Secure Communications Processing for Distributed Languages

Martín Abadi
ma@pa.dec.com
Systems Research Center
Compaq

Cédric Fournet
fournet@microsoft.com
Microsoft Research

Georges Gonthier*
Georges.Gonthier@inria.fr
INRIA Rocquencourt

Abstract

Communications processing is an important part of distributed language systems with facilities such as RPC (remote procedure call) and RMI (remote method invocation). For security, messages may require cryptographic operations in addition to ordinary marshaling. We investigate a method for wrapping communications processing around an entity with secure local communication, such as a single machine or a protected network. The wrapping extends security properties of local communication to distributed communication. We formulate and analyze the method within a process calculus.

1 Cryptography and distributed languages

Because security in distributed systems often relies on cryptography, much research has been devoted to the intrinsic properties of cryptographic algorithms and protocols. Although these properties are necessary and interesting, they do not provide a complete account of the use of cryptography; they focus on mechanisms out of context, rather than on the intended or actual higher-level effect of those mechanisms. For example, protocol specifications and analyses often discuss the creation and exchange of cryptographic keys but not the security properties of the higher-level communication that relies on these keys (e.g., [13, 8, 30, 28]).

Much sophisticated machinery complements cryptographic algorithms and protocols, forming complex, delicate systems (cf. [7]). This machinery helps bridge the gap between local computation and distributed computation. In addition to network services (for example, for naming), this machinery performs communications processing, which includes conversions between internal data representations and network data representations (marshaling and unmarshaling) and cryptographic operations. Communications processing effectively maps application program interfaces (APIs) for secure communication to particular mechanisms. A sound cryptographic basis is necessary but not sufficient for the security of communications processing.

This paper investigates a method for wrapping communications processing (both marshaling and cryptographic operations) around an entity with secure local communication, such as a single machine or a protected network. It focuses on a generic wrapper that operates much as a firewall with an encrypting tunnel (e.g., [14, 6, 2]): the wrapper relays messages to and from a public network, applying cryptography and converting message formats through appropriate marshaling and unmarshaling procedures. The messages may convey new capabilities for communication. The cryptography is encapsulated in protocols that we treat largely as black boxes. Assuming that these protocols are correct, we argue that our method is correct by establishing some of its properties within a process calculus.

Communications processing is an important part of distributed language systems with facilities such as RPC (remote procedure call) [12] and RMI (remote method invocation) [10, 32]. Like our method, such systems include marshaling and often rely on cryptography for security [11, 21, 31, 29]. However, the specifics of our method are apparently new, and so is the formal precision with which we are able to define it and analyze it. Since our aim is to provide a foundation for secure distributed language systems (and not a popular artifact), we can subordinate compatibility and efficiency to generality and correctness.

A recently published companion to this paper [4] gives an account of a simple API for secure communication and of a cryptographic implementation of this API. The API is embodied in a minimal programming language with primitive secure channels, the join-

*Partly supported by ESPRIT CONFER-2 WG-21836.

calculus [16, 17, 18, 19]. The implementation is itself expressed as a translation from the join-calculus to an extension of the join-calculus with constructs for public-key encryption, called the sjoin-calculus. The present paper provides an alternative translation from the join-calculus to the sjoin-calculus (suggested briefly in [4]): the wrapping is applied to a join-calculus process and yields a sjoin-calculus process. This alternative translation has the distinguishing feature that it requires encryption only for messages that cross trust boundaries. (The join-calculus and the sjoin-calculus are analogous to the pi-calculus [26] and the spi-calculus [5], respectively, but better suited for our purposes [1, 3].)

Sections 2 and 4 review the join-calculus and the sjoin-calculus, respectively. Section 3 introduces an example. Section 5 describes the communication facilities that our sjoin-calculus processes use. Sections 6 and 7, which are the core of this paper, present and analyze our method for wrapping a join-calculus process and obtaining a sjoin-calculus process. Section 8 revisits the example of section 3. Finally, section 9 concludes. An appendix contains some technical definitions. This paper assumes some ease with formal notation but avoids many additional definitions and all proofs, which appear in the companion paper and in a longer manuscript [3].

2 The join-calculus (review)

The join-calculus is a calculus of concurrent processes that communicate through named, one-directional channels [16]. It can express functional and imperative constructs, and constitutes the core of a distributed programming language [19, 15]. 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 [26]; 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 [18], a simple monomorphic type system suffices for our present purposes. We write $\langle \sigma_1, \dots, \sigma_n \rangle$ for the type of channels that carry tuples with n values of respective types $\sigma_1, \dots, \sigma_n$, 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 $\sigma = \langle \sigma, \sigma \rangle$. 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. Further, we write $\widetilde{v}$ for a tuple $v_1, v_2, \dots, v_n$.

In addition to a category of names, the syntax of the join-calculus includes categories of processes, definitions, and join-patterns. Processes, definitions, and join-patterns are defined recursively in Figure 1. The operator $\mid$ has highest precedence, so for example $\mathsf{def}\ D\ \mathsf{in}\ P \mid P'$ means $\mathsf{def}\ D\ \mathsf{in}\ (P \mid P')$.

Intuitively, the semantics of processes is as follows.

- $x\langle \widetilde{v} \rangle$ sends the tuple of values $\widetilde{v}$ on the channel named x. This message is asynchronous, in the sense that it does not require any form of handshake or acknowledgment.
- $\mathsf{def}\ D\ \mathsf{in}\ P$ is the process P in the scope of the local definitions given in D.
- $\mathsf{if}\ v = v'\ \mathsf{then}\ P\ \mathsf{else}\ P'$ tests whether $v = v'$, and then runs the process P or the process P' depending on the result of the test.
- $P \mid P'$ is the parallel composition of the processes P and P'.
- $\mathsf{0}$ is the null process, which does nothing.

A join-pattern is a non-empty list of message patterns, each of the form $x\langle y_1, \dots, y_n \rangle$. The names $y_1, \dots, y_n$ 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 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.

v	$::=$		values
		x	name
P	$::=$		processes
		$x\langle\widetilde{v}\rangle$	message
	$\mid$	$\mathsf{def}\ D\ \mathsf{in}\ P$	local definition
	$\mid$	$\mathsf{if}\ v = v'\ \mathsf{then}\ P\ \mathsf{else}\ P'$	comparison
	$\mid$	$P \mid P'$	parallel composition
	$\mid$	$\mathsf{0}$	null process
D	$::=$		definitions
		$J \triangleright P$	reaction rule
	$\mid$	$D \wedge D'$	conjunction of definitions
J	$::=$		join-patterns
		$x\langle\widetilde{y}\rangle$	message pattern
	$\mid$	$J \mid J'$	join of patterns

Figure 1: Grammar of the join-calculus.

- 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 nondeterministic.

Definitions obey lexical scoping rules. In particular, given a process $\mathsf{def}\ D\ \mathsf{in}\ P$, a channel name defined in D is recursively bound in the whole of $\mathsf{def}\ D\ \mathsf{in}\ P$, including D.

As a small example, we consider the process:

$$\mathsf{def}\ x\langle y\rangle \triangleright (\mathsf{if}\ y = u\ \mathsf{then}\ z\langle\rangle\ \mathsf{else}\ \mathsf{0})\ \mathsf{in}\ x\langle u\rangle$$

The definition $x\langle y\rangle \triangleright (\ldots)$ introduces the channel x and causes an empty message on z in reaction to the message u on x; the body $x\langle u\rangle$ simply sends the message u on x. Intuitively, we may care both about the integrity of the message on x (so that a message on z is triggered) and about its secrecy. In other words, we may want that the process behave like $z\langle\rangle$, which sends a message on z and does not reveal u, no matter what an attacker does. The following equivalence expresses this property:

$$\mathsf{def}\ x\langle y\rangle \triangleright (\mathsf{if}\ y = u\ \mathsf{then}\ z\langle\rangle\ \mathsf{else}\ \mathsf{0})\ \mathsf{in}\ x\langle u\rangle \quad \approx \quad z\langle\rangle$$

We say that two processes P_1 and P_2 are equivalent, and write $P_1 \approx P_2$, when no context can distinguish one from the other. Intuitively, we may think of the context of P_1 and P_2 as an attacker; then $P_1 \approx P_2$ means that an attacker cannot cause P_1 and P_2 to behave in ways that would enable the attacker to distinguish one from the other [5]. Thus, $P_1 \approx P_2$ implies both an integrity property (limiting the effect of the attacker) and a secrecy property (limiting the observations of the attacker).

