \rangle$ | message |
| | $\mid$ | $\mathsf{def}_S\ D\ \mathsf{in}\ P$ | local definition |
| | $\mid$ | $\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$ | comparison |
| | $\mid$ | $\mathsf{repl}\ P$ | replication |
| | $\mid$ | $\mathsf{decrypt}\ v\ \mathsf{using}\ v'\ \mathsf{to}\ V\ \mathsf{in}\ P\ \mathsf{else}\ Q$ | decryption |
| | $\mid$ | $\mathsf{check}\ v\ \mathsf{authenticates}\ v'\ \mathsf{using}\ v''\ \mathsf{in}\ P\ \mathsf{else}\ Q$ | signature verification |
| | $\mid$ | $P \mid P'$ | parallel composition |
| | $\mid$ | $P \setminus S$ | restriction |
| | $\mid$ | $0$ | null process |
| $D$ | $::=$ | | definitions |
| | | $J \triangleright P$ | reaction rule |
| | $\mid$ | $\mathsf{fresh}\ x$ | fresh name |
| | $\mid$ | $\mathsf{key}\ x$ | fresh shared key |
| | $\mid$ | $\mathsf{keys}\ x^+, x^-$ | fresh pair of keys |
| | $\mid$ | $D \wedge D'$ | conjunction |
| $J$ | $::=$ | | join patterns |
| | | $x\langle V_1, \ldots, V_n \rangle$ | message pattern |
| | $\mid$ | $J \mid J'$ | synchronization |
| $V$ | $::=$ | | value patterns |
| | | $x$ | formal parameter |
| | $\mid$ | $V.V$ | decomposition |
| | $\mid$ | '$v$' | quoted value |

Figure 4: Grammar of the low-level calculus.

### 3.1 Syntax and semantics

This calculus is an open variant of the sjoin-calculus with signatures, with type representations of type BitString, and with compound BitString values. Its syntax is in Figure 4; it is constrained by well-formedness conditions on exported names and on restrictions, as in section 2.1 (e.g., decryptions and signature verifications cannot export names, and only exported names may be restricted). Its operational semantics is in Figure 5.

Values of type BitString can be concatenated. We rely on pattern-matching syntax in join-patterns in order to extract the parts of the resulting compound values. So we distinguish formal parameters $x$ from value patterns '$v$' in the grammar for join-patterns, and require that the substitution used to match a message against a join-pattern should not affect the value patterns in the join-pattern. Formal parameters are bound, while value patterns may contain free variables.

The value $\{u\}_v$ represents the result of encrypting $u$ using $v$ as encryption key; $\{u\}_v$, $u$, and $v$ are all of type BitString. The process

$$\mathsf{decrypt}\ v\ \mathsf{using}\ v'\ \mathsf{to}\ V\ \mathsf{in}\ P\ \mathsf{else}\ Q$$

attempts to decrypt $v$ using $v'$ as decryption key. If the decryption succeeds and yields the cleartext $V\sigma$ for some substitution $\sigma$, then $P\sigma$ runs. Otherwise, $Q$ runs. The else branch can be omitted.

Similarly, the value $(u)_v$ represents the result of signing $u$ using $v$ as signature key; $(u)_v$, $u$, and $v$ are all of type BitString. The process

$$\mathsf{check}\ v\ \mathsf{authenticates}\ v'\ \mathsf{using}\ v''\ \mathsf{in}\ P\ \mathsf{else}\ Q$$

runs $P$ if $v$ is the result of signing $v'$ with the inverse of $v''$; otherwise, it runs $Q$. (This addition of signatures is convenient but not essential; with some care, signatures can be identified with encryptions.)

In our translations, we use data structures that can be coded in this calculus, in particular association tables (assoc ..., from [3]). We postpone the specification of association tables to an appendix.

### 3.2 Trivial public-key infrastructure

Informally, we assume given a public-key infrastructure that can reliably deliver the public key and the network address for every principal name. In our low-level model, we keep this assumption implicit: we directly use the public key $a^+$ as the name of the principal $a$ and as its network address.

*Structural equivalence $\equiv$ for low-level processes* is the smallest equivalence such that:

1. $P \equiv Q$ when $P$ is $\alpha$-convertible to $Q$
2. $\wedge$ and $|$ are associate-commutative modulo $\equiv$, with unit $\mathsf{0}$ for $|$
3. $\equiv$ is closed by application of evaluation contexts $E(\cdot) \ ::= \ (\cdot) \ | \ P|E(\cdot) \ | \ E(\cdot) \setminus S \ | \ \mathsf{def}_S\ D \ \mathsf{in}\ E(\cdot)$
4. $P \mid \mathsf{def}_S\ D \ \mathsf{in}\ Q \ \equiv \ \mathsf{def}_S\ D \ \mathsf{in}\ P \mid Q$ when $\mathsf{fv}(P) \cap \mathsf{dv}(D) \subseteq S$
5. $\mathsf{def}_S\ D \ \mathsf{in}\ \mathsf{def}_{S'}\ D' \ \mathsf{in}\ P \ \equiv \ \mathsf{def}_{S \uplus S'}\ D \wedge D' \ \mathsf{in}\ P$ when $\mathsf{fv}(D) \cap \mathsf{dv}(D') \subseteq S'$
6. $\mathsf{def}_S\ D \ \mathsf{in}\ P \ \equiv \ (\mathsf{def}_{S \cup S'}\ D \ \mathsf{in}\ P) \setminus S'$ when $S' \subseteq \mathsf{dv}(D) \setminus S$
7. $P \setminus S \mid Q \ \equiv \ (P \mid Q) \setminus S$ when $\mathsf{fv}(Q) \cap S = \emptyset$
8. $(P \setminus S) \setminus S' \ \equiv \ P \setminus (S \uplus S')$

*Reduction $\rightarrow$ for low-level processes* is the smallest relation closed by application of evaluation contexts such that:

