Secure Implementation of Channel Abstractions

Martín Abadi
ma@pa.dec.com
Digital Equipment Corporation
Systems Research Center

Cédric Fournet
Cedric.Fournet@inria.fr
INRIA Rocquencourt

Georges Gonthier
Georges.Gonthier@inria.fr
INRIA Rocquencourt

Abstract

Communication in distributed systems often relies on useful abstractions such as channels, remote procedure calls, and remote method invocations. The implementations of these abstractions sometimes provide security properties, in particular through encryption. In this paper we study those security properties, focusing on channel abstractions. We introduce a simple high-level language that includes constructs for creating and using secure channels. The language is a variant of the join-calculus and belongs to the same family as the pi-calculus. We show how to translate the high-level language into a lower-level language that includes cryptographic primitives. In this translation, we map communication on secure channels to encrypted communication on public channels. We obtain a correctness theorem for our translation; this theorem implies that one can reason about programs in the high-level language without mentioning the subtle cryptographic protocols used in their lower-level implementation.

1 Communication constructs and security

While cryptography is useful for distributed applications, application code should not be concerned with the details of key establishment and management, but should instead rely on abstractions and services that encapsulate cryptographic operations. In recent years, several APIs (application program interfaces) for security have appeared, providing such abstractions and services [22, 23, 26, 24, 9, 21, 34]. Although there are substantial differences among these APIs, they generally offer the promise of making application code more modular, simpler, and ultimately more robust.

In this paper we consider high-level abstractions that largely hide the difficulties of network security from applications. These high-level abstractions support the pleasing illusion that all application address spaces are on the same machine, and that a centralized operating system provides security for them. In reality, these address spaces could be spread across a network, and security could depend on several local operating systems and on cryptographic protocols across machines.

For example, when an application requires secure communication between a client and a server, the application code may include a call to create a secure channel [21]. In the local case, the implementation of the secure channel can rely on a local operating system, whereas in the distributed case it involves a network connection and a key. Similarly, the application code may include a method invocation, which may look the same whether the method invocation is local or remote [34, 35, 33]. Security for remote method invocations may be guaranteed by the transport implementation, fairly transparently to the application.

Up to now, the design and implementation of abstractions for secure communication has been more an art than a science. In particular, little is known on how to analyze programs that use these abstractions, or on what it means to implement these abstractions correctly. We address these issues, focusing on channel abstractions.

- We introduce a simple high-level language that includes constructs for creating and using secure channels. The language is a variant of the join-calculus [15] and belongs to the same family as the pi-calculus [30, 28].
- We show how to translate the high-level language into a lower-level language that includes cryptographic primitives (like the spi-calculus [4]). In this translation, we implement communication on secure channels by encrypted communication on public channels.
- We state and prove a soundness property for our translation. This property guarantees that one can reason about programs in the high-level language independently of their implementation.

- Finally, we briefly consider variants of our translation, bringing it closer to practicality. These variants avoid encryption in cases where it is unnecessary.

Our definitions and proofs are built in a modular fashion on top of small cryptographic protocols. In recent years there have been important advances on methods for analyzing cryptographic protocols (e.g., [12]). We rely on some of those advances, but we are not interested in protocols in isolation. We construct only a few protocols and show that their low-level details match the high-level definition of secure channels. To our knowledge, results of this kind have not previously appeared in the literature.

This paper represents the confluence of several projects, including the development of secure network objects [34] and of the join-calculus [15, 16, 17, 18], and the application of the pi-calculus and the spi-calculus for reasoning about security protocols [4, 5, 6, 1, 13]. Those projects encountered some of the questions treated here; the join-calculus provides a more convenient setting for this study than network-object systems or the pi-calculus [2, 3]. Those projects also produced the work most closely related to that described here, including partial encodings of encryption in the pi-calculus [4, 13]. Our translation goes in the direction opposite to those encodings.

In the next section we describe the source language and the target language of our translation, and give an informal overview of some of the themes of the paper. In sections 3 and 4, we present our translation, with several variants, and in section 5 we study its theory. We conclude in section 6. In this summary we omit many examples, results, and proofs, which are in the full paper [3].

2 From the join-calculus to the sjoin-calculus

In this section, we introduce our high-level language and our low-level language. The appendix contains formal details, including the definition of binary relations $\equiv$ and $\rightarrow$ on processes. Intuitively, $P \equiv Q$ means that P and Q are structurally equivalent (almost syntactically equivalent, modulo some rearrangements); and $P \rightarrow Q$ means that P may perform one step of internal computation and then behave as Q.

2.1 The join-calculus

Our high-level language is a variant of the join-calculus. This is a name-passing calculus, like the pi-calculus, with an emphasis on distributed programming. It constitutes the core of a programming language with a fully distributed implementation [18]. The join-calculus also draws on functional languages with pattern matching and on object-oriented languages.

The syntax of the join-calculus assumes a set of names. In the pure join-calculus, names are used only as names of channels, and a value is simply a name. We represent names with lowercase identifiers x, y, *foo*, ..., and write $\widetilde{v}$ for a tuple of values $v_1, v_2, \ldots, v_n$.

The syntax includes categories of values, processes, definitions, and join-patterns, defined in Figure 1. The operator $|$ has highest precedence, so $\mathsf{def}\ D\ \mathsf{in}\ P \mid P'$ means $\mathsf{def}\ D\ \mathsf{in}\ (P \mid P')$. We use $\mathsf{repl}\ P$ as an abbreviation for $\mathsf{def}\ x\langle\rangle \triangleright (P \mid x\langle\rangle)\ \mathsf{in}\ x\langle\rangle$ for a fresh x; this is the replication of P (also written $!P$ [29]).