The appendix gives more precise definitions of the semantics of processes and of the equivalence relation.

3 An example

We describe a larger example in some detail, both as explanation of the join-calculus and as additional mo-

$$
\begin{array}{rclcl}
C[\cdot] & \stackrel{\text{def}}{=} & \mathsf{def}\ \ \mathit{entry}\langle a,n\rangle \mid \mathit{open}\langle\rangle & \triangleright & a\langle n+\text{“ won”}\rangle \mid \mathit{closed}\langle n\rangle \\
 & & \wedge\ \ \mathit{entry}\langle a',n'\rangle \mid \mathit{closed}\langle n\rangle & \triangleright & a'\langle n+\text{“ won”}\rangle \mid \mathit{closed}\langle n\rangle\ \mathsf{in}\ [\cdot] \\
C_i[\cdot] & \stackrel{\text{def}}{=} & \mathsf{def}\ \ \mathit{entry}_i\langle a,n\rangle \mid \mathit{open}_i\langle\rangle & \triangleright & \mathit{entry}\langle a,n\rangle \mid \mathit{entry}\langle \mathit{closed}_i,n\rangle \\
 & & \wedge\ \ \mathit{entry}_i\langle a',n'\rangle \mid \mathit{closed}_i\langle s\rangle & \triangleright & a'\langle s\rangle \mid \mathit{closed}_i\langle s\rangle\ \mathsf{in}\ [\cdot]
\end{array}
$$

Figure 2: Some contexts for the contest example.

tivation for our work. Our example concerns a contest. In its simplest variant, the contest relies on a server that creates a channel and publishes its name; the participant whose entry arrives first on this channel is the winner. At the level of abstraction of the join-calculus, it is relatively simple to understand the contest, to state its properties, and to consider variants, in great part because the join-calculus description of the contest hides challenging aspects of distributed communication (for example, cryptographic protection). We address some of those aspects in later sections.

For convenience in this example, we use strings as though they were primitive in the join-calculus, writing + for string concatenation. Strings could easily be added to the join-calculus, but they can also be encoded through standard methods. As usual, a context is a process with a hole; if $C[\cdot]$ is a context, then $C[P]$ is the result of filling its hole with P.

The context $C[\cdot]$ of Figure 2 essentially defines the contest. The state of the contest is described by placing a process in $C[\cdot]$, for example as in:

$$C[\mathit{open}\langle\rangle \mid \mathit{plug}\langle \mathit{entry}\rangle]$$

and as in:

$$C[\mathit{closed}\langle \mathit{Alice}\rangle \mid \mathit{plug}\langle \mathit{entry}\rangle]$$

The channel *entry* is the one on which participants should send their entries. The channel *open* is local to the server; the presence of a message on this channel signifies that the contest is open. Similarly, the channel *closed* serves to represent that the contest is closed. (It is common to represent local state in this manner in process calculi.) The channel *plug* is free, and its only purpose is the publication of *entry*. The process $\mathit{open}\langle\rangle \mid \mathit{plug}\langle \mathit{entry}\rangle$ emits an empty message on the channel *open* and emits the name *entry* on the channel *plug*. When the server receives a first pair a, n on *entry*, it replies on a, with the string $n+$“ *won*”. Thus, a is used as a continuation address; n is arbitrary and represents the identity of the participant that sent the message. The server records n locally by producing the message $\mathit{closed}\langle n\rangle$ while consuming the message $\mathit{open}\langle\rangle$. Later, whenever the server receives another pair a', n' on *entry*, it replies on a' with the same string $n+$“ *won*”, thus telling the new participant that an entry with the name n had priority. Thus, $C[\mathit{open}\langle\rangle \mid \mathit{plug}\langle \mathit{entry}\rangle]$ represents the starting state of the contest while $C[\mathit{closed}\langle \mathit{Alice}\rangle \mid \mathit{plug}\langle \mathit{entry}\rangle]$ represents a state where *Alice* has won.

In this example, an attacker cannot intercept or modify the entries in the contest or the server's replies. For instance, an attacker cannot modify the name n of the winner and cannot learn the name n' of a loser. We present some of these security properties, expressing them as equivalences; they follow from the scoping rules of the join-calculus and can be proved easily through standard bisimulation techniques [25].

- Only a participant with access to *entry* may win the contest. The following equivalence concerns the extreme case where in fact only one participant has access to *entry*:

$$
\begin{array}{rl}
 & C[\mathit{open}\langle\rangle \mid \mathit{entry}\langle a, \mathit{Alice}\rangle] \\
\approx & C[\mathit{closed}\langle \mathit{Alice}\rangle \mid a\langle \mathit{Alice}+\text{“ won”}\rangle]
\end{array}
$$

- The outcome of the contest cannot be affected once the contest is closed:

$$
\begin{array}{rl}
 & C[\mathit{closed}\langle \mathit{Alice}\rangle \mid \mathit{plug}\langle \mathit{entry}\rangle] \\
\approx & \left(\begin{array}{l} \mathsf{def}\ \mathit{entry}\langle a,n\rangle \triangleright a\langle \mathit{Alice}+\text{“ won”}\rangle\ \mathsf{in} \\ \mathit{plug}\langle \mathit{entry}\rangle \end{array}\right)
\end{array}
$$

  The latter process behaves monotonously, just like the contest after closing.

- A loser remains anonymous with respect to other participants:

$$
\begin{array}{rl}
 & C[\mathit{closed}\langle \mathit{Alice}\rangle \mid \mathit{plug}\langle \mathit{entry}\rangle \mid \mathit{entry}\langle a', \mathit{Bob}\rangle] \\
\approx & C[\mathit{closed}\langle \mathit{Alice}\rangle \mid \mathit{plug}\langle \mathit{entry}\rangle \mid \mathit{entry}\langle a', \mathit{Pat}\rangle]
\end{array}
$$

  This equivalence means that the outer context cannot distinguish an entry that contains *Bob* and an entry that contains *Pat*. The two processes being compared are in fact equivalent to:

$$C[\mathit{closed}\langle \mathit{Alice}\rangle \mid \mathit{plug}\langle \mathit{entry}\rangle \mid a'\langle \mathit{Alice}+\text{“ won”}\rangle]$$

  which is independent of *Bob* and *Pat*.

v	$::=$		values
		x	name
	$\mid$	$\{\widetilde{v}\}_v$	encryption
P	$::=$		processes
		...	as for the join-calculus
	$\mid$	$\mathsf{decrypt}\ v\ \mathsf{using}\ v'\ \mathsf{to}\ \widetilde{x}\ \mathsf{in}\ P\ \mathsf{else}\ P'$	decryption
D	$::=$		definitions
		...	as for the join-calculus
	$\mid$	$\mathsf{fresh}\ x$	fresh name
	$\mid$	$\mathsf{keys}\ x^+, x^-$	fresh pair of keys
J	$::=$		join-patterns
		...	as for the join-calculus

Figure 3: Grammar of the sjoin-calculus.