| | |
|---|---|
| JOIN | $\mathsf{def}_S\ D \ \mathsf{in}\ J\sigma \ \rightarrow \ \mathsf{def}_S\ D \ \mathsf{in}\ P\sigma$ when $D = J \triangleright P$ or $D = J \triangleright P \wedge D'$ and $dom(\sigma) = \mathsf{rv}(J)$ |
| REPL | $\mathsf{repl}\ P \ \rightarrow \ P \mid \mathsf{repl}\ P$ |
| TEST | $\mathsf{if}\ u = u \ \mathsf{then}\ P \ \mathsf{else}\ Q \ \rightarrow \ P$ |
| | $\mathsf{if}\ u = v \ \mathsf{then}\ P \ \mathsf{else}\ Q \ \rightarrow \ Q$ when $u \neq v$ |
| DECRYPT | $\mathsf{def}_S\ \mathsf{keys}\ x^+, x^- \ \mathsf{in}\ \mathsf{decrypt}\ \{V\sigma\}_{x^+} \ \mathsf{using}\ x^- \ \mathsf{to}\ V \ \mathsf{in}\ P \ \mathsf{else}\ P' \ \rightarrow \ \mathsf{def}_S\ \mathsf{keys}\ x^+, x^- \ \mathsf{in}\ P\sigma$ when $dom(\sigma) = \mathsf{rv}(V)$ |
| | $\mathsf{def}_S\ \mathsf{keys}\ x^+, x^- \ \mathsf{in}\ \mathsf{decrypt}\ v \ \mathsf{using}\ x^- \ \mathsf{to}\ V \ \mathsf{in}\ P \ \mathsf{else}\ P' \ \rightarrow \ \mathsf{def}_S\ \mathsf{keys}\ x^+, x^- \ \mathsf{in}\ P'$ when $v \neq \{V\sigma\}_{x^+}$ for all $\sigma$ |
| CHECK | $\mathsf{def}_S\ \mathsf{keys}\ x^+, x^- \ \mathsf{in}\ \mathsf{check}\ (v)_{x^-} \ \mathsf{authenticates}\ v \ \mathsf{using}\ x^+ \ \mathsf{in}\ P \ \mathsf{else}\ P' \ \rightarrow \ \mathsf{def}_S\ \mathsf{keys}\ x^+, x^- \ \mathsf{in}\ P$ |
| | $\mathsf{def}_S\ \mathsf{keys}\ x^+, x^- \ \mathsf{in}\ \mathsf{check}\ u \ \mathsf{authenticates}\ v \ \mathsf{using}\ x^+ \ \mathsf{in}\ P \ \mathsf{else}\ P' \ \rightarrow \ \mathsf{def}_S\ \mathsf{keys}\ x^+, x^- \ \mathsf{in}\ P'$ when $u \neq (v)_{x^-}$ |
| STRUCT | $\dfrac{P \ \equiv \ Q \qquad Q \ \rightarrow \ Q' \qquad Q' \ \equiv \ P'}{P \ \rightarrow \ P'}$ |

The semantics of $\mathsf{key}\ x$ is that of $\mathsf{keys}\ x^+, x^-$ in the case $x^+ = x^- = x$.

Figure 5: Operational semantics for the low-level calculus.

(Alternatively, we could maintain a local cache that maps $a$ to $a^+$ and to $a$'s address, and have our protocols query the public-key infrastructure whenever the data is not present in the cache.)

For each principal $a$ that we translate, we generate a pair of keys $a^+, a^-$. The public key $a^+$ is available to all principals who refer to $a$ or have free variables belonging to $a$, and also to any attacker. On the other hand, the private key $a^-$ is never communicated outside the translation of the principal, although it may appear as part of the signature of a message to be verified using $a^+$.

## 3.3 Encoding of channel names

Next we define the wire format for communicating channel names. Channel names represent abstract communication capabilities; to communicate such capabilities, we use a combination of identifiers and certificates.

For $a \in \mathcal{A}$, each channel name $x \in \mathcal{N}_a$ of type $\tau$ is encoded as a compound value of type $\mathsf{BitString}$

$$[\![x]\!] \stackrel{\text{def}}{=} a^+.\widehat{x}.(\widehat{x}.\tau)_{a^-}$$

that contains the public key $a^+$ of principal $a$, a local id $\widehat{x}$, and a certificate that consists of $\widehat{x}$ and $\tau$ signed with $a$'s private key $a^-$.

The certificate proves that $a$ says that the channel name that $[\![x]\!]$ represents really is defined at $a$ with type $\tau$ and was originally exported by $a$. Thus, it permits a sound implementation of the principal extraction operation. In our approach, the certificate is checked when $[\![x]\!]$ is received at a location; but a lazier check would suffice.

Because $[\![x]\!]$ includes this unguessable certificate, the constraints on the choice of the local id $\widehat{x}$ are fairly weak: the choice must not be correlated in any way with the execution of $a$, and $\widehat{x}$ must be different from $\widehat{y}$ for any other channel $y$ of type $\tau$ exported by $a$. In particular, we could allow an attacker to guess $\widehat{x}$ since the certificate still guarantees the authenticity of $[\![x]\!]$. For simplicity, however, we generate a fresh name for each local id.

## 3.4 Network model

Our model of the low-level network corresponds to a wide-area IP network. Messages are idealized packets with an origin address, a destination address, and some payload data. These three components are arbitrary $\mathsf{BitString}$ values.

We model as a single process the network that interconnects all locations and the presence of hostile locations that can intercept and emit any message on the network, independently of its origin and destination address.

Let $A$ be a finite subset of $\mathcal{A}$. Informally, $A$ contains the names of honest principals. For every $a \in A$, the low-level network exports a name $send_a : \langle \mathsf{BitString}, \mathsf{BitString}\rangle$ and imports a name $recv_a : \langle \mathsf{BitString}, \mathsf{BitString}\rangle$ as its interface to the implementation of principal $a$.

An attacker may have access to some certificates, and may try to use those certificates for impersonating some principals. Yet, the attacker should not be given direct access to the secret keys of the principals. In addition, for syntactic sanity, the attacker should not export the keys of a principal $a \in A$ or a name $x$ that $a$ exports. These assumptions are the subject of the following definition.

**Definition 1 (Restrained process)** *A process $N$ is* restrained *(with respect to $A$ and $X$) when it does not export names $a^+, a^-$ with $a \in A$ or $\hat{x}$ with $x \in X$, and when names $a^-$ with $a \in A$ occur free in $N$ only in values of the form $(\widehat{x}.\tau)_{a^-}$ with $x \in X \cap \mathcal{N}_a$ of type $\tau$.*

In addition, we introduce some explicit assumptions on the behavior of the low-level network. These assumptions imply that, for example, an attacker cannot permanently disconnect two honest principals. The attacker can emit and intercept any number of messages, for all addresses, but it cannot entirely prevent messages between honest principals to pass through the network unnoticed, at least from time to time. Thus, the low-level network should have a persistent routing property.

**Definition 2 (Valid network)** *The set $\mathcal{IP}$ of* valid networks *is the largest set of processes such that $N \in \mathcal{IP}$ implies:*

**Routing** *For all $a, b \in A$ and every* $\mathsf{BitString}$ *name $m$,*

$$N \mid send_a\langle b^+, m\rangle \quad \rightarrow^* \quad N \mid recv_b\langle a^+, m\rangle$$

**Exports** *$recv_a \notin \mathsf{xv}(N)$ for all $a \in A$ (that is, $N$ does not export any of the names $recv_a$).*

**Closure under reduction** *If $N \rightarrow N'$, then $N' \in \mathcal{IP}$.*

**Closure under parallel composition** *For all $P$, if $N \mid P$ is well-formed and $recv_a \notin \mathsf{xv}(P)$ for all $a \in A$, then $N \mid P \in \mathcal{IP}$.*

**Closure under extrusion** *If $N \equiv (N' \mid recv_a\langle u, v\rangle) \setminus X$, with $X \subseteq \mathsf{fv}(u) \cup \mathsf{fv}(v)$, then $N' \in \mathcal{IP}$.*

For instance, a reliable implementation of the network could simply perform the multiplexing for all addresses $a^+$ with $a \in A$, as described in the process below.

$$N \quad \overset{\text{def}}{=} \quad \mathsf{def}_{\{send_a \mid a \in A\}} \bigwedge_{a,b \in A} send_a\langle `b^+\text{'}, x\rangle \triangleright recv_b\langle a^+, x\rangle \quad \mathsf{in}\ 0$$

This process $N$ is a valid network. Our results apply to this particular process, but also to any other valid network.