The meaning of most of the process constructs should be clear. The process $x\langle\widetilde{v}\rangle$ sends the tuple of values $\widetilde{v}$ on the channel x; this message is asynchronous, in the sense that it does not require any form of handshake. The process $\mathsf{if}\ v = v'\ \mathsf{then}\ P\ \mathsf{else}\ P'$ compares the values v and v' and branches on the result of this comparison; this construct, sometimes called matching, is an addition to the original syntax of the join-calculus. The process $\mathsf{def}\ D\ \mathsf{in}\ P$ is the process P in the scope of the definitions given in D. Definitions obey lexical scoping rules. In particular, a channel name defined in D is recursively bound in the whole of $\mathsf{def}\ D\ \mathsf{in}\ P$.

A join-pattern is a non-empty list of message patterns of the form $x\langle y_1, \ldots, y_n\rangle$. The names $y_1, \ldots, y_n$ are bound formal arguments, 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 $J \triangleright P$ says that P may run whenever there are messages that match J. In the case where J is $x\langle y_1, \ldots, y_m\rangle \mid x'\langle y'_1, \ldots, y'_{m'}\rangle$, we say that J is matched when there are tuples $v_1, \ldots, v_m$ and $v'_1, \ldots, v'_{m'}$ on the channels x and x', respectively. When this happens, the messages are consumed, and P is run, with the actual arguments $v_1, \ldots, v_m$ and $v'_1, \ldots, v'_{m'}$ replaced for the corresponding formal arguments. The reader may infer the full definition of matching from this special case, or consult the appendix.

A definition $D \wedge D'$ is simply the conjunction of D and D'. The same defined name may appear in both conjuncts, as in $x\langle y\rangle \triangleright P \wedge x\langle z\rangle \triangleright Q$; when there is a message $x\langle v\rangle$, either P or Q may run—the choice between them is non-deterministic.

Each channel has an arity (a fixed size for the tuples passed on the channel). We enforce the consistent use of arities with a type system. While there exists a rich, polymorphic type system for the join-calculus [17], a

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.

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 2: Grammar of the sjoin-calculus.

simple monomorphic type system suffices for our purposes. We write $\langle\sigma_1, \dots, \sigma_n\rangle$ for the type of channels that carry tuples with n components of respective types $\sigma_1, \dots, \sigma_n$, and restrict attention to types of this form. Types may be recursively defined, as in $\sigma = \langle\sigma, \sigma\rangle$ (formally, using a fixpoint operator). We assume that each name is associated with a type, and that there are infinitely many names for each type. Throughout, we consider only well-typed processes.

As a first example, we consider the process $P \stackrel{\text{def}}{=} \mathsf{def}\ x\langle y, y'\rangle \triangleright z\langle y\rangle\ \mathsf{in}\ x\langle u, v\rangle$. Here, u and v are arbitrary values, z is the free name of a channel, x is the bound name of a channel, and y and y' are two other bound names that serve as formal parameters. The process P introduces the channel x, and attaches to it the definition $x\langle y, y'\rangle \triangleright z\langle y\rangle$. Thus, when a pair y, y' appears as a message on x, its first component is forwarded on z. The body of P is $x\langle u, v\rangle$, which sends the pair u, v on x.

Informally, we may think of x as a secure channel. Suppose that we put P in an arbitrary (and possibly hostile) context $C[\cdot]$. In $C[P]$, the context $C[\cdot]$ cannot send on the channel x defined in P, so it cannot cause P to forward something else than u on z. Furthermore, $C[\cdot]$ cannot intercept the messages that travel on x in P, so in particular $C[\cdot]$ has no information about the value of v. As in the pi-calculus and in the spi-calculus, such security properties are guaranteed by scoping, and they can be rephrased as equivalences between processes. We say that P_1 and P_2 are equivalent, and write $P_1 \approx P_2$, when no context can distinguish one from the other. (Section 5 contains the definition of $\approx$.) For example, the equivalence $P \approx z\langle u\rangle$ equates P to a process that simply sends u on z, and it implies that the context of P cannot cause P to forward something else than u on z. Similarly, we may express the secrecy of v by the equivalence def $x\langle y, y'\rangle \triangleright z\langle y\rangle$ in $x\langle u, v\rangle \approx$ def $x\langle y, y'\rangle \triangleright z\langle y\rangle$ in $x\langle u, w\rangle$, for all w. The processes being equated are P and the result of replacing v with w in P.

2.2 The sjoin-calculus

Our low-level language is an extension of the join-calculus with constructs for encryption and decryption, and with names that can be used as keys, nonces, or other tags. We call it the sjoin-calculus, by analogy with the spi-calculus. Its grammar is in Figure 2.

The set of values includes not only names but also ciphertexts of the form $\{\widetilde{v}\}_v$. In $\{\widetilde{v}\}_v$, the subscript v is the encryption key, and $\widetilde{v}$ is the cleartext. Informally, we assume that the cleartext or the encryption key cannot be extracted from the ciphertext without prior knowledge of a corresponding decryption key. We also assume that prior knowledge of the decryption key reveals the cleartext, and that the cleartext contains sufficient redundancy so that successful decryption is evident. Ciphertexts can be compared, as for example in the process if $\{v\}_x = \{v'\}_x$ then P else P'; this comparison would enable a recipient of $\{v\}_x$ with knowledge of x and v' to deduce whether v equals v'.

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

The construct fresh x introduces the fresh name x. The construct 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, and have no formal meaning. In general, the names x^+ and x^- will be different; this distinction models the separation of encryption key and decryption key in public-key cryptosystems (e.g., [25]). However, we use keys x, x when x is a shared key (so x is its own inverse), and abbreviate keys x, x to key x. This notation yields a concise model of shared-key cryptography as a special case of public-key cryptography.

The type system for the sjoin-calculus is a straightforward extension of that for the join-calculus: we add a basic type BitString, with the rules that the names introduced by fresh x and keys x^+, x^- are of type BitString, and that encryption and decryption apply only when their arguments are of type BitString and yield results of type BitString.