The contest server may spread its work over auxiliary components (proxies), as follows. Each of the proxies makes available a distinct channel $entry_i$ for entries, forwards the first entry that it receives to the server, waits for an answer, and remembers this answer so that it can immediately return it in response to later entries. The context $C_i[\cdot]$ of Figure 2 describes proxy i. The main difference between a proxy and the server is that, when it receives its first entry $entry_i\langle a, n\rangle$, a proxy cannot decide by itself whether this entry is the winning one. The proxy leaves this decision to the server by submitting two entries with the same name n but with different continuation addresses. The message $entry\langle a, n\rangle$ will lead to a direct response on a to participant n; in effect, this message extends the capability to send messages on a to the server. On the other hand, the message $entry\langle closed_i, n\rangle$ will result in an identical response on $closed_i$ to the proxy. The proxy remembers this response, so it can handle subsequent entries without consulting the server.

Combining the contexts $C[\cdot]$ and $C_i[\cdot]$, we can assemble a system with a main server and two proxies:

$$C\left[\begin{array}{ll} & open\langle\rangle \\ \mid & C_1[open_1\langle\rangle \mid plug_1\langle entry_1\rangle] \\ \mid & C_2[open_2\langle\rangle \mid plug_2\langle entry_2\rangle] \end{array}\right]$$

This system shares the essential security properties of the system without proxies (for instance, the anonymity of losers). The question of whether the proxies may misbehave does not arise: since we have their code, we can verify them.

Intuitively, the main server, the proxies, and the participants may be located at different sites. A distributed implementation of the contest may therefore rely on some amount of cryptography. Obviously, the implementation could protect the channels between the sites with cryptographic protocols. In addition, the implementation could represent channel names as unguessable capabilities, so only the processes that obtain a continuation address as the result of legitimate communication can send messages to the address. Like other high-level programming languages, the join-calculus abstracts away these difficult details of distributed communication. We reconsider these details in section 8.

4 The sjoin-calculus (review)

The sjoin-calculus (named by analogy with the spi-calculus [5]) is an extension of the join-calculus with constructs for encryption and decryption, and with names that can be used not only for channels but also as keys, nonces, or other tags. It relies on a simple black-box model of cryptographic operations (cf. [22]). In this model, neither cleartext nor encryption key can be extracted from a ciphertext without knowledge of the corresponding decryption key; knowledge of this key reveals the cleartext, which contains sufficient redundancy so that successful decryption is evident. The grammar of the sjoin-calculus is given in Figure 3.

The set of values includes not only names but also ciphertexts of the form $\{\widetilde{v}\}_v$. In the ciphertext $\{\widetilde{v}\}_v$, the subscript v is the encryption key and the tuple $\widetilde{v}$ is the cleartext.

Ciphertexts can be compared, as for example in the process $\mathsf{if}\ \{v\}_x = \{v'\}_x\ \mathsf{then}\ P\ \mathsf{else}\ P'$. This comparison would enable a recipient of $\{v\}_x$ with knowledge of x and v' to deduce whether v equals v'.

The syntax of processes includes a new decryption form. The process $\mathsf{decrypt}\ v\ \mathsf{using}\ v'\ \mathsf{to}\ \widetilde{x}\ \mathsf{in}\ P\ \mathsf{else}\ P'$ attempts to decrypt v using v' as decryption key. If this decryption succeeds, then P runs, with the results of the decryption substituted for $\widetilde{x}$; if the decryption fails, then P' runs. For example, if y^+ is the name of an encryption key and y^- is the name of the inverse decryption key, then the process $\mathsf{decrypt}\ \{z\}_{y^+}\ \mathsf{using}\ y^-\ \mathsf{to}\ x\ \mathsf{in}\ P\ \mathsf{else}\ P'$ will decrypt $\{z\}_{y^+}$ using y^- and will run P with z substituted for x.

The syntax of definitions includes constructs for introducing names and pairs of keys. The construct $\mathsf{fresh}\ x$ introduces the fresh name x. The construct $\mathsf{keys}\ x^+, x^-$ introduces the names of keys x^+ and x^-; the key x^+ is an encryption key and x^- is the inverse decryption key. Note that x^+ and x^- range over ordinary names: the symbols $+$ and $-$ are used only conventionally.

Given this context-free grammar, one can write silly expressions, for example $\mathsf{def}\ \mathsf{keys}\ x^+, x^-\ \mathsf{in}\ x^+\langle y \rangle$ which uses a key x^+ as a channel. A straightforward extension of the type system of the join-calculus excludes such expressions. This extension consists in adding a basic type $\mathsf{BitString}$, with the rules that the names introduced by the constructs $\mathsf{fresh}\ x$ and $\mathsf{keys}\ x^+, x^-$ are of type $\mathsf{BitString}$, and that encryption and decryption operations apply only to arguments of type $\mathsf{BitString}$ and yield results of type $\mathsf{BitString}$.

5 Low-level communication

Next we introduce some definitions that enable us to discuss encrypted communication on a public network in the context of the sjoin-calculus.

Our informal assumptions about the public network are mostly standard in the literature on protocols [27]. We assume an asynchronous network, where messages may be lost or intercepted by an intruder; the intruder may also modify, duplicate, or inject messages. In networks without any default traffic, an intruder that sees an encrypted message may deduce significant information simply from the appearance of the message, even if it cannot decrypt it. Therefore, we require that the network has enough "noise" to prevent traffic analysis attacks. (Alternatively, if this assumption is not quite met, we can qualify our results to allow for the possibility of such attacks.)

We model the network interface available to a process P as a pair of channels, *emit* and *recv* (of types $\langle \mathsf{BitString} \rangle$ and $\langle \langle \mathsf{BitString} \rangle \rangle$, respectively). Using these channels, P can send and receive messages of type $\mathsf{BitString}$, as follows.

- For output, P sends its message on *emit*.
- For input, P sends a continuation channel κ on *recv*. The network will then return a single message on κ. The message may not be intended for P, and it may even be unintelligible to P, so P may need to do some filtering and retrying.

We write $\mathcal{E}nv\,[\,\cdot\,]$ for a context that defines the channels *emit* and *recv*, publishes their names, and produces the required "noise". Intuitively, $\mathcal{E}nv\,[\,P\,]$ represents a situation where both P and any process running in parallel may use *emit* and *recv*. We omit the precise definition of $\mathcal{E}nv\,[\,\cdot\,]$, which is not necessary for this paper and appears in [4].

On top of the public network, we assume a protocol for transmitting a tuple $\widetilde{v}$ of values using a key pair x^+, x^-. The protocol may require multiple messages; it consists of two sjoin-calculus processes $E_x[\widetilde{v}]$ and R_x.

- Using the key x^+ for encryption, $E_x[\widetilde{v}]$ sends $\widetilde{v}$.
- Using the key x^- for decryption, R_x receives messages, then forwards their cleartext contents on an auxiliary, internal channel x°.

Intuitively, the protocol should guarantee the integrity and secrecy of $\widetilde{v}$. In [4], we give a precise correctness criterion for such a protocol. Moreover, relying on nonces and confounders, we give two correct protocols. The definitions and results of this paper assume only that we use a correct protocol, but its choice does not matter.

6 The wrapping

Any correct protocol for sending a single message is the basis of a compositional translation from the join-calculus to the sjoin-calculus [4]. That translation replaces every join-calculus communication step with an execution of the protocol. It has a direct inductive definition, where for example the parallel composition of two processes is mapped to the parallel composition of their respective translations. That translation does not make any assumptions about the distribution of the process being translated. Intuitively, every message could be exposed on a hostile network by crossing machine boundaries or Intranet boundaries. Because encryption is applied uniformly and abundantly, the security properties of the translation do not depend on any notion of security perimeter (of "inside" and "outside").