## 3.5 Traffic analysis

It is prudent to have some kind of background traffic on the network in order to protect against trivial traffic-analysis attacks. As background traffic, we will use messages that propagate public keys.

It is also prudent to emit every encrypted message just once, so that traffic observed by an attacker is a flow of pairwise-disjoint values that are also different from any value that the attacker may build. Accordingly, we define some syntactic sugar that helps in preserving message distinctness; whenever we write a value $v$ containing the placeholder $\_$ in a message $x\langle v\rangle$, this abbreviates a process $\mathsf{def\ fresh}\ d\ \mathsf{in}\ x\langle v\{^d/_\_\}\rangle$ for some name $d$ that does not occur in $x\langle v\rangle$. Conversely, whenever we write a placeholder $\_$ in a pattern, this stands for a variable that does not appear elsewhere.

## 4 Translation structure

In this section, we describe a mapping from the high-level calculus to the low-level calculus. In particular, we show how to translate a high-level process within a running principal and how cryptographic protocols can be used for communication between principals. We also describe filtering and multiplexing "glue" code. Although we specify the interface for point-to-point communication protocols, we leave the presentation of actual protocols for the next section.

## 4.1 Framework

We describe a translation framework for principals with names in a given, finite set $A$. The translation of each principal does not rely on the translation of any other principal in $A$. To translate a configuration that contains principals with names in $A$, we independently map every principal $a[P]$ to an implementation running $P$ at a network address $a^+ : \mathsf{BitString}$, and ensure that communication between principals occurs only through a public network.

Within a given principal, we translate a process compositionally, by substituting $a^+ : \mathsf{BitString}$ for $a : \mathsf{Principal}$ and $\mathsf{let}\ a^+.\_ = t_\tau(x)\ \mathsf{in}\ [\![P]\!]$ for $\mathsf{let}\ a = prin(x)\ \mathsf{in}\ P$ when $x$ has type $\tau$. The name $t_\tau$ will be provided by the context; it gives access to the table of network representations for exported or remote channels of type $\tau$.

Our translation of a configuration $C$ is also compositional. It is parameterized by a family $(B_a)_{a \in A}$ of finite sets of principal names, where $A$ contains at least the names of all the principals defined in $C$.

$$\begin{aligned} [\![a[P]]\!] &\overset{\text{def}}{=} \mathcal{F}^{a:\mathsf{xv}(P)}_{B_a:\mathsf{fv}(P)}([\![P]\!]) \\ [\![C \parallel C']\!] &\overset{\text{def}}{=} [\![C]\!] \mid [\![C']\!] \\ [\![C \setminus S]\!] &\overset{\text{def}}{=} [\![C]\!] \setminus \widehat{S} \end{aligned}$$

The context $\mathcal{F}^{a:X}_{B:S}(\,\cdot\,)$, defined in Figure 7, serves as a wrapper (or firewall) that takes care of filtering, multiplexing, and cryptographic operations; it is explained below.

## 4.2 The context $\mathcal{F}^{a:X}_{B:S}(\,\cdot\,)$

Next we detail the stack of filters and protocols that locally implement communications processing with authentication. We assume that $a$ is the name of the principal being implemented, and that $\Sigma$ is a set of channel types that contains all the types of channel names in $S \cup X$ and that is closed under decomposition (that is, if $\langle \tau_1, \dots, \tau_n\rangle \in \Sigma$ then $\tau_1, \dots, \tau_n \in \Sigma \cup \{\mathsf{Principal}\}$).

The context $\mathcal{F}^{a:X}_{B:S}(\,\cdot\,)$ uses mechanisms similar to those of [3], but it does much more processing (e.g., multiplexing steps from typed channels to global uids, then to remote locations). Its overall structure is sketched in Figure 6; its definition is given in Figure 7, top-down.

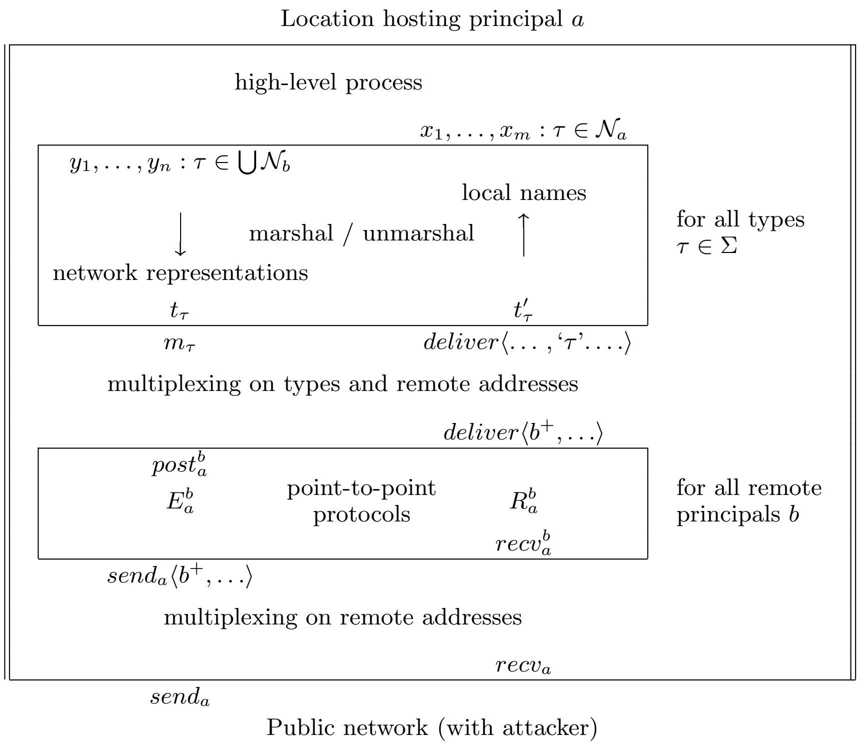
Location hosting principal $a$
high-level process
$x_1, \ldots, x_m : \tau \in \mathcal{N}_a$
$y_1, \ldots, y_n : \tau \in \bigcup \mathcal{N}_b$
local names
marshal / unmarshal
network representations
for all types $\tau \in \Sigma$
$t_\tau$
$t'_\tau$
$m_\tau$
$deliver\langle \ldots, `\tau' \ldots \rangle$
multiplexing on types and remote addresses
$deliver\langle b^+, \ldots \rangle$
$post_a^b$
$E_a^b$
point-to-point protocols
$R_a^b$
$recv_a^b$
for all remote principals $b$
$send_a\langle b^+, \ldots \rangle$
multiplexing on remote addresses
$recv_a$
$send_a$
Public network (with attacker)

Figure 6: Protocol stack for $a$'s communications: interfaces.