Our first example for the sjoin-calculus continues our first example for the join-calculus, which concerns the process $P \stackrel{\text{def}}{=}$ def $x\langle y, y'\rangle \triangleright z\langle y\rangle$ in $x\langle u, v\rangle$. When this process is implemented on a single machine, the local operating system can guarantee the desired security properties for the channel x. On the other hand, suppose that the definition $x\langle y, y'\rangle \triangleright z\langle y\rangle$ and the body $x\langle u, v\rangle$ are located in different machines. In that case, the channel x might be implemented by a TCP connection between the machines. This implementation would be subject to a variety of attacks, which proper use of encryption can thwart. In particular, we may associate a key pair x^+, x^- with x in order to encrypt messages on x, as in the following tentative implementation of P in the sjoin-calculus:

$$\begin{array}{ll} P' & \stackrel{\text{def}}{=} \\ \mathsf{def} & \mathsf{keys}\ x^+, x^- \\ \wedge & a\langle c\rangle \triangleright (\mathsf{decrypt}\ c\ \mathsf{using}\ x^-\ \mathsf{to}\ y, y'\ \mathsf{in}\ z\langle y\rangle\ \mathsf{else}\ 0) \\ \mathsf{in} & e\langle a, \{u, v\}_{x^+}\rangle \end{array}$$

Our intent is that the expressions $a\langle c\rangle \triangleright$(decrypt c using x^- to y, y' in $z\langle y\rangle$ else 0) and $e\langle a, \{u, v\}_{x^+}\rangle$ may be located in different machines. The name e represents a public, unprotected channel that P' uses (like an Ethernet). Instead of transmitting u and v on a private channel x, P' transmits them on e after encryption under x^+ and tagging with a. It is the responsibility of the environment to relay $\{u, v\}_{x^+}$ on a. On receipt of a message c on a, the expression $a\langle c\rangle \triangleright$(decrypt c using x^- to y, y' in $z\langle y\rangle$ else 0) tries to decrypt a pair y, y' using x^-. In case of success, it forwards y on z; otherwise it stops.

While P' may resemble a secure distributed implementation of P, it still has several gaps. For example, nothing prevents an attacker from replaying $\{u, v\}_{x^+}$ on a. Whereas P' would then output u on z twice, P outputs u on z only once.

The techniques that we describe in this paper address these difficulties. We show a systematic, mechanical way of implementing a process such as P by translating it to a lower-level process similar to P'. Like P', the lower-level processes that we construct rely on cryptographic protection for communication across machines rather than on abstract secure channels. (We do not rely on the trivial translation that maps every process to itself.) On the other hand, our translation is more sophisticated than P' might suggest. It protects against attacks and—at least within our formal framework—we can prove that it is correct. In particular, we can prove that if two processes are equivalent in the join-calculus, then their implementations are equivalent in the sjoin-calculus, and vice versa.

3 Low-level communication

In this section, we discuss our model of low-level communication, and produce two protocols that transmit a single message. Either of these protocols, or any other protocol with the same properties, can be used in the translation of section 4.

3.1 The low-level network model

Our informal assumptions about the network used by sjoin-calculus processes are similar to those of Needham and Schroeder [31]: "We assume that an intruder can interpose a computer in all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. [...] We also assume that each principal has a secure environment in which to compute, such as is provided by a personal computer or would be by a secure shared operating system."

In our model, anyone can send and receive messages of type BitString through the network interface consisting of the channels *emit* and *recv*. For output, a process P sends its message on *emit*. For input, P sends a continuation channel κ of type $\langle\mathsf{BitString}\rangle$ on *recv*; the network returns a message on κ. The expression $\mathsf{def}\ recv\langle\kappa\rangle \mid emit\langle m\rangle \triangleright \kappa\langle m\rangle\ \mathsf{in}\ P$ describes the network's behavior.

Processes have to filter for messages that are destined to them. We rely on keys as the basis for this filtering, with the following definition:

$$\begin{array}{l} \mathsf{let}\ \widetilde{y} = \mathsf{filter}\ k\ \mathsf{in}\ P \quad \stackrel{\text{def}}{=} \\ \quad \mathsf{def}\ \kappa\langle m\rangle \triangleright \ \ \mathsf{decrypt}\ m\ \mathsf{using}\ k\ \mathsf{to}\ \widetilde{y}\ \mathsf{in}\ P \\ \qquad\qquad\qquad\ \ \mathsf{else}\ emit\langle m\rangle \mid recv\langle\kappa\rangle \\ \quad \mathsf{in}\ recv\langle\kappa\rangle \end{array}$$

This process picks a message m from the network and attempts to decrypt it using a key k. If the decryption succeeds, then the process executes P with m's contents replaced for $\widetilde{y}$; otherwise, the process returns m to the network and retries. This method of addressing is fairly rudimentary and costly on an Ethernet and even more on a wide-area network. Unfortunately, using cleartext addresses would expose us to traffic-analysis attacks.

Because of the simplicity of our network interface, injecting noise into the network in order to prevent traffic analysis is straightforward. The process W creates the desired noise:

$$W \quad \stackrel{\text{def}}{=} \quad \mathsf{repl}\ (\mathsf{def\ fresh}\ m\ \mathsf{in\ repl}\ emit\langle m\rangle)$$

Whenever we study the implementations of join-calculus processes, we do it in a context that defines *emit* and *recv*, in the presence of W, and in addition in the presence of the process $plug\langle emit, recv\rangle$. This process transmits the network-interface channels *emit* and *recv* on a public channel *plug*, so that any outside process has access to them. Combining these expressions, we obtain the following context:

$$\begin{array}{rcl} \mathcal{E}nv\,[\,\cdot\,] & \stackrel{\text{def}}{=} & \mathsf{def}\ recv\langle\kappa\rangle \mid emit\langle m\rangle \triangleright \kappa\langle m\rangle\ \mathsf{in} \\ & & plug\langle emit, recv\rangle \mid W \mid \,\cdot \end{array}$$