An implementation of the join-calculus that does not assume anything about distribution may be elegant, but it may also be rather wasteful: it requires encryption even for messages internal to a protected site. In this paper we describe an implementation that avoids encryption in those cases. Instead of being defined inductively, the implementation of a process P is simply P put in a suitable context. This context serves as a wrapper or filter. It does not disturb messages internal to P. It does however catch messages that cross the boundary of P, adding or removing encryption, and translating contents (marshaling and unmarshaling). For example, a message that would leave P on a channel x gets turned into a $\mathsf{BitString}$ message encrypted with a corresponding key x^+; if the message contains the channel name v, then v gets replaced with the corresponding key v^+. Conversely, when a message arrives at the filter encrypted with a key y^+ and the filter has the key y^-, the contents of the message are translated and relayed on a channel y.

This filter resembles a firewall with an encrypting tunnel, so we refer to it as a firewall (a little abusively perhaps). From the programmer's point of view, this filter serves as a run-time system for distributed communication that takes care of encryption and decryption and of marshaling and unmarshaling.

6.1 Data structures

Our firewall keeps track of the correspondence between encryption keys on the outside and channel names on the inside of a process. This correspondence is recorded in association tables. 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 $\mathsf{assoc}\ S,\ t(x) = T,\ t'(x^+) = T'$ introduces an association table with contents S. This definition binds two lookup functions t and t', and attaches the processes T and T' to them.

- The contents S is a finite set of pairs (x, v) where x is a channel name and v is a value of type $\mathsf{BitString}$.

- In a definition $\mathsf{assoc}\ S,\ t(x) = T,\ t'(x^+) = T'$ of an association table, the processes T and T' are both of the form $\mathsf{def}\ D\ \mathsf{in}\ Q \mid \mathsf{enter}\ x, x^+$. The process $\mathsf{enter}\ x, x^+$, which appears once in T and once in T', has the role of entering the association between x and x^+ in the table, adding the pair (x, x^+) to S. In T, x^+ is defined in D while x is free. In T', conversely, x is defined in D while x^+ is free.

- The process $\mathsf{let}\ x^+ = t(v)\ \mathsf{in}\ P$ looks for a key associated with v in S. If one is found, then P is executed with the key substituted for x^+. Otherwise, the process T attached to t is executed; it creates a key and enters the association between v and this key. In parallel, P is executed with the key substituted for x^+.

- Similarly, the process $\mathsf{let}\ x = t'(v')\ \mathsf{in}\ P'$ looks for a channel name associated with v' in S and, if one is not found, creates a channel name. In any case P' is executed with a channel name associated with v' substituted for x.

6.2 The dynamics of filtering

The interface between a process and its environment can evolve as they exchange new names in messages. To capture this dynamic aspect of communication, we draw on techniques developed in earlier work on the join-calculus [16]: mutually recursive definitions filter all messages and unfold new filters for their arguments. Some complications arise because the join-calculus features several types of channels. Accordingly, we maintain a channel-key association table for every type that may traverse the firewall.

Assume that S is a finite set of names and that Σ is a set of channel types that contains all the types of names in S and that is closed under decomposition (that is, if $\langle \sigma_1, \ldots, \sigma_n \rangle \in \Sigma$ then $\sigma_1, \ldots, \sigma_n \in \Sigma$). For every $\sigma \in \Sigma$, let S_σ be the set of names of type σ in S.

The definition $D_\Sigma(S)$ creates an association table for every type σ in Σ, with an entry for each name in S_σ. This definition is given in Figure 4, top-down. Throughout, we associate three names x^+, x^-, and x° with each channel name x. We may write $D_{x \triangleright E_x}$ and $D_{x^\circ \triangleright x}$ for $D_{\sigma,\, x \triangleright E_x}$ and $D_{\sigma,\, x^\circ \triangleright x}$, respectively, when σ is unimportant or clear from context.

In our construction of a firewall in section 6.3, we use a definition $D_\Sigma(S)$ for intercepting and forwarding messages, performing conversions between channel names and keys, encrypting and decrypting, and unfolding new filters for processing future messages. We explain $D_\Sigma(S)$ further by distinguishing two ways in which entries appear in association tables at the firewall:

- Suppose that the entry (x, x^+) has been registered in an association table at the firewall from the inside, by a call to $t_\sigma(x)$. This call would have allocated a new pair of keys x^+, x^-, unfolded a definition $D_{\sigma,\, x^\circ \triangleright x}$, and forked a process R_x.

  Whenever a process outside the firewall knows the key x^+ and produces a message on x by running the process $E_x[y_1^+, \ldots, y_n^+]$ on the public network, the process R_x first performs a decryption with x^-, then delivers the message $x^\circ\langle y_1^+, \ldots, y_n^+\rangle$ to a second stage.

  In this second stage, the effect of receiving y_1^+, $\ldots, y_n^+$ on x° is described in $D_{\sigma,\, x^\circ \triangleright x}$. According to $D_{\sigma,\, x^\circ \triangleright x}$, the encryption keys re-

$$
\begin{array}{rcl}
D_\Sigma(S) & \stackrel{\text{def}}{=} & \bigwedge_{\sigma\in\Sigma} \mathsf{assoc}\ \{(x,x^+) \mid x \in S_\sigma\},\ t_\sigma(x) = T_\sigma,\ t'_\sigma(x^+) = T'_\sigma \\
T_\sigma & \stackrel{\text{def}}{=} & \mathsf{def}\ D_{\sigma,\, x^\circ \triangleright x} \wedge \mathsf{keys}\ x^+, x^-\ \mathsf{in}\ R_x \mid \mathsf{enter}\ x, x^+ \\
T'_\sigma & \stackrel{\text{def}}{=} & \mathsf{def}\ D_{\sigma,\, x \triangleright E_x}\ \mathsf{in}\ \mathsf{enter}\ x, x^+ \\
D_{\langle\sigma_1,\ldots,\sigma_n\rangle,\, x^\circ \triangleright x} & \stackrel{\text{def}}{=} & \begin{array}{l} x^\circ\langle y_1^+,\ldots,y_n^+\rangle \triangleright \\ \quad \mathsf{let}\ y_1 = t'_{\sigma_1}(y_1^+)\ \mathsf{in}\ \ldots \mathsf{let}\ y_n = t'_{\sigma_n}(y_n^+)\ \mathsf{in} \\ \quad x\langle y_1,\ldots,y_n\rangle \end{array} \\
D_{\langle\sigma_1,\ldots,\sigma_n\rangle,\, x \triangleright E_x} & \stackrel{\text{def}}{=} & \begin{array}{l} x\langle y_1,\ldots,y_n\rangle \triangleright \\ \quad \mathsf{let}\ y_1^+ = t_{\sigma_1}(y_1)\ \mathsf{in}\ \ldots \mathsf{let}\ y_n^+ = t_{\sigma_n}(y_n)\ \mathsf{in} \\ \quad E_x[y_1^+,\ldots,y_n^+] \end{array}
\end{array}
$$

Figure 4: Filter definitions.

ceived $(y_1^+,\ldots,y_n^+)$ are mapped to channel names $(y_1,\ldots,y_n)$. For this purpose, the firewall performs a lookup for every key y_i^+ in the association table corresponding to the expected type for y_i. Once every name y_i has been obtained, the plain message $x\langle y_1,\ldots,y_n\rangle$ is delivered to some definition inside the firewall.

In case a key y_i^+ is not present in the corresponding association table, the allocator T'_{σ_i} registers a fresh name y_i so that future messages sent on y_i inside the firewall are encrypted and forwarded outside using the key y_i^+. Note that, as a result of this method, the same key y_i^+ may eventually appear in the association tables for several different types, associated with a different channel name in each of these tables.

- In the other direction, suppose that the entry (x,x^+) has been registered in an association table at the firewall from the outside, by a call to $t'_\sigma(x^+)$. This call would have given rise to a definition $D_{\sigma,\, x \triangleright E_x}$ that binds x.