**Typed interface to the source process** The upper part of the context contains tables that record the association between local channels $x$ and their wire representations $[\![x]\!]$, for each type $\tau$. For a given set of names $S$, for every $\tau \in \Sigma$, we let $S_\tau$ be the set of names of type $\tau$ in $S$. The table for type $\tau$ contains representations for all channels in $S_\tau \cup X_\tau$. The tables are accessed bidirectionally. The representation of a channel $x$ is looked up for output on $x$, for exporting $x$, and for extracting the identity of $x$'s destination. Conversely, the channel $x$ associated with a representation $[\![x]\!]$ is looked up for input on $x$ and for importing $x$. The tables are extended with $[\![x]\!]$ when they are queried with a new local channel name $x \in \mathcal{N}_a$, and conversely with a new local surrogate when they are queried with a new representation. The validity of representations (and of the certificates within) must always be checked before accessing the tables in order to ensure the integrity of the principal extraction operation.

**Multiplexing on types and remote addresses** Going down, $m_\tau$ gathers wire representations for all arguments, and passes a concatenation of BitString values to the emission protocol running for the remote address $b^+$. Conversely, on the way up, *deliver* extracts a type representation from its BitString argument, and uses it to drive the unmarshaling of the channel and of its arguments.

**Interface to the point-to-point communication protocol** The local components running at $a$ of a communication protocol between $a$ and $b$ consist of two independent processes, one for handling high-level messages from $a$ to $b$ (the emitter, written $E_a^b$), and one for handling high-level messages from $b$ to $a$ (the receiver, written $R_a^b$). The process $E_a^b$ exports $post_a^b$, for handling outgoing high-level messages from $a$ to $b$, and uses the channel $send_a$, for sending low-level messages. Conversely, the process $R_a^b$ exports $recv_a^b$, for receiving low-level messages, and uses the channel *deliver* for delivering incoming messages to $a$. (For more elaborate protocols, we would supply each end of the protocol with both input and output capabilities, so that receivers may also issue control messages toward emitters.)

**Multiplexing on remote addresses** The lower part of the protocol stack simply uses the alleged address $b^+$ of an incoming message to select the relevant reception protocol $R_a^b$. Self-addressed messages are discarded, in order to avoid livelocks.

**Distribution of public keys to peers** This name-service subprotocol organizes the dissemination of public keys. It is grafted just above $E_a^b$ and $R_a^b$. It mostly consists of a table $D_B$ that associates with every known remote principal $b$ the names $recv_a^b, post_a^b$ exported by the running protocol for communication with $b$. The first rule of the definition $D_{ns}$ intercepts messages of the form $`ns'.c^+$ delivered by $R_a^b$, then it looks up the protocol interface for $c$. If $c^+$ was not in $D_B$ beforehand, this lookup will register $c^+$ and start two new processes $E_a^c$ and $R_a^c$. Finally, there are messages $port\langle post_a^b \rangle$ and $addr\langle b^+ \rangle$ for every registered remote principal $b$; the second rule of the definition $D_{ns}$ nondeterministically selects two such messages, $port\langle post_a^b \rangle$ and $addr\langle c^+ \rangle$, and sends $c^+$ to $b$ through $post_a^b$. With these definitions, for every pair of translated principals $a$ and $b$, there is a possibility that the implementation of $a$ sends a message to the implementation of $b$.

$$
\begin{array}{lcl}
\mathcal{F}^{a:X}_{B:S}(\,\cdot\,) & \stackrel{\text{def}}{=} & \left(\begin{array}{l} \mathsf{def}_{a^+,a^-,\widehat{X}}\ \mathsf{keys}\ a^+, a^- \wedge \bigwedge_{x\in X}\mathsf{fresh}\ \widehat{x}\ \mathsf{in} \\ \mathsf{def}_{recv_a}\ D_a \wedge D_B \wedge D_{ns} \wedge \bigwedge_{\tau\in\Sigma} D_\tau\ \mathsf{in}\,(\,\cdot\,) \mid \prod_{b\in B}(E^b_a \mid R^b_a \mid P^b_a) \end{array}\right) \setminus X \cup \{recv^b_a, post^b_a \mid b \in B\} \\
D_{y:\tau} & \stackrel{\text{def}}{=} & y\langle \_, v_2, \ldots, v_n\rangle \triangleright m_\tau\langle y, v_2, \ldots, v_n\rangle \\
D_\tau & \stackrel{\text{def}}{=} & \begin{array}{ll} & \mathsf{assoc}\ \ \{(x, [\![x]\!]) \mid x \in S_\tau \cup X_\tau\}, \\ & \quad t_\tau(x) = \mathsf{def\ fresh}\ j\ \mathsf{in\ enter}\ x, a^+.j.(j.\tau)_{a^-}, \\ & \quad t'_\tau(w) = \mathsf{def}\ D_{y:\tau}\ \mathsf{in\ enter}\ y, w \\ \wedge & \bigwedge_{y\in S_\tau} D_{y:\tau} \\ \wedge & m_\tau\langle x, y_2, \ldots, y_n\rangle \triangleright \\ & \quad \mathsf{let}\ b^+.w = t_\tau(x)\ \mathsf{in\ let}\ recv^b_a, post^b_a = t(b^+)\ \mathsf{in} \\ & \quad \mathsf{let}\ v_2 = t_{\tau_2}(y_2)\ \mathsf{in}\ \ldots \mathsf{let}\ v_n = t_{\tau_n}(y_n)\ \mathsf{in} \\ & \quad post^b_a\langle \tau.w.v_2.\cdots.v_n\rangle \\ \wedge & deliver\langle b^+, `\tau'.(j.s).(b^+_2.j_2.s_2).\cdots.(b^+_n.j_n.s_n)\rangle \triangleright \\ & \quad \mathsf{check}\ s\ \mathsf{authenticates}\ j.\tau\ \mathsf{using}\ a^+\ \mathsf{in\ let}\ x = t'_\tau(a^+.j.s)\ \mathsf{in} \\ & \quad \mathsf{check}\ s_2\ \mathsf{authenticates}\ j_2.\tau_2\ \mathsf{using}\ b^+_2\ \mathsf{in\ let}\ y_2 = t'_{\tau_2}(b^+_2.j_2.s_2)\ \mathsf{in} \\ & \quad \vdots \\ & \quad \mathsf{check}\ s_n\ \mathsf{authenticates}\ j_n.\tau_n\ \mathsf{using}\ b^+_n\ \mathsf{in\ let}\ y_n = t'_{\tau_n}(b^+_n.j_n.s_n)\ \mathsf{in} \\ & \quad x\langle b^+, y_2, \ldots, y_n\rangle \end{array} \\
D_{ns} & \stackrel{\text{def}}{=} & \begin{array}{ll} & deliver\langle b^+, `ns'.c^+\rangle \triangleright \mathsf{let}\ recv^c_a, post^c_a = t(c^+)\ \mathsf{in}\ 0 \\ \wedge & port\langle post\rangle \mid addr\langle c^+\rangle \triangleright port\langle post\rangle \mid addr\langle c^+\rangle \mid post\langle ns.c^+\rangle \end{array} \\
P^b_a & \stackrel{\text{def}}{=} & port\langle post^b_a\rangle \mid addr\langle b^+\rangle \\
D_B & \stackrel{\text{def}}{=} & \begin{array}{l} \mathsf{assoc}\ \{(b^+, recv^b_a, post^b_a) \mid b \in B\} \cup \{(a^+, drop_a, drop_a)\}, \\ \quad t(b^+) = (E^b_a \mid R^b_a \mid P^b_a \mid \mathsf{enter}\ b^+, recv^b_a, post^b_a) \setminus \{recv^b_a, post^b_a\} \end{array} \\
D_a & \stackrel{\text{def}}{=} & \begin{array}{ll} & recv_a\langle b^+, v\rangle \triangleright \mathsf{let}\ recv^b_a, post^b_a = t(b^+)\ \mathsf{in}\ recv^b_a\langle v\rangle \\ \wedge & drop_a\langle v\rangle \triangleright 0 \end{array}
\end{array}
$$