Note that, while an intruder may intercept any message, we do not guarantee that it actually will intercept every message. (In this we seem to differ from Needham and Schroeder.) The intruder has access to the channels *emit* and *recv*, but cannot completely control their behavior. This assumption is fairly realistic, and simplifies our task. For example, the translations of the processes $x\langle\rangle$ and $\mathsf{def}\ y\langle\rangle \triangleright x\langle\rangle\ \mathsf{in}\ y\langle\rangle$ should be indistinguishable, since these processes are indistinguishable according to the semantics of the join-calculus. On the other hand, an intruder that was in charge of transporting every message could observe a difference between $x\langle\rangle$ and $\mathsf{def}\ y\langle\rangle \triangleright x\langle\rangle\ \mathsf{in}\ y\langle\rangle$ by interrupting all messages on y, then observing whether a message on x appears anyway. There would however be no difficulty in guaranteeing that an intruder can obtain a copy of every message, by changing the definition of $\mathcal{E}nv\,[\,\cdot\,]$; our results would still hold.

3.2 Translation framework

Below we present two alternative methods for implementing a message on a channel x. In both cases, we map a message on x to an execution of a protocol that uses the keys x^+, x^- and the channels *emit* and *recv*.

Both protocols are based on definitions of two sjoin-calculus processes: $E_x[\widetilde{v}]$, which sends $\widetilde{v}$ using the encryption key x^+, and R_x, which uses the key x^- to decrypt a message and forwards its contents in clear on an internal channel x°. Naively, one may define:

$$\begin{array}{rcl} E_x[\widetilde{v}] & \stackrel{\text{def}}{=} & emit\langle\{\widetilde{v}\}_{x^+}\rangle \\ R_x & \stackrel{\text{def}}{=} & \mathsf{let}\ \widetilde{y} = \mathsf{filter}\ x^-\ \mathsf{in}\ x^\circ\langle\widetilde{y}\rangle \end{array}$$

where the length of $\widetilde{y}$ is deduced from the type of x. The naive protocol that relies on these definitions is subject to obvious attacks.

Throughout, we associate three values x^+, x^-, and x° with each channel name x; both x^+ and x^- are of type BitString while x° is of type $\langle\mathsf{BitString}, \ldots, \mathsf{BitString}\rangle$ with the same arity as x. We assume that the values x^+, x^-, and x° are not names that appear in the join-calculus processes under consideration. We ensure that the mappings $x \mapsto x^-$ and $x \mapsto x^\circ$ are injective, and that the mapping $x \mapsto x^+$ is injective given the type of x.

3.3 Basic protocol

The basic protocol is an enhancement of the naive protocol in several respects. The emitter of a message repeats the message indefinitely, in case the message is intercepted. The emitter also adds a fresh component c to each message. This component serves both as a "confounder" (for preventing the generation and detection of two identical messages) and as a unique identifier (for thwarting replay attacks). The recipient of a message records its unique identifier, and drops any message whose unique identifier has been recorded previously. We omit the details of how to program a data structure with unique identifiers in the sjoin-calculus. We simply write the declaration $\mathsf{uids}\ t$ to denote the definition of an initially empty set of unique identifiers t; and write $\mathsf{if\ not\ tset}\ t(c)\ \mathsf{then}\ P$ for a process that atomically tests whether the unique identifier c is in t, and if not adds c to t and then triggers the execution of P. We arrive at the following definitions, where the length of $\widetilde{y}$ is deduced from the type of x, and where F_x is an auxiliary process:

$$\begin{array}{rcl} E_x[\widetilde{v}] & \stackrel{\text{def}}{=} & \mathsf{def\ fresh}\ c\ \mathsf{in\ repl}\ emit\langle\{c,\widetilde{v}\}_{x^+}\rangle \\ F_x & \stackrel{\text{def}}{=} & \mathsf{let}\ c,\widetilde{y} = \mathsf{filter}\ x^-\ \mathsf{in} \\ & & \mathsf{if\ not\ tset}\ t_x(c)\ \mathsf{then}\ x^\circ\langle\widetilde{y}\rangle \\ R_x & \stackrel{\text{def}}{=} & \mathsf{def\ uids}\ t_x\ \mathsf{in\ repl}\ F_x \end{array}$$

3.4 Session-based protocol

The basic protocol has a major drawback: the recipients of messages must remember the unique identifiers of those messages forever. The session-based protocol avoids this drawback by relying on a three-message, challenge-response dialogue. The emitter first sends a new shared key k encrypted with x^+. The recipient returns a challenge, and waits for a single response; the challenge is a new shared key k' encrypted with k. The actual payload data is transmitted in the final message, encrypted with k'. The corresponding definitions are:

$$\begin{array}{rcl} E_x[\widetilde{v}] & \stackrel{\text{def}}{=} & \mathsf{def\ key}\ k\ \mathsf{in} \\ & & (\mathsf{repl}\ emit\langle\{k\}_{x^+}\rangle) \mid \\ & & \mathsf{let}\ k' = \mathsf{filter}\ k\ \mathsf{in}\ (\mathsf{repl}\ emit\langle\{\widetilde{v}\}_{k'}\rangle) \\ F_x & \stackrel{\text{def}}{=} & \mathsf{let}\ k = \mathsf{filter}\ x^-\ \mathsf{in} \\ & & \mathsf{def\ key}\ k'\ \mathsf{in} \\ & & (\mathsf{repl}\ emit\langle\{k'\}_k\rangle) \mid \\ & & \mathsf{let}\ \widetilde{y} = \mathsf{filter}\ k'\ \mathsf{in}\ x^\circ\langle\widetilde{y}\rangle \\ R_x & \stackrel{\text{def}}{=} & \mathsf{repl}\ F_x \end{array}$$

The key k' can be regarded as a session key; the steps where the emitter sends k under x^+ and the recipient replies with k' under k constitute a key-establishment protocol for a session. Here the session conveys a single useful payload, $\widetilde{v}$. However, it would be straightforward to extend the protocol to longer dialogues. It would also be possible to add acknowledgments and time-outs, causing the protocol to terminate in a realistic manner, but reasoning about the resulting processes seems rather complex.