  Whenever a message $x\langle y_1,\ldots,y_n\rangle$ is emitted within the firewall, first every channel name y_i is replaced with an encryption key y_i^+ (possibly extending the association tables if these names are crossing the firewall for the first time), then the resulting contents is encrypted by the sending process $E_x[y_1^+,\ldots,y_n^+]$.

The definition of $D_\Sigma(S)$ extends smoothly to variants of the join-calculus with basic types (such as the types of integers and strings). Instead of including an association table in $D_\Sigma(S)$ for each basic type σ, we assume given a bijective function t_σ for marshaling values from σ to BitString and its inverse t'_σ for unmarshaling values from BitString to σ.

6.3 Putting a process behind a firewall

Next we show how to construct a firewall for a process. The firewall will depend only on the output interface of the process, that is, on the free names of the process. We bind all those free names in the firewall, and register them associating each of them with a free encryption key. When the process exports other names, as it runs, the firewall evolves too. Thus, we guarantee that the firewall intercepts every message between the process and its environment. We put a process P behind a firewall simply by placing it in a context $\mathcal{F}_S[\,\cdot\,]$, where S is a finite set that includes all the free names of P. The context $\mathcal{F}_S[\,\cdot\,]$ is defined by:

$$\mathcal{F}_S[P] \quad \stackrel{\text{def}}{=} \quad \mathsf{def}\ D_\Sigma(S) \wedge \bigwedge_{x\in S} D_{x \triangleright E_x}\ \mathsf{in}\ P$$

where Σ is the smallest set of types that contains the types of all the names in S and that is closed under decomposition. Every well-typed join-calculus process uses only a finite number of types [15], so Σ is finite, so the context $\mathcal{F}_S[\,\cdot\,]$ is well-defined. Without loss of generality, we can assume that the lookup functions t_σ, t'_σ are not free in P.

For each name $x \in S$ of type σ, the firewall of $\mathcal{F}_S[P]$ has an entry (x,x^+) in the table for σ and a definition $D_{x \triangleright E_x}$. This definition takes outputs from P on x, marshals them, and relays them using encryption with the key x^+. In the course of computation, though, P may export the names of channels defined in P, for input. Accordingly, it is sometimes useful to describe the state of a firewall that exports encryption keys that correspond to those names.

We therefore generalize the definition of the firewall context as follows. Given a process $P \stackrel{\text{def}}{=} \mathsf{def}\ D\ \mathsf{in}\ Q$, assume that S and X are two finite disjoint sets of names such that S includes all the free names of P and

X includes only names defined in D. We let:

$$\begin{array}{rcl}\mathcal{F}_S^X[D,\ Q] & \stackrel{\text{def}}{=} & \mathsf{def}\ D_\Sigma(X \cup S) \\ & & \quad \wedge\ \bigwedge_{x \in X} D_{x^\circ \triangleright x} \wedge \bigwedge_{x \in S} D_{x \triangleright E_x} \\ & & \quad \wedge\ D\ \mathsf{in} \\ & & Q \mid \prod_{x \in X} R_x\end{array}$$

where Σ is the smallest set of types that contains the types of all the names in $X \cup S$ and that is closed under decomposition, and where $\prod_{x \in X} R_x$ is the parallel composition of all processes R_x for $x \in X$.

Intuitively, S is the output interface of P while X is its input interface. This input interface consists of names in D that have been exported. For every such name x, the firewall has an entry (x, x^+) in an association table and a definition $D_{x \triangleright E_x}$. In addition, the firewall forks a process R_x for receiving messages encrypted with x^+.

The firewall is well-typed and provides a well-typed interface for the process that it encloses. Its only interface with the environment consists of the names *emit* and *recv* and of values of type $\mathsf{BitString}$. It does not make any further assumptions about the types used by the environment. The environment may even overload one of its keys, using it as though it represented several channels of different types and causing it to appear in several association tables at the firewall, without a security compromise. This property of the firewall is essential: security should not depend on a typing assumption about the environment, since the environment could well violate such an assumption.

7 Analysis

In this section we analyze the method defined in section 6. Our main theorems show that wrapping preserves equivalences between processes and that it is compositional (up to equivalence).

7.1 Behavior through a firewall

Intuitively, we may think of $\mathcal{F}_S[P]$ as an implementation of P. Technically, this intuition is not quite satisfactory, particularly if we adopt a standard concept of implementation according to which Q implements P if and only if every possible behavior of Q is a possible behavior of P (e.g., [20]).

1. This intuition is not accurate, because $\mathcal{F}_S[P]$ may behave in ways in which P could not behave. In particular, $\mathcal{F}_S[P]$ typically sends messages on the channels *emit* and *recv* even when these names are not present in P.
2. This intuition is not quite sufficient, because it does not imply that the proposed implementation shares the security properties of P: the implementation relation need not preserve those properties (e.g., [23, 24]).

Security properties can sometimes be phrased as equivalences (as in section 3), so the second objection suggests that we phrase our result in terms of equivalences rather than in terms of implementation relations. The first objection suggests that we should take the context $\mathcal{E}nv\,[\,\cdot\,]$ into account and that we should not directly equate $\mathcal{F}_S[P]$ and P. We arrive at the following first theorem:

Theorem 1 *For all join-calculus processes P and Q, and every finite set of names S that includes all the free names of P and Q,*

$$P \approx Q \quad \textit{if and only if} \quad \mathcal{E}nv\,[\,\mathcal{F}_S[P]\,] \approx \mathcal{E}nv\,[\,\mathcal{F}_S[Q]\,]$$

In programming-language terminology, this theorem is a full-abstraction result. From a security perspective, full abstraction means that attackers have the same power at the two levels of abstraction being compared [1]. It helps transfer properties established for a high-level process to its low-level counterpart, simplifying arguments about the latter. In this case, however, the "only if" direction of full abstraction is rather trivial, as it follows from basic congruence properties. The "if" direction remains important: it shows that, although $\mathcal{F}_S[P]$ and P may behave differently, the behavior of $\mathcal{F}_S[P]$ corresponds bijectively to the behavior of P.

Despite the benefits of putting a process behind a firewall, a firewall is not a guarantee of security when the process may behave improperly as a result of incompetence or malice. Suppose that x is a channel name free in P, that y is bound in P, and that P includes y in a message on x. In $\mathcal{F}_S[P]$, the firewall will translate P's message relaying a key y^+ instead of y to the outside. Perhaps y was intended for internal use only, so messages on y may be trusted in P somehow (wrongly). Then an attack against $\mathcal{F}_S[P]$ may be able to exploit the knowledge of y^+, since the firewall will decrypt any message encrypted with y^+ and relay its contents on y. In this situation, the firewall does not help, but it does not hurt either (as Theorem 1 indicates). By giving up some transparency, the firewall could be supplemented with restrictions for auditing or preventing the escape of keys like y^+.

7.2 Multiple firewalls

Further theorems address issues of distribution by analyzing systems with multiple firewalls. The join-calculus can be refined to account for computations distributed over several sites [17]. For our purposes, distribution simply means a fairly arbitrary partition of processes and definitions. Thus, when we write:

$$\mathsf{def}\ D\ \mathsf{in}\ P \mid Q$$

we may intend that the different parts of this expression (D, P, Q) be executed at several sites interconnected by a public network—although we do not rely on any special syntax to indicate this in the join-calculus. This section shows that, following our method, we can wrap a firewall around the code for each site, making the site boundaries explicit. Each firewall communicates with its outside (including the firewalls for other sites) through a poor interface: the public network, where traffic is unprotected and limited to type $\mathsf{BitString}$. Moreover, we do not assume a particular low-level cryptographic protocol (given by $E_x[\widetilde{v}]$ and R_x) or a particular higher-level application protocol for D, P, and Q. Nevertheless, the distribution across sites preserves security properties.