In $D_{ns}$, $ns$ is a $\mathsf{BitString}$ constant distinct from any channel type representation $\tau$. In $D_\tau$ and $D_{y:\tau}$, $\tau$ abbreviates the channel type $\langle \mathsf{Principal}, \tau_2, \ldots, \tau_n\rangle$. The definition $\mathsf{assoc}\ \{(x, [\![x]\!]) \mid \ldots\},\ t_\tau = \ldots,\ t'_\tau = \ldots$ introduces a two-way association table with current content $\{(x, [\![x]\!]) \mid \ldots\}$ and with lookup functions $t_\tau$ and $t'_\tau$. There are no lookup functions $t_{\mathsf{Principal}}$ and $t'_{\mathsf{Principal}}$. If $\tau_i = \mathsf{Principal}$ in $D_\tau$, then in the rule for $m_\tau$ we omit the lookup of $v_i$ and output $y_i$ instead; and in the rule for $deliver$ we input $y_i$ instead of $b^+_i.j_i.s_i$, so omit the check of $s_i$ and the lookup of $y_i$.

Figure 7: Protocol stack for $a$'s communications: implementation.

This part of our model is rather abstract; it could be refined to save some bandwidth, for instance by giving a low priority to these background messages. On the other hand, it seems hard to model a more realistic name-server that would resist all traffic-analysis attacks.

## 5 Point-to-point communication protocols

Point-to-point communication protocols are responsible for communicating messages reliably from one location to another. A single instance of a protocol can convey all traffic from one location to another, independently of how many high-level channels might be in use; there is no need to fork new instances as names are created or exported (cf. [2, 3]). Here we focus on two particular protocols; we omit a more general treatment of correctness conditions for protocols.

It is common for authentication protocols to protect against replay attacks, for example by the use of nonces. Our protocols integrate protection against high-level replay attacks that are seldom considered explicitly in the authentication literature; they avoid a situation where an emitter believes incorrectly that an application message was lost, and resends it with the consequence that the message is accepted twice. Our protocols are designed so that each message is delivered exactly once to its intended recipient.

Robust authentication protocols incorporate precautions of several other sorts (e.g., [5, 7]). In particular, a principle of explicitness helps in avoiding confusions between messages or sessions [5]. In one of our protocols (shown in Figure 9), for example, this principle dictates that a signed message should include the address of its intended recipient, even if this address may (incorrectly) be deemed evident from context. Another helpful principle is "sign before encrypting" [7]. It is also relevant for the protocol of Figure 9.

In our protocols, we use both public-key cryptography and shared-key cryptography (for session keys). Using pure public-key cryptography, message contents are signed and encrypted by the sender for the receiver. With a key shared by two parties, on the other hand, encryption suffices to ensure both authenticity and secrecy. One drawback of shared keys is that they must be established, then kept on both sides, so both parties must maintain some state.

Next we present two small protocols. These protocols have the same structure. They use replication to withstand interceptions, and simple sequence numbers to prevent replay attacks. In both protocols, the emitter and the receiver maintain a counter, whose value tags every message. The emitter increments the counter when a message is posted;

$$
\begin{array}{lll}
E_a^b & \stackrel{\text{def}}{=} & \mathsf{def}_{k_a^b, post_a^b} \quad \mathsf{key}\ k_a^b \\
& & \qquad \wedge \quad state\langle i\rangle \mid post_a^b\langle v\rangle \triangleright state\langle i+1\rangle \mid \mathsf{repl}\ send_a\langle b^+, \{\_.i.v\}_{k_a^b}\rangle \\
& & \mathsf{in}\ state\langle 0\rangle
\end{array}
$$

$$
\begin{array}{lll}
R_b^a & \stackrel{\text{def}}{=} & \mathsf{def}_{recv_b^a} \quad recv_b^a\langle m\rangle \triangleright \mathsf{decrypt}\ m\ \mathsf{using}\ k_a^b\ \mathsf{to}\ \_.i.v\ \mathsf{in}\ krecv_b^a\langle i, v\rangle \\
& & \qquad \wedge \quad state\langle i\rangle \mid krecv_b^a\langle i', v\rangle \triangleright \\
& & \qquad\qquad \mathsf{if}\ i' = i\ \mathsf{then}\ state\langle i+1\rangle \mid deliver\langle a^+, v\rangle\ \mathsf{else}\ state\langle i\rangle \\
& & \mathsf{in}\ state\langle 0\rangle
\end{array}
$$

Figure 8: A protocol with a shared key.

$$
\begin{array}{lll}
E_a^b & \stackrel{\text{def}}{=} & \mathsf{def}_{post_a^b} \quad state\langle i\rangle \mid post_a^b\langle v\rangle \triangleright state\langle i+1\rangle \mid \mathsf{repl}\ send_a\langle b^+, \{\_.i.v.(b^+.i.v)_{a^-}\}_{b^+}\rangle \\
& & \mathsf{in}\ state\langle 0\rangle
\end{array}
$$

$$
\begin{array}{lll}
R_b^a & \stackrel{\text{def}}{=} & \mathsf{def}_{recv_b^a} \quad recv_b^a\langle m\rangle \triangleright \mathsf{decrypt}\ m\ \mathsf{using}\ b^-\ \mathsf{to}\ \_.i.v.s\ \mathsf{in} \\
& & \qquad\qquad \mathsf{check}\ s\ \mathsf{authenticates}\ b^+.i.v\ \mathsf{using}\ a^+\ \mathsf{in}\ pkrecv_b^a\langle i, v\rangle \\
& & \qquad \wedge \quad state\langle i\rangle \mid pkrecv_b^a\langle i', v\rangle \triangleright \\
& & \qquad\qquad \mathsf{if}\ i' = i\ \mathsf{then}\ state\langle i+1\rangle \mid deliver\langle a^+, v\rangle\ \mathsf{else}\ state\langle i\rangle \\
& & \mathsf{in}\ state\langle 0\rangle
\end{array}
$$

Figure 9: A protocol with public-key cryptography.

the receiver checks it and increments it as the message is delivered.