4 Translations

In section 3 we describe sjoin-calculus protocols that can be used as implementations for single join-calculus messages. Since message-passing is the fundamental computation step in the join-calculus, these protocols are the basis for complete implementations of the join-calculus in the sjoin-calculus.

We actually have two techniques for these implementations. The first technique is a compositional translation from the join-calculus to the sjoin-calculus. It does not make any assumptions about the distribution of the process being translated. In principle, every message of the source process could travel on a physically insecure network, and therefore needs cryptographic protection. The second technique relies on a context $\mathcal{F}_S[\,\cdot\,]$ that operates as a filter, applying encryption and converting message formats (much like a firewall with an encrypting tunnel [11, 7]). For example, in $\mathcal{F}_S[P]$, when a message leaves P on a channel x, the context $\mathcal{F}_S[\,\cdot\,]$ turns it into a message encrypted with the key x^+; if the message contains the channel name y, then y gets replaced with y^+. Additionally, $\mathcal{F}_S[\,\cdot\,]$ decrypts any incoming message encrypted with y^+, translating its contents and forwarding the result on y. This filtering translation allows us to avoid encryption on messages that we consider local. We omit the definition of the filtering translation in this summary, because it is fairly long and challenging.

4.1 A compositional translation

If P is a join-calculus process, then we write $[\![P]\!]$ for its translation to the sjoin-calculus; we write $[\![\,\cdot\,]\!]$ for the translation function. The definition of $[\![\,\cdot\,]\!]$ is in Figure 3. There, we write $\mathsf{dv}[D]$ for the set of names defined in D, and $\widetilde{v}^+$ for $v_1^+,\ldots,v_n^+$ if $\widetilde{v}$ is $v_1,\ldots,v_n$.

The translation yields processes that can be executed in a distributed manner without any "built-in" security properties of channels. In $[\![P \mid P']\!]$, the components $[\![P]\!]$ and $[\![P']\!]$ may be located in different machines. Similarly, in $[\![\mathsf{def}\ D\ \mathsf{in}\ P]\!]$, the components $[\![P]\!]$ and $\mathsf{def}\ [\![D]\!]\ \mathsf{in}\ \prod_{x\in\mathsf{dv}[D]} R_x$ may be located in different machines. Except for a few continuations for internal

$$
\begin{aligned}
[\![x\langle\widetilde{v}\rangle]\!] &\stackrel{\text{def}}{=} E_x[\widetilde{v}^+] \\
[\![\mathsf{def}\ D\ \mathsf{in}\ P]\!] &\stackrel{\text{def}}{=} \mathsf{def}\ \textstyle\bigwedge_{x\in\mathsf{dv}[D]}\mathsf{keys}\ x^+, x^-\ \mathsf{in} \\
&\qquad \Big(\mathsf{def}\ [\![D]\!]\ \mathsf{in}\ \textstyle\prod_{x\in\mathsf{dv}[D]} R_x\Big) \mid [\![P]\!] \\
[\![\mathsf{if}\ v = v'\ \mathsf{then}\ P\ \mathsf{else}\ P']\!] &\stackrel{\text{def}}{=} \mathsf{if}\ v^+ = v'^+\ \mathsf{then}\ [\![P]\!]\ \mathsf{else}\ [\![P']\!] \\
[\![P \mid P']\!] &\stackrel{\text{def}}{=} [\![P]\!] \mid [\![P']\!] \\
[\![0]\!] &\stackrel{\text{def}}{=} 0
\end{aligned}
$$

(a) On processes.

$$
\begin{aligned}
[\![x\langle\widetilde{y}\rangle]\!] &\stackrel{\text{def}}{=} x^\circ\langle\widetilde{y}^+\rangle \\
[\![J \mid J']\!] &\stackrel{\text{def}}{=} [\![J]\!] \mid [\![J']\!]
\end{aligned}
$$

(b) On patterns.

$$
\begin{aligned}
[\![J \triangleright P]\!] &\stackrel{\text{def}}{=} [\![J]\!] \triangleright [\![P]\!] \\
[\![D \wedge D']\!] &\stackrel{\text{def}}{=} [\![D]\!] \wedge [\![D']\!]
\end{aligned}
$$

(c) On definitions.

Figure 3: Definition of $[\![\,\cdot\,]\!]$.

use, channels convey only encrypted messages and encryption keys.

As an example, we translate the simple process considered in section 2. According to the definitions, $[\![\mathsf{def}\ x\langle y, y'\rangle \triangleright z\langle y\rangle\ \mathsf{in}\ x\langle u, v\rangle]\!]$ is:

$$
\begin{aligned}
&\mathsf{def\ keys}\ x^+, x^-\ \mathsf{in} \\
&(\mathsf{def}\ x^\circ\langle y^+, y'^+\rangle \triangleright E_z[y^+]\ \mathsf{in}\ R_x) \mid E_x[u^+, v^+]
\end{aligned}
$$

This process relies on the keys x^+ and x^- and on the channel x°, which correspond to the channel x. The subprocess $E_x[u^+, v^+]$ sends the keys u^+, v^+, encrypted with x^+, on *emit*. The subprocess R_x may pick up this message through *recv*, decrypt it using x^-, and forward its contents on x°. The definition $x^\circ\langle y^+, y'^+\rangle \triangleright E_z[y^+]$ implies that, if u^+, v^+ are received on x°, then u^+ is resent on *emit*, now encrypted with z^+.