Section 6.3 shows how to enclose a single process within a firewall. More generally, firewalls can be placed around any part of a composite system. For example, we can write:

$$\mathcal{E}nv\,[\,\mathcal{F}_S[P] \mid \mathcal{F}_S[Q]\,]$$

representing two separate processes P and Q, each with its own firewall, on a public network modeled as the environment $\mathcal{E}nv\,[\,\cdot\,]$. Each firewall maintains its own association tables; the two firewalls may map an encryption key to different local names of channels. We have the following compositionality property:

Theorem 2 *For all join-calculus processes P and Q, and every finite set of names S that includes all the free names of P and Q,*

$$\mathcal{E}nv\,[\,\mathcal{F}_S[P \mid Q]\,] \quad \approx \quad \mathcal{E}nv\,[\,\mathcal{F}_S[P] \mid \mathcal{F}_S[Q]\,]$$

This property means that we can partition the process $P \mid Q$ into two components P and Q, locate these components at different sites with their own firewalls, let them communicate with each other and with the environment through the firewalls, and be sure that the environment will not be able to detect any difference with the situation where $P \mid Q$ is at a single site with one firewall.

Another compositionality property deals with the distribution of definitions; it is analogous but more complex and more general.

Theorem 3 *For all join-calculus processes P and Q, for all join-calculus definition D, let X be the set of names that are free in P and defined in D. For every finite set S disjoint from X that includes all the free names of* $\mathsf{def}\ D\ \mathsf{in}\ P \,|\, Q$,

$$\begin{array}{ll} & \mathcal{E}nv\,[\,\mathcal{F}_S[\mathsf{def}\ D\ \mathsf{in}\ P \,|\, Q]\,] \\ \approx & \mathcal{E}nv \left[\begin{array}{l} \mathsf{def}\ \bigwedge_{x \in X} \mathsf{keys}\ x^+, x^-\ \mathsf{in} \\ \mathcal{F}_{X \cup S}[P] \mid \mathcal{F}^X_S[D,\ Q] \end{array} \right] \end{array}$$

This property means that we can partition the process $\mathsf{def}\ D\ \mathsf{in}\ P \,|\, Q$ into two components and still guarantee that communication on channels defined in D proceeds much as though D, P, and Q were all at the same site. One component consists of D and Q; communication between D and Q is local. The names in X represent the input interface of this component. The other component consists of P. A definition introduces a key pair x^+, x^- for each name $x \in X$: the firewalls for the two sites can communicate using x^+, x^- but neither x^+ nor x^- is available to the outside initially.

When P sends a message on a channel $x \in X$, the message traverses the two firewalls, undergoing first encryption with x^+ and then decryption with x^-. If the message contains another name $y \in X \cup S$, this name is converted to a key y^+ and back to a channel name as it traverses the firewalls. If the message contains a name z that P is exporting for the first time, a new key z^+ appears and is successively recorded at both firewalls; this key is associated with z at the firewall for P and with a new name at the firewall for D and Q. The result is the establishment of a secure communication path in the reverse direction, from D and Q back to P. As other channel names are exchanged, the components keep communicating just as they would do if they were at the same site. The firewalls are transparent, and they extend transparently whenever a new channel name traverses them.

8 An example, revisited

In order to illustrate the use of our method, we revisit the example of section 3 and describe two implementations of the contest server defined there.

First we discuss the effect of wrapping a firewall around the contest server in its initial state, which is described by the process:

$$C[\mathit{open}\langle\rangle \mid \mathit{plug}\langle \mathit{entry}\rangle]$$

This process has a single free name, *plug*, but its type is quite complicated. We explain this type bottom-up, writing String for the basic type of strings. The type of *a* and *closed* is $\langle\mathsf{String}\rangle$, so the type of *entry* is $\langle\langle\mathsf{String}\rangle, \mathsf{String}\rangle$ and the type of *plug* is

$$\mathcal{E}nv\left[\mathcal{F}_{\{plug_1,plug_2\}}\left[\mathsf{def}\ D\ \mathsf{in}\ \begin{array}{rl} & open\langle\rangle \\ | & C_1[open_1\langle\rangle \mid plug_1\langle entry_1\rangle] \\ | & C_2[open_2\langle\rangle \mid plug_2\langle entry_2\rangle] \end{array}\right]\right]$$

$$\approx\ \mathcal{E}nv\left[\begin{array}{l} \mathsf{def\ keys}\ entry^+, entry^-\ \mathsf{in} \\ \quad \mathcal{F}^{\{entry\}}_{\emptyset}[D,\ open\langle\rangle] \\ |\quad \mathcal{F}_{\{plug_1,entry\}}[C_1[open_1\langle\rangle \mid plug_1\langle entry_1\rangle]] \\ |\quad \mathcal{F}_{\{plug_2,entry\}}[C_2[open_2\langle\rangle \mid plug_2\langle entry_2\rangle]] \end{array}\right]$$

Figure 5: An equivalence in the contest example.

$\langle\langle\langle\mathsf{String}\rangle,\mathsf{String}\rangle\rangle$. In order to protect the contest server, our firewall contains a table for each of the three types in the following set Σ:

$$\{\langle\mathsf{String}\rangle, \langle\langle\mathsf{String}\rangle,\mathsf{String}\rangle, \langle\langle\langle\mathsf{String}\rangle,\mathsf{String}\rangle\rangle\}$$

An implementation of the contest server with a single site is described by:

$$I \stackrel{\text{def}}{=} \mathcal{E}nv\left[\mathcal{F}_{\{plug\}}[C[open\langle\rangle \mid plug\langle entry\rangle]]\right]$$

Initially the firewall contains only the pair ($plug$, $plug^+$) in the table for $\langle\langle\langle\mathsf{String}\rangle,\mathsf{String}\rangle\rangle$. As a result, the message $plug\langle entry\rangle$ is intercepted, the contents $entry$ is marshaled, a fresh pair of keys $entry^+, entry^-$ is stored in the table for $\langle\langle\mathsf{String}\rangle,\mathsf{String}\rangle$, and two processes are forked: $E_{plug}[entry^+]$ which encrypts and sends $entry^+$, and R_{entry} which awaits messages encrypted with $entry^+$. The evolution of I is summarized in the following equivalence:

$$I\ \approx\ \mathcal{E}nv\left[\begin{array}{l}\mathsf{def\ keys}\ entry^+, entry^-\ \mathsf{in} \\ E_{plug}[entry^+] \mid \mathcal{F}^{\{entry\}}_{\{plug\}}[D,\ open\langle\rangle]\end{array}\right]$$

where D is the definition such that $C[\cdot] = \mathsf{def}\ D\ \mathsf{in}\ [\cdot]$. The environment can then obtain the encryption key $entry^+$, typically but not necessarily by using a process R_{plug}, and hence can participate in the contest. Later, whenever a new entry is received by R_{entry} and unmarshaled, it provides a key for encrypting a response; a local continuation channel of type $\langle\mathsf{String}\rangle$ is allocated for this response. The size of the table for $\langle\mathsf{String}\rangle$ may grow each time an entry is processed.

Similarly, we can wrap three-table firewalls around all the processes of section 3. By applying Theorem 1 to the security properties stated in section 3, we obtain corresponding implementation-level security properties.

As a second implementation of the contest server, we consider the variant with proxies. This variant can naturally be mapped to several sites, each protected by a firewall of the kind described above. Applying Theorem 3 and Theorem 2 we obtain the equivalence of Figure 5, which expresses that the distribution of proxies is transparent. The top process describes a centralized system where the contest server and the proxies are all at a single site protected by a firewall. The bottom process describes a distributed implementation with three components protected by their own firewalls. The two processes have different structure, but participants in the contest cannot distinguish them. More importantly, perhaps, sjoin-calculus attackers cannot distinguish these two processes either—so, in our model, communication between the proxies and the server over the public network does not enable attacks.