- The first protocol is given in Figure 8. It relies on a shared key $k_a^b$. The emitter $E_a^b$ encrypts under this key. The receiver $R_b^a$ filters low-level incoming messages twice, first as messages encrypted under the key, then as messages with the expected sequence number.

- The second protocol is given in Figure 9. It relies on public-key cryptography. The emitter $E_a^b$ signs and encrypts its messages. The receiver $R_b^a$ decrypts each message, checks its signature, and finally checks its sequence number.

We have considered more substantial, realistic protocols that combine public-key and shared-key cryptography. They include, in particular, key-agreement exchanges, secure acknowledgments (so that message repetition is unnecessary), and techniques for minimizing state. The definitions of those protocols fit in our framework but are considerably more complex.

## 6 Correctness

In the statements of our results, we rely on a fairly standard notion of observational equivalence, $\approx$. For the high-level calculus, we let observational equivalence be the largest weak bisimulation on configurations that is closed by application of any well-formed configuration context and preserves barbs and extruded names. (As usual, a barb is the presence of any message on a channel whose name is free.) For the low-level calculus, we adopt the same definition, substituting "evaluation context" for "configuration context".

We also rely on a compatibility condition for low-level interfaces:

**Definition 3 (Compatible low-level interface)** *Let $C$ be a configuration of the high-level calculus. A* low-level interface compatible with $C$ *is a triple $A, X, (B_a)_{a\in A}$ that consists of the finite set $A \subset \mathcal{A}$ of principals defined in $C$, the finite set $X = \mathsf{xv}(C)$ of names exported by $C$, and a family $B_a \subset \mathcal{A} \setminus \{a\}$ of finite sets indexed by $A$ such that:*

1. *For any $b \in \mathcal{A}$ and $x \in \mathcal{N}_b$, if $b$ or $x$ appears in the process located at $a$ within $C$, then $b \in B_a$.*

2. *The undirected graph on $A$ with edges $\{(a, b) \mid b \in B_a\}$ is connected.*

The conditions on the sets $B_a$ guarantee that the filters initially bind the free names for every translated principal of $A$, and that the translated principals of $A$ may all know about one another by transitivity.

Our results relate configurations of principals to their translations, placed in a context that represents the network. They are full-abstraction properties. Roughly, they say that two configurations of the high-level calculus are observationally equivalently if and only if their translations are. (Roughly, $C \approx C'$ if and only if $[\![C]\!] \approx [\![C']\!]$.) We specify that the translations have access to a valid network $N$ by elaborating the observational equivalence in the low-level calculus. Nevertheless, we do not fix an attacker in the low-level calculus, or assume that the attacker is particularly benign: we quantify over $N$'s, and observational equivalence has a further, built-in quantification over arbitrary contexts. Crucially, we do not assume that every public key is associated with a trusted principal, and we translate each principal independently of its context. Thus, the attacker can use any free principal name, and any number of additional principals with values of its choice as public keys.

The following theorem is for the public-key protocol of Figure 9. We write $A^-$ for the set $\{a^- \mid a \in A\}$.

**Theorem 1** *Let $C$ and $C'$ be two high-level configurations with the same compatible low-level interface $A, X, (B_a)_{a \in A}$. Let $[\![ \cdot ]\!]$ be the translation using the public-key protocol, with parameter $(B_a)_{a \in A}$.*

*Then $C \approx C'$ if and only if, for every valid network $N$ restrained with respect to $A$ and $X$,*

$$(N \mid [\![C]\!]) \setminus A^- \approx (N \mid [\![C']\!]) \setminus A^-$$

Analogous theorems deal with other protocols, in particular with the shared-key protocol of Figure 8. Theorem 2 handles the case where the translations of two principals $a$ and $b$ have a shared key $k_a^b$ and use the shared-key protocol instead of the public-key protocol; for simplicity it does not treat the establishment of $k_a^b$. Since the network should not have access to the key $k_a^b$ exported by $E_a^b$, the statement of the theorem contains a restriction on a set of keys $K$ that includes $k_a^b$.

**Theorem 2** *Let $C$ and $C'$ be two high-level configurations with the same compatible low-level interface $A, X, (B_a)_{a \in A}$. Let $K$ be a subset of $\{k_a^b \mid (a,b) \in A^2, a \in B_b, b \in B_a\}$, and let $[\![ \cdot ]\!]$ be the translation using the shared-key protocol with key $k_a^b$ when $k_a^b \in K$ and the public-key protocol otherwise, with parameter $(B_a)_{a \in A}$.*

*Then $C \approx C'$ if and only if, for every valid network $N$ restrained with respect to $A$ and $X$,*

$$(N \mid ([\![C]\!] \setminus K)) \setminus A^- \approx (N \mid ([\![C']\!] \setminus K)) \setminus A^-$$

## 7 Related work and conclusion

There has been much work on the design and analysis of authentication protocols (e.g., [29, 18, 26, 23, 13, 9, 8, 21, 22, 4, 30, 20, 25]). Some of that work, like ours, relies on process calculi. There has also been significant work on the design of programmable systems with authentication (e.g., [10, 19, 33, 32]), but much less on the analysis of those systems. As this paper illustrates, process calculi provide a useful basis for important parts of that analysis.

As mentioned in the introduction, the correctness of authentication protocols is a notoriously subtle and challenging issue. The literature describes many attacks on authentication protocols (e.g., [5, 7]). Some of those expose true security flaws. Others, it has been argued, may lead to unexpected results but do not actually permit security breaches. (For example, if a key-establishment protocol can be derailed so that it terminates with two parties knowing different "shared" keys, then the attack can be discovered and neutralized when the two parties try to use the keys.) In such arguments, experts typically have in mind a particular context in which the protocols should be used. In our setting, the context is explicit: the protocols are part of the output of a compiler for high-level programs with a precise semantics. Therefore, we can define and prove correctness properties of our complete translation method, validating the protocols and also the rest of the method. Indeed, there is more to implementing authentication than designing an authentication protocol.

## Appendix

### Review of the join-calculus

This appendix contains a review of the join-calculus; much of it is borrowed verbatim from a previous paper [3]. This review does not cover the authentication constructs, which are the subject of the main body of the paper.