As stated in section 2, $\mathsf{def}\ x\langle y, y'\rangle \triangleright z\langle y\rangle\ \mathsf{in}\ x\langle u, v\rangle$ is equivalent to $z\langle u\rangle$. The translation of $z\langle u\rangle$ is simply $E_z[u^+]$. In section 5 we prove that if $P \approx Q$ then $\mathcal{E}nv\,[\,[\![P]\!]\,] \approx \mathcal{E}nv\,[\,[\![Q]\!]\,]$. For this example, our theorem implies that no context can distinguish the translation of $\mathsf{def}\ x\langle y, y'\rangle \triangleright z\langle y\rangle\ \mathsf{in}\ x\langle u, v\rangle$ from $E_z[u^+]$. This equivalence holds because the context cannot tamper with our cryptographic implementation of the channel x.

5 Soundness results

In this section we define the relation of equivalence on processes, mentioned informally in other sections, and then we state our main theorems and discuss their significance.

5.1 Observational equivalence and other relations

We use a form of observational equivalence as a basis for reasoning about processes. The concept of observational equivalence goes back to Milner's work on CCS [27]; more specifically, we adopt a reduction-based semantics, as for example in the work of Honda and Yoshida [20]. All the definitions are fairly standard, and apply to the join-calculus and to the sjoin-calculus.

Evaluation contexts are the contexts generated by the following grammar:

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

where P ranges over processes and D over definitions of the calculus under consideration.

For every process P and name x, we define the assertions $P \downarrow_x$ and $P \Downarrow_x$ as follows:

$$
\begin{aligned}
P \downarrow_x &\stackrel{\text{def}}{=} \exists C[\,\cdot\,],\, \widetilde{v}.\, P \equiv C[x\langle\widetilde{v}\rangle] \\
&\qquad \textit{with } C[\,\cdot\,] \textit{ an evaluation context that} \\
&\qquad \textit{does not bind } x \\
P \Downarrow_x &\stackrel{\text{def}}{=} \exists P'.\, P \rightarrow^* P' \wedge P' \downarrow_x
\end{aligned}
$$

where $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$. That is, $P \downarrow_x$ holds if P may output on x immediately, and $P \Downarrow_x$ holds if P may output on x either immediately or after some internal reductions.

Observational equivalence ($\approx$) is the largest symmetric relation $\mathcal{R}$ on processes such that:

1. if $P \mathrel{\mathcal{R}} Q$ and $P \Downarrow_x$ then $Q \Downarrow_x$;
2. $\mathcal{R}$ is a congruence for all evaluation contexts, that is, for all evaluation contexts $C[\,\cdot\,]$, if $P \mathrel{\mathcal{R}} Q$ then $C[P] \mathrel{\mathcal{R}} C[Q]$;
3. $\mathcal{R}$ is a weak bisimulation, that is, if $P \mathrel{\mathcal{R}} Q$ and $P \rightarrow P'$ then, for some Q', $P' \mathrel{\mathcal{R}} Q'$ and $Q \rightarrow^* Q'$.

May-testing equivalence ($\simeq_{may}$) is the largest symmetric relation $\mathcal{R}$ that meets only requirements (1) and (2). *Expansion* ($\succeq$) [32] is the largest relation $\mathcal{R}$ that meets requirements (1), (2), and (3), such that its converse meets those requirements too, and such that if $P \mathrel{\mathcal{R}} Q$ and $P \rightarrow P'$ then, for some Q', $P' \mathrel{\mathcal{R}} Q'$ and $Q \rightarrow^= Q'$, where $\rightarrow^=$ is the reflexive closure of $\rightarrow$.

5.2 Main results

Our main results about the translations hold whether the translations are built on the basic protocol or on the session-based protocol. In fact, we need only the following common properties of the two protocols. Let us write $\mathcal{E}nv_x[\cdot]$ as abbreviation for $\mathcal{E}nv\,[\,\mathsf{def\ keys}\ x^+, x^-\ \mathsf{in}\ \mathit{plug}'\langle x^+\rangle \mid [\cdot]\,]$. There exists a set of processes $\mathbf{R}$ (intuitively, representing the states of a receiver after interaction with its context) such that:

1. $R_x \in \mathbf{R}$.

2. The free names of $E_x[\cdot]$ are at most *emit*, *recv*, and names of type $\mathsf{BitString}$. For every $R \in \mathbf{R}$, the free names of R are at most *emit*, *recv*, x°, and names of type $\mathsf{BitString}$. Intuitively, this condition implies that communication from $E_x[\cdot]$ to R_x is limited to messages of types $\mathsf{BitString}$ and $\langle\mathsf{BitString}\rangle$ on the channels *emit* and *recv*.

3. For every $R \in \mathbf{R}$, it is not the case that $R \downarrow_{emit}$ or that $R \downarrow_{x^\circ}$. Intuitively, this condition says that every process in $\mathbf{R}$ is passive, in the sense that it never sends messages on *emit* or x° on its own.

4. For every $R \in \mathbf{R}$, if $\widetilde{v}$ is a tuple of values of type $\mathsf{BitString}$ whose length matches the arity of x then $\mathcal{E}nv_x[\,R \mid E_x[\widetilde{v}]\,] \succeq \mathcal{E}nv_x[\,R \mid x^\circ\langle\widetilde{v}\rangle\,]$. Intuitively, this condition says that messages are transmitted reliably and secretly when an instance of the sending process $E_x[\cdot]$ is put in parallel with the receiving process R_x or any other process in $\mathbf{R}$.

5. For every value v, if x^+ and x^- do not occur in v and $\mathcal{E}nv_x[\,R_x \mid \mathit{emit}\langle v\rangle\,] \rightarrow P$ then $P \succeq \mathcal{E}nv_x[\,R_x \mid \mathit{emit}\langle v\rangle\,]$. Intuitively, this condition describes the behavior of R_x in a context that does not have access to the keys x^+ or x^-; essentially, R_x remains invisible.