9 Conclusions

In this paper we define a method that combines local communication with encrypted communication on public channels and we study its security. The method is largely independent of the nature of the channels and of the programs at their endpoints. Those endpoints require no modification, only a new layer with marshaling and cryptographic operations and with tables that dynamically associate channels with keys. This transparency property has obvious benefits, but it also has a cost—simpler and cheaper specialized methods probably exist for particular applications. In addition, our method does not include important optimizations such as the multiplexing of application channels on node-to-node channels.

Our approach relies on a precise model of a distributed language, assumes the use of adequate cryptographic algorithms, and must be complemented with appropriate high-level security policies. In return, it yields precise definitions and guarantees. We believe that employing the join-calculus and similar formalisms is an effective way to achieve this degree of precision.

Appendix

This appendix reviews the scoping rules, the operational semantics, and the definition of equivalence ($\approx$) for the sjoin-calculus (omitting the treatment of association tables, given in [3]). The corresponding definitions for the join-calculus fragment are analogous.

Scopes

First, in Figure 6, we define the sets of free names ($\mathsf{fv}[v]$, $\mathsf{fv}[P]$, and $\mathsf{fv}[D]$), defined names ($\mathsf{dv}[J]$ and $\mathsf{dv}[D]$), and received names ($\mathsf{rv}[J]$), for values, processes, join-patterns, and definitions. A name is fresh with respect to an expression or set of expressions when it does not occur free in them. We write $\{^v/_x\}$ for the substitution of the value v for the name x, write $\{^{\widetilde{v}}/_{\widetilde{x}}\}$ for the substitution of the values $v_1, \ldots, v_n$ for the names $x_1, \ldots, x_n$ when $\widetilde{v} = v_1, \ldots, v_n$ and $\widetilde{x} = x_1, \ldots, x_n$, and let σ range over arbitrary substitutions. We usually identify expressions up to renaming of bound names, assuming implicit α-conversion in order to avoid name clashes. We require that, in every join-pattern, all names be distinct. We also require that each name be defined in at most one clause of the form $\mathsf{keys}\ x, y$ or $\mathsf{fresh}\ z$.

Operational semantics

We present our operational semantics in chemical style [9], as a variant of the reflexive chemical abstract machine [16]. The state of a computation is represented by a pair of multisets $(\mathcal{D}, \mathcal{P})$, called a *chemical solution*, and written $\mathcal{D} \vdash \mathcal{P}$, where:

- $\mathcal{P}$ is a multiset of processes, intuitively the processes running;
- $\mathcal{D}$ is a multiset of definitions.

The rules for computation operate on chemical solutions. They form two families, *structural rules* and *reduction rules*.

Structural rules are reversible, and express the syntactic rearrangements of expressions in a solution. We write them in the form $\mathcal{D}_1 \vdash \mathcal{P}_1 \rightleftharpoons \mathcal{D}_2 \vdash \mathcal{P}_2$, where $\rightharpoonup$ represents "heating" and $\leftharpoondown$ represents "cooling". We usually omit the parts of $\mathcal{D}_1$, $\mathcal{P}_1$, $\mathcal{D}_2$, and $\mathcal{P}_2$ that are the same on both sides of $\rightleftharpoons$. With this abbreviation convention, we adopt the following structural rules for the sjoin-calculus:

$$
\begin{array}{lrcl}
\textsc{Str-Null} & \vdash \mathsf{0} & \rightleftharpoons & \vdash \\
\textsc{Str-Par} & \vdash P_1 \mid P_2 & \rightleftharpoons & \vdash P_1, P_2 \\
\textsc{Str-And} & D_1 \wedge D_2 \vdash & \rightleftharpoons & D_1, D_2 \vdash \\
\textsc{Str-Def} & \vdash \mathsf{def}\ D\ \mathsf{in}\ P & \rightleftharpoons & D\sigma_{\mathrm{dv}} \vdash P\sigma_{\mathrm{dv}}
\end{array}
$$

with the side condition for Str-Def that $dom(\sigma_{\mathrm{dv}}) = \mathsf{dv}[D]$ and that $range(\sigma_{\mathrm{dv}})$ consists of fresh and distinct names. The first three rules state that $\mid$ and $\wedge$ are associative and commutative, with unit $\mathsf{0}$ for $\mid$. The rule Str-Def describes the introduction of new names and reaction rules in a solution; its side condition ensures that we follow a static scoping discipline.

As an example, we rewrite the rule Str-Def without our abbreviation convention. It becomes: for every process $\mathsf{def}\ D\ \mathsf{in}\ P$ and multisets $\mathcal{D}$ and $\mathcal{P}$, for every substitution σ_{dv} that maps $\mathsf{dv}[D]$ to pairwise distinct names not in $\mathsf{fv}[\mathcal{P}] \cup \mathsf{fv}[\mathcal{D}] \cup \mathsf{fv}[\mathsf{def}\ D\ \mathsf{in}\ P]$,

$$\mathcal{D} \vdash \mathcal{P} \cup \{\mathsf{def}\ D\ \mathsf{in}\ P\} \ \rightleftharpoons\ \mathcal{D} \cup \{D\sigma_{\mathrm{dv}}\} \vdash \mathcal{P} \cup \{P\sigma_{\mathrm{dv}}\}$$

Reduction rules represent proper, basic computation steps. We write them in the form $\mathcal{D}_1 \vdash \mathcal{P}_1 \longrightarrow \mathcal{D}_2 \vdash \mathcal{P}_2$. We adopt the reduction rules of Figure 7 for the sjoin-calculus, with the following side conditions: in Red, that $dom(\sigma_{\mathrm{rv}}) = \mathsf{rv}[J]$; in the second clause of Compare, that $w \neq v$; in the first clause of Decrypt, that $\widetilde{v}$ and $\widetilde{y}$ are tuples of the same length; in the second clause of Decrypt, that w is not of the form $\{\widetilde{v}\}_{x^+}$ with $\widetilde{v}$ and $\widetilde{y}$ of the same length. The rule Red describes the use of a definition clause to consume messages and to produce a new instance of a guarded process. The two rules Compare concern the comparison of values. The two rules Decrypt concern attempts to decrypt values. Note that a process that attempts to decrypt with a non-key will get stuck, like $\mathsf{decrypt}\ w\ \mathsf{using}\ \{\}_{x^+}\ \mathsf{to}\ \widetilde{y}\ \mathsf{in}\ P\ \mathsf{else}\ P'$ which uses $\{\}_{x^+}$ instead of a key; however, this problem does not affect the processes that we construct, and it does not modify the capabilities of attackers.