The join-calculus is a calculus of concurrent processes that communicate through named, one-directional channels [15]. It can express functional and imperative constructs, and constitutes the core of a distributed programming language [17, 14]. From a security perspective, we may say that the channels of the join-calculus have a strong secrecy property: only the process that creates a channel can receive messages on the channel. They also have a useful integrity property: for sending a message on a channel, it is necessary to have its name, which is an unforgeable capability. Any process that knows the name of a channel may transmit the name to other processes, possibly sending the name outside the lexical scope of its definition. In this important respect, the join-calculus resembles the pi-calculus [27]; it also resembles object-oriented languages where object references are capabilities for invoking methods.

Each channel has an associated arity—a fixed, integer size for the tuples passed on the channel. We require that names be used consistently in processes, respecting their arities, and enforce this requirement by adopting a type system. While there exists a rich, polymorphic type system for the join-calculus [16], a simple monomorphic type system suffices for our present purposes. We write $\langle \tau_1, \ldots, \tau_n \rangle$ for the type of channels that carry tuples with $n$ values of respective types $\tau_1, \ldots, \tau_n$, and restrict attention to types of this form. We allow types to be recursively defined (formally, using a fixpoint operator), so we may have for example $\tau = \langle \tau, \tau \rangle$. We assume that each name is associated with a type (although we usually keep this type implicit), and that there are infinitely many names for each type. Throughout, we consider only well-typed processes.

In the pure join-calculus, as we describe it here, names are used only as names of channels, and the set of values is defined to be the set of names. In extensions, names are included in a larger set of values. In any case, the contents of messages are values. We use lowercase identifiers $x$, $y$, *foo*, *bar*, ... to represent names, and $u$, $v$, ... to represent values.

Intuitively, the semantics of processes is as follows.

- $x\langle v_1, \ldots, v_n \rangle$ sends the tuple $v_1, \ldots, v_n$ on the channel named $x$. This message is asynchronous, in the sense that it does not require any form of handshake or acknowledgment.
- $\mathsf{def}_S\ D\ \mathsf{in}\ P$ is the process $P$ in the scope of the local definitions given in $D$, exporting the set of names $S$. (The set of names $S$ would not appear in the basic join-calculus, but here we are using an open variant [11].)
- $\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$ tests whether $u = v$, and then runs the process $P$ or the process $Q$ depending on the result of the test.
- $P \mid Q$ is the parallel composition of the processes $P$ and $Q$.
- $\mathsf{0}$ is the null process, which does nothing.

A join-pattern is a non-empty list of message patterns, each of the form $x\langle y_1, \ldots, y_n \rangle$. The names $y_1, \ldots, y_n$ are bound, and should all be distinct. The name $x$ is also bound; intuitively, it is the name of a channel being defined. A join-pattern is much like a guard for a definition, in the sense that a definition $J \triangleright P$ says that the process $P$ may run when

$$\begin{array}{ll}
\text{ASSOC} & \mathsf{def}_S\ D \wedge \mathsf{assoc}\ U \uplus \{(v, v_1, \ldots, v_n)\},\ t(x) = T,\ t'(x_1) = T' \ \mathsf{in}\ Q \mid \mathsf{let}\ y_1, \ldots, y_n = t(v)\ \mathsf{in}\ P \\
& \quad \to \mathsf{def}_S\ D \wedge \mathsf{assoc}\ U \uplus \{(v, v_1, \ldots, v_n)\},\ t(x) = T,\ t'(x_1) = T' \ \mathsf{in}\ Q \mid P\{^{v_1}/_{y_1}, \ldots, {}^{v_n}/_{y_n}\} \\
\text{ASSOC'} & \mathsf{def}_S\ D \wedge \mathsf{assoc}\ U \uplus \{(v, v_1)\},\ t(x) = T,\ t'(x_1) = T' \ \mathsf{in}\ Q \mid \mathsf{let}\ y = t'(v_1)\ \mathsf{in}\ P \\
& \quad \to \mathsf{def}_S\ D \wedge \mathsf{assoc}\ U \uplus \{(v, v_1)\},\ t(x) = T,\ t'(x_1) = T' \ \mathsf{in}\ Q \mid P\{^{v}/_{y}\} \\
\text{ALLOC} & \mathsf{def}_S\ D \wedge \mathsf{assoc}\ U,\ t(x) = T,\ t'(x_1) = T' \ \mathsf{in}\ Q \mid \mathsf{let}\ y_1, \ldots, y_n = t(v)\ \mathsf{in}\ P \qquad \text{(See the text for side conditions.)} \\
& \quad \to \mathsf{def}_S\ D \wedge D'\sigma \wedge \mathsf{assoc}\ U \uplus \{(v, v_1\sigma, \ldots, v_n\sigma)\},\ t(x) = T,\ t'(x_1) = T' \ \mathsf{in}\ Q \mid Q'\sigma \mid P\{^{v_1\sigma}/_{y_1}, \ldots, {}^{v_n\sigma}/_{y_n}\} \\
\text{ALLOC'} & \mathsf{def}_S\ D \wedge \mathsf{assoc}\ U,\ t(x) = T,\ t'(x_1) = T' \ \mathsf{in}\ Q \mid \mathsf{let}\ y = t'(v_1)\ \mathsf{in}\ P \qquad \text{(See the text for side conditions.)} \\
& \quad \to \mathsf{def}_S\ D \wedge D'\sigma \wedge \mathsf{assoc}\ U \uplus \{(v\sigma, v_1)\},\ t(x) = T,\ t'(x_1) = T' \ \mathsf{in}\ Q \mid Q'\sigma \mid P\{^{v\sigma}/_{y}\}
\end{array}$$

Figure 10: Operational semantics of the auxiliary association tables for the low-level calculus.

there are messages that match the join-pattern $J$. (If there are messages that match the join-pattern $J$ several times, then as many instances of $P$ may run.) Next we explain the notion of matching through a few special cases.

- Let us consider first the case where $J$ is simply the join-pattern $x\langle y\rangle$. The join-pattern $J$ is matched when a message $v$ has been sent on $x$. When this happens, the message is consumed, and $P$ is run, with the actual argument $v$ substituted for the formal argument $y$. (Thus, $x\langle y\rangle \triangleright P$ is analogous to the definition of a function with name $x$, formal argument $y$, and body $P$.)
- In the more general case where $J$ is the join-pattern $x\langle y_1, \ldots, y_m\rangle$, we say that $J$ is matched when a tuple $v_1, \ldots, v_m$ has been sent on $x$ (with the same $m$). When this happens, the message is consumed, and $P$ is run, with the actual arguments $v_1, \ldots, v_m$ substituted for the formal arguments $y_1, \ldots, y_m$.
- Finally, in the case where $J$ is the join-pattern $x\langle y_1, \ldots, y_m\rangle \mid x'\langle y'_1, \ldots, y'_{m'}\rangle$, we say that $J$ is matched when there are messages on both of the channels $x$ and $x'$, and these messages have $m$ and $m'$ components, respectively. When this happens, the messages are consumed, and $P$ is run, with the actual arguments substituted for the corresponding formal arguments.