6. For every $R \in \mathbf{R}$, if x^- does not occur in v and $\mathcal{E}nv_x[\,R \mid \mathit{emit}\langle v\rangle\,] \rightarrow P$ then $P \succeq \mathcal{E}nv_x[\,R' \mid Q\,]$ for some $R' \in \mathbf{R}$ and some process Q such that x^- does not occur in Q. Intuitively, this condition describes the behavior of a process in $\mathbf{R}$ in a context that has access to the encryption key x^+ (but not to the decryption key x^-); in this situation, the context should still not interfere with messages from other senders that use x^+.

Our main results are full-abstraction theorems for our translations. First we consider the compositional translation.

Theorem 1 *The compositional translation is fully-abstract, up to observational equivalence: for all join-calculus processes P and Q,*

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

The statement of this theorem may seem ambiguous, because $P \approx Q$ can be read as the equivalence of P and Q as join-calculus processes or as sjoin-calculus processes. However, it can be shown that these two readings coincide.

The next theorem implies that the filtering translation is fully abstract too.

Theorem 2 *The filtering translation is equivalent to the compositional translation, up to observational equivalence: $\mathcal{E}nv\,[\,\mathcal{F}_S[P]\,] \approx \mathcal{E}nv\,[\,[\![P]\!]\,]$ for every join-calculus process P and every set of names S that includes all the free names of P.*

In general, full-abstraction results for one equivalence do not imply full-abstraction results for coarser equivalences. However, we have full-abstraction results for a large class of equivalences.

Theorem 3 *Let $\mathcal{R}$ be a transitive relation on sjoin-calculus processes that is a congruence for evaluation contexts and such that $\approx\, \subseteq \mathcal{R}$. For all join-calculus processes P and Q,*

$$P \mathrel{\mathcal{R}} Q \quad \textit{if and only if} \quad \mathcal{E}nv\,[\,[\![P]\!]\,] \mathrel{\mathcal{R}} \mathcal{E}nv\,[\,[\![Q]\!]\,]$$

In particular, may-testing equivalence ($\simeq_{may}$) is coarser than observational equivalence and is a congruence. (It is perhaps the most important relation that satisfies these conditions, because of its correspondence with safety properties [14].) Therefore, we obtain, for example:

$$P \simeq_{may} Q \quad \text{if and only if} \quad \mathcal{E}nv\,[\,[\![P]\!]\,] \simeq_{may} \mathcal{E}nv\,[\,[\![Q]\!]\,]$$

Theorem 2 immediately implies that analogous results hold for the filtering translation.

5.3 Discussion

The "upwards" direction of our full-abstraction results says that two join-calculus processes are equivalent only if their implementations in the sjoin-calculus are equivalent. This property is important because it means that our translations do not identify "too many" join-calculus processes or even—trivially—all join-calculus processes.

For security, however, full abstraction is interesting mainly in the "downwards" direction [2]: if two join-calculus processes are equivalent, then their implementations in the sjoin-calculus are equivalent too. Hence, it is sound to reason about a join-calculus process and to infer properties of its sjoin-calculus implementation, without explicitly considering the low-level communication mechanisms of the implementation and without concern for low-level attacks on those communication mechanisms.

For example, suppose that we have written a join-calculus process that publishes a communication channel. We can assess the consequences of this publication within the high-level calculus, considering only the high-level contexts in which the process may be used. Those consequences may be undesirable—our results do not exclude that possibility. Our results do imply that we need not think about lower-level details of how the channel is represented by keys, or about low-level attacks that this representation might permit.

We should mention, as a caveat, that the low-level attacks that we can rule out are only those that can be expressed in the sjoin-calculus. Although the sjoin-calculus is rich, it does have limitations. In particular, it does not include any notion of real time or of network bandwidth, so it excludes from consideration any attack that depends on observing real-time behavior or bandwidth usage.

Our full-abstraction results are rather robust, as they hold for a range of equivalence relations on processes, including observational equivalence and may-testing equivalence, and also intermediate equivalences that incorporate notions of fairness, such as fair-testing equivalence [10]. Much as in the spi-calculus, observational equivalence can often be proved more directly than may-testing equivalence. While may-testing equivalence probably has a more intuitive meaning, our results for observational equivalence demonstrate that our translations preserve even the branching structure of processes.

The strength of our results has a price: in order to obtain these results, we have adopted a sophisticated and somewhat expensive implementation strategy. For example, our results would not hold if we did not protect against traffic-analysis attacks. For this protection, we rely on the use of noise. Since in practice noise consumes network bandwidth, it would be attractive to omit the noise; one would still hope to prove a correctness result—at least a weak one. Formulating and proving such results is an intriguing subject for future research.

6 Conclusions

This paper describes an implementation of a high-level language with secure channels in terms of a lower-level language with primitives for encryption and decryption. The implementation relies on cryptographic protocols for protecting communication over a public network. The design and analysis of good cryptographic protocols always requires both ingenuity and care. However, this paper does not focus on cryptographic protocols, but instead takes a broader perspective. It considers how the protocols fit into larger systems, and proves that we can reason about programs in the high-level language without explicit concern for lower-level attacks on the protocols.

Several variants and extensions of our results would be attractive, for example the treatment of remote method invocation, and the mapping down to a low-level language where keys are not formal symbols but finite sequences of bits. More generally, many interesting questions in security can be seen as problems of refinement—relating views at different levels of abstraction (e.g., [19]). Different views of security are appropriate for different communities of users, administrators, designers, and programmers. The gaps between these views are sources of confusion and vulnerability. Unfortunately, some of these gaps are still a fuzzy subject for research, because they stem from fuzzy human expectations.

Appendix: Some definitions

In this appendix we detail the operational semantics of the sjoin-calculus; this appendix serves also as a review of the operational semantics of the join-calculus, since it is a special case of that of the sjoin-calculus.

Scopes

In Figure 4, 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 $\mathsf{keys}\ x, y$, $\mathsf{fresh}\ z$, or $\mathsf{uids}\ t$ (but we allow $\mathsf{keys}\ x, x$).