Chemical semantics provides a concise way to define concurrent reduction modulo structural equivalence. This presentation, however, is equivalent to a more traditional presentation based on reduction on processes (instead of chemical solutions). For processes, we define the relations of *structural equivalence* and of *reduction modulo structural equivalence* by a combination of heating, reduction, and cooling of chemical solutions:

$$
\begin{array}{rcl}
P \equiv P' & \stackrel{\text{def}}{=} & \emptyset \vdash \{P\} \rightleftharpoons^* \emptyset \vdash \{P'\} \\
P \rightarrow P' & \stackrel{\text{def}}{=} & \emptyset \vdash \{P\} \rightleftharpoons^* \longrightarrow \rightleftharpoons^* \emptyset \vdash \{P'\}
\end{array}
$$

Equivalence

Evaluation contexts are the contexts generated by the following grammar:

$$C[\,\cdot\,] \quad ::= \quad [\,\cdot\,] \;\big|\; P \,|\, C[\,\cdot\,] \;\big|\; \mathsf{def}\ D\ \mathsf{in}\ C[\,\cdot\,]$$

For every process P and name x, $P \Downarrow_x$ holds if P may output on x, either immediately or after some reduction. (We omit a formal definition.)

$$
\begin{aligned}
\mathsf{fv}[x] &\stackrel{\text{def}}{=} \{x\} \\
\mathsf{fv}[\{v_1,\ldots,v_n\}_v] &\stackrel{\text{def}}{=} \mathsf{fv}[v] \cup \textstyle\bigcup_{i\in 1..n} \mathsf{fv}[v_i]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{fv}[x\langle v_1,\ldots,v_n\rangle] &\stackrel{\text{def}}{=} \{x\} \cup \textstyle\bigcup_{i\in 1..n} \mathsf{fv}[v_i] \\
\mathsf{fv}[\mathsf{if}\ v = v'\ \mathsf{then}\ P\ \mathsf{else}\ P'] &\stackrel{\text{def}}{=} \mathsf{fv}[P] \cup \mathsf{fv}[P'] \cup \mathsf{fv}[v] \cup \mathsf{fv}[v'] \\
\mathsf{fv}[\mathsf{decrypt}\ v\ \mathsf{using}\ v'\ \mathsf{to}\ \widetilde{x}\ \mathsf{in}\ P\ \mathsf{else}\ P'] &\stackrel{\text{def}}{=} (\mathsf{fv}[P] \setminus \{\widetilde{x}\}) \cup \mathsf{fv}[P'] \cup \mathsf{fv}[v] \cup \mathsf{fv}[v'] \\
\mathsf{fv}[\mathsf{def}\ D\ \mathsf{in}\ P] &\stackrel{\text{def}}{=} (\mathsf{fv}[P] \cup \mathsf{fv}[D]) \setminus \mathsf{dv}[D] \\
\mathsf{fv}[P \mid P'] &\stackrel{\text{def}}{=} \mathsf{fv}[P] \cup \mathsf{fv}[P'] \\
\mathsf{fv}[0] &\stackrel{\text{def}}{=} \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{rv}[x\langle y_1,\ldots,y_n\rangle] &\stackrel{\text{def}}{=} \{y_1,\ldots,y_n\} & \mathsf{dv}[x\langle y_1,\ldots,y_n\rangle] &\stackrel{\text{def}}{=} \{x\} \\
\mathsf{rv}[J \mid J'] &\stackrel{\text{def}}{=} \mathsf{rv}[J] \uplus \mathsf{rv}[J'] & \mathsf{dv}[J \mid J'] &\stackrel{\text{def}}{=} \mathsf{dv}[J] \uplus \mathsf{dv}[J']
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{fv}[J \triangleright P] &\stackrel{\text{def}}{=} \mathsf{dv}[J] \cup (\mathsf{fv}[P] \setminus \mathsf{rv}[J]) & \mathsf{dv}[J \triangleright P] &\stackrel{\text{def}}{=} \mathsf{dv}[J] \\
\mathsf{fv}[\mathsf{fresh}\ x] &\stackrel{\text{def}}{=} \{x\} & \mathsf{dv}[\mathsf{fresh}\ x] &\stackrel{\text{def}}{=} \{x\} \\
\mathsf{fv}[\mathsf{keys}\ x^+, x^-] &\stackrel{\text{def}}{=} \{x^+, x^-\} & \mathsf{dv}[\mathsf{keys}\ x^+, x^-] &\stackrel{\text{def}}{=} \{x^+, x^-\} \\
\mathsf{fv}[D \wedge D'] &\stackrel{\text{def}}{=} \mathsf{fv}[D] \cup \mathsf{fv}[D'] & \mathsf{dv}[D \wedge D'] &\stackrel{\text{def}}{=} \mathsf{dv}[D] \cup \mathsf{dv}[D']
\end{aligned}
$$

Figure 6: Scopes.

$$
\begin{array}{llcl}
\textsc{Red} & J \triangleright P \vdash J\sigma_{\mathrm{rv}} & \longrightarrow & J \triangleright P \vdash P\sigma_{\mathrm{rv}} \\[1ex]
\textsc{Compare} & \vdash \mathsf{if}\ v = v\ \mathsf{then}\ P\ \mathsf{else}\ P' & \longrightarrow & \vdash P \\
 & \vdash \mathsf{if}\ v = w\ \mathsf{then}\ P\ \mathsf{else}\ P' & \longrightarrow & \vdash P'
\end{array}
$$

$$
\begin{array}{llcl}
\textsc{Decrypt} & \mathsf{keys}\ x^+, x^- & \vdash & \mathsf{decrypt}\ \{\widetilde{v}\}_{x^+}\ \mathsf{using}\ x^-\ \mathsf{to}\ \widetilde{y}\ \mathsf{in}\ P\ \mathsf{else}\ P' \\
\longrightarrow & \mathsf{keys}\ x^+, x^- & \vdash & P\left\{{}^{\widetilde{v}}/_{\widetilde{y}}\right\} \\[1ex]
 & \mathsf{keys}\ x^+, x^- & \vdash & \mathsf{decrypt}\ w\ \mathsf{using}\ x^-\ \mathsf{to}\ \widetilde{y}\ \mathsf{in}\ P\ \mathsf{else}\ P' \\
\longrightarrow & \mathsf{keys}\ x^+, x^- & \vdash & P'
\end{array}
$$

Figure 7: Reduction rules.

Observational equivalence ($\approx$) is the largest symmetric relation $\mathcal{R}$ on processes such that $P \mathcal{R} Q$ implies that: (1) if $P \Downarrow_x$ then $Q \Downarrow_x$; (2) if $C[\cdot]$ is an evaluation context then $C[P] \mathcal{R} C[Q]$; (3) if $P \rightarrow P'$ then, for some Q', $P' \mathcal{R} Q'$ and $Q \rightarrow^* Q'$. (Our results hold also if $\approx$ denotes other equivalences, for example testing equivalence.)

References

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

[2] M. Abadi, A. Birrell, R. Stata, and E. Wobber. Secure web tunneling. *Computer Networks and ISDN Systems*, 30(1–7):531–539, Apr. 1998. Proceedings of the 7th International World Wide Web Conference.

[3] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. Manuscript, full version of [4] and this paper, on the Web at `http://join.inria.fr/`.

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

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

[6] K. F. Alden and E. P. Wobber. The AltaVista tunnel: Using the Internet to extend corporate networks. *Digital Technical Journal*, 9(2):5–15, Oct. 1997. On the Web at `http://www.digital.com/info/DTJQ01/DTJQ01HM.HTM`.

[7] R. Anderson. Why cryptosystems fail. In *1st ACM Conference on Computer and Communications Security*, pages 215–227, Nov. 1993.

[8] M. Bellare and P. Rogaway. Provably secure session key distribution: The three party case. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, 1995.

[9] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Comput. Sci.*, 96:217–248, 1992.

[10] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. *Software Practice and Experience*, S4(25):87–130, Dec. 1995.

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

[12] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[13] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February 1989.

[14] P.-C. Cheng, J. A. Garay, A. Herzberg, and H. Krawczyk. Design and implementation of modular key management protocol and IP secure tunnel on AIX. In *Proceedings of the 5th USENIX UNIX Security Symposium*, June 1995.

[15] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, Palaiseau, Nov. 1998.

[16] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96*, pages 372–385. ACM, Jan. 1996.

[17] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, Aug. 1996.

[18] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus. In A. Mazurkiewicz and J. 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.

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

[20] L. Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32(1):32–45, Jan. 1989.

[21] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.

[22] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, pages 112–121, Nov. 1998.

[23] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[24] J. McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–66, Jan. 1996.

[25] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[26] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, Sept. 1992.

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

[28] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2), 1998.

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

[30] F. J. Thayer Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings 1998 IEEE Symposium on Security and Privacy*, pages 160–171, May 1998.

[31] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure network objects. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 211–221, May 1996.

[32] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.