In addition to definitions of the form $J \triangleright P$, the grammar allows definitions of the form $D \wedge D'$. A definition $D \wedge D'$ is simply the conjunction of the definitions $D$ and $D'$. A conjunction like $x\langle y\rangle \triangleright P \wedge x\langle z\rangle \triangleright Q$, where the same defined name $x$ appears in two conjuncts, is legal; when there is a message $x\langle v\rangle$, either $P$ or $Q$ may run—the choice between them is non-deterministic.

## Association tables

Our filters use association tables to keep track of the correspondences between names and their wire formats, and between keys and protocol interfaces. These association tables are auxiliary data structures that can be encoded in the join-calculus and a fortiori in the sjoin-calculus. We omit their encoding, and describe only their interface.

- The definition $\mathsf{assoc}\ U,\ t(x) = T,\ t'(x_1) = T'$ introduces an association table with content $U$. This definition binds two lookup functions $t$ and $t'$, and attaches the processes $T$ and $T'$ to them; $t'$ is optional.
- The content $U$ is a finite set of tuples $(v, v_1, \ldots, v_n)$ where $v, v_1, \ldots, v_n$ are values; $n$ is fixed for the table, and $t'$ can be present only if $n = 1$.
- In a definition $\mathsf{assoc}\ U,\ t(x) = T,\ t'(x_1) = T'$ of an association table, the processes $T$ and $T'$ are both of the form $\mathsf{def}\ D\ \mathsf{in}\ Q \mid \mathsf{enter}\ v, v_1, \ldots, v_n$ (up to $\equiv$). The process $\mathsf{enter}\ v, v_1, \ldots, v_n$, which appears once in $T$ and once in $T'$, has the role of entering the association between $v$ and $v_1, \ldots, v_n$ in the table, adding the tuple $(v, v_1, \ldots, v_n)$ to $U$. In $T$, $v$ must be $x$ and must be free, and each of $v_1, \ldots, v_n$ must contain some name defined in $D$. Conversely, if $T'$ is present, $n$ must equal 1 and, in $T'$, $v_1$ must be $x_1$ and must be free, and $v$ must contain some name defined in $D$.
- The process $\mathsf{let}\ y_1, \ldots, y_n = t(v)\ \mathsf{in}\ P$ looks for a tuple associated with $v$ in $U$. If one is found, then $P$ is executed with the tuple substituted for $y_1, \ldots, y_n$. Otherwise, the process $T$ attached to $t$ is executed; it creates a tuple and enters it in the table, and $P$ is executed with the tuple substituted for $y_1, \ldots, y_n$.
- Similarly, the process $\mathsf{let}\ y = t'(v_1)\ \mathsf{in}\ P'$ looks for a value associated with $v_1$, and creates one if required. In any case $P'$ is executed with that value substituted for $y$.
- The construct $\mathsf{let}\ V = t(v)\ \mathsf{in}\ P$, which also does pattern-matching on the ($\mathsf{BitString}$) value returned by the table, is just shorthand for

  $$\mathsf{def}\ \kappa\langle V\rangle \triangleright P\ \mathsf{in}\ \mathsf{let}\ y = t(v)\ \mathsf{in}\ \kappa\langle y\rangle$$

  where $\kappa$ is a fresh name.

The operational semantics of association tables is given by the four rules of Figure 10. (Simpler versions apply when $D$ is missing; we do not need them.) ASSOC represents the case where the value $v$ in the query $t(v)$ already appears in a tuple $(v, v_1, \ldots, v_n)$ in the table; then $v_1, \ldots, v_n$ are substituted for the formal parameters $y_1, \ldots, y_n$ in $P$. ALLOC represents the case where no tuple is associated with $v$ and a new tuple is added to the table. It has the side conditions that the process $T$ attached to $t(x)$ is $\mathsf{def}\ D'\ \mathsf{in}\ Q' \mid \mathsf{enter}\ x, v_1, \ldots, v_n$ (up to $\equiv$), that there is no tuple in $U$ whose first element is $v$, that $\sigma$ is a substitution that maps $x$ to $v$ and names in $\mathsf{dv}(D')$ to fresh and distinct names. ASSOC and ALLOC apply when $t'(x_1) = T'$ is omitted. ASSOC' and ALLOC' only apply for tables with $n = 1$; the side conditions of ALLOC' are similar to those of ALLOC.

## References

[1] Martín Abadi. Protection in programming-language translations. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 868–883, July 1998.

[2] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 105–116, June 1998.

[3] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure communications processing for distributed languages. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 74–88, May 1999.

[4] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1), January 1999. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.

[5] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[6] Roberto M. Amadio. On modelling mobility. To appear in Theoretical Computer Science, 1998.

[7] Ross Anderson and Roger Needham. Robustness principles for public key protocols. In *Proceedings of Crypto '95*, pages 236–247, 1995.

[8] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 419–428, May 1998.

[9] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Advances in Cryptology—CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer Verlag, August 1993.

[10] Andrew D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.

[11] Michele Boreale, Cédric Fournet, and Cosimo Laneve. Bisimulations in the join-calculus. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98)*, pages 68–86. Chapman and Hall, June 1998.

[12] Gérard Boudol, Ilaria Castellani, Matthew Hennessy, and Astrid Kiehn. A theory of processes with localities. *Formal Aspects of Computing*, 6:165–200, 1994.

[13] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, volume 23, 5, pages 1–13, December 1989.

[14] Cédric Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, Palaiseau, November 1998.

[15] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96*, pages 372–385. ACM, January 1996.

[16] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ML for the join-calculus. In Antoni Mazurkiewicz and Jòzef Winkowski, editors, *Proceedings of the 8th International Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 196–212. Springer-Verlag, July 1997.

[17] Cédric Fournet and Luc Maranget. The join-calculus language (version 1.03). Source distribution and documentation available from `http://join.inria.fr/`, June 1997.

[18] Richard A. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, May 1989.

[19] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[20] Pat Lincoln, John Mitchell, Mark Mitchell, and Andre Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, pages 112–121, November 1998.

[21] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996.

[22] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 31–43, 1997.

[23] Catherine Meadows. A system for the specification and analysis of key management protocols. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 182–195, 1991.

[24] Catherine Meadows. Panel on languages for formal specification of security protocols. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, page 96, 1997.

[25] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[26] Jonathan K. Millen, Sidney C. Clark, and Sheryl B. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, SE-13(2):274–288, February 1987.

[27] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, September 1992.

[28] James H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, January 1973.

[29] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[30] Steve Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, September 1998.

[31] Peter Sewell and Jan Vitek. Secure composition of insecure components. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 136–150, 1999.

[32] Sun Microsystems, Inc. RMI enhancements. Web pages at `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/`, 1997.

[33] Leendert van Doorn, Martín Abadi, Mike Burrows, and Edward Wobber. Secure network objects. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 211–221, May 1996.