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

$$
\begin{array}{rclrcl}
\mathsf{rv}[x\langle y_1, \cdots, y_n \rangle] & \stackrel{\text{def}}{=} & \{y_1, \cdots, y_n\} & \mathsf{dv}[x\langle y_1, \cdots, y_n \rangle] & \stackrel{\text{def}}{=} & \{x\} \\
\mathsf{rv}[J \mid J'] & \stackrel{\text{def}}{=} & \mathsf{rv}[J] \uplus \mathsf{rv}[J'] & \mathsf{dv}[J \mid J'] & \stackrel{\text{def}}{=} & \mathsf{dv}[J] \uplus \mathsf{dv}[J'] \\
\\
\mathsf{fv}[J \triangleright P] & \stackrel{\text{def}}{=} & \mathsf{dv}[J] \cup (\mathsf{fv}[P] \setminus \mathsf{rv}[J]) & \mathsf{dv}[J \triangleright P] & \stackrel{\text{def}}{=} & \mathsf{dv}[J] \\
\mathsf{fv}[\mathsf{fresh}\ x] & \stackrel{\text{def}}{=} & \{x\} & \mathsf{dv}[\mathsf{fresh}\ x] & \stackrel{\text{def}}{=} & \{x\} \\
\mathsf{fv}[\mathsf{keys}\ x^+, x^-] & \stackrel{\text{def}}{=} & \{x^+, x^-\} & \mathsf{dv}[\mathsf{keys}\ x^+, x^-] & \stackrel{\text{def}}{=} & \{x^+, x^-\} \\
\mathsf{fv}[D \wedge D'] & \stackrel{\text{def}}{=} & \mathsf{fv}[D] \cup \mathsf{fv}[D'] & \mathsf{dv}[D \wedge D'] & \stackrel{\text{def}}{=} & \mathsf{dv}[D] \cup \mathsf{dv}[D']
\end{array}
$$

Figure 4: Scopes.

Chemical semantics

We present our operational semantics in the chemical style of Berry and Boudol [8], as a variant of the reflexive chemical abstract machine [15].

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) and $\mathcal{D}$ is a multiset of definitions. The rules for computation operate on chemical solutions. They form two families, *structural rules* and *chemical 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:

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

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 0 for $\mid$. The rule Str-Def describes the introduction of new names and reaction rules in a solution, according to a static scoping discipline.

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 given in Figure 5 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.

Although tset and repl are defined constructs, it is convenient to treat them as primitive in the sjoin-calculus, with the rules of Figure 6. This addition does not affect the resulting definition of observational equivalence, but it impacts the expansion relation. The structural rule Str-Repl states that processes in repl statements can be replicated. In the reduction rule Red-Uids, uids $t\{\widetilde{u}\}$ denotes a table that already

RED	$J \triangleright P \vdash J\sigma_{\mathrm{rv}}$	$\longrightarrow$	$J \triangleright P \vdash P\sigma_{\mathrm{rv}}$
COMPARE	$\vdash \mathsf{if}\ v = v\ \mathsf{then}\ P\ \mathsf{else}\ P'$	$\longrightarrow$	$\vdash P$
	$\vdash \mathsf{if}\ v = w\ \mathsf{then}\ P\ \mathsf{else}\ P'$	$\longrightarrow$	$\vdash P'$
DECRYPT			
	$\mathsf{keys}\ x^+, x^- \vdash \mathsf{decrypt}\ \{\widetilde{v}\}_{x^+}\ \mathsf{using}\ x^-\ \mathsf{to}\ \widetilde{y}\ \mathsf{in}\ P\ \mathsf{else}\ P'$	$\longrightarrow$	$\mathsf{keys}\ x^+, x^- \vdash P\left\{{}^{\widetilde{v}}/_{\widetilde{y}}\right\}$
	$\mathsf{keys}\ x^+, x^- \vdash \mathsf{decrypt}\ w\ \mathsf{using}\ x^-\ \mathsf{to}\ \widetilde{y}\ \mathsf{in}\ P\ \mathsf{else}\ P'$	$\longrightarrow$	$\mathsf{keys}\ x^+, x^- \vdash P'$

Figure 5: Reduction rules for the sjoin-calculus.

STR-REPL	$\vdash \mathsf{repl}\ P$	$\rightleftharpoons$	$\vdash P \mid \mathsf{repl}\ P$
RED-UIDS	$\mathsf{uids}\ t\{\widetilde{u}\} \vdash \mathsf{if\ not\ tset}\ t(v)\ \mathsf{then}\ P$	$\longrightarrow$	$\mathsf{uids}\ t\{\widetilde{u}, v\} \vdash P$

Figure 6: Additional rules.

contains a given set $\{\widetilde{u}\}$ of values, and $\mathsf{uids}\ t\{\widetilde{u}, v\}$ denotes the same table with an additional value v, assumed distinct from all values in $\{\widetilde{u}\}$. This rule states that $\mathsf{if\ not\ tset}\ t(v)\ \mathsf{then}\ P$ atomically tests whether v is in t, and if not then it adds v to t and starts process P. When v is among the values in $\{\widetilde{u}\}$, $\mathsf{if\ not\ tset}\ t(v)\ \mathsf{then}\ P$ is inert. (We identify $\mathsf{uids}\ t\{\}$ with $\mathsf{uids}\ t$.)

For processes, we define the relations of *structural equivalence* ($\equiv$) and of *reduction modulo structural equivalence* ($\rightarrow$) by a combination of heating, reduction, and cooling of chemical solutions. Relying on these definitions, everywhere else in this paper we work with processes rather than with chemical solutions.

$$P \equiv P' \quad \stackrel{\text{def}}{=} \quad \emptyset \vdash \{P\} \rightleftharpoons^* \emptyset \vdash \{P'\}$$
$$P \rightarrow P' \quad \stackrel{\text{def}}{=} \quad \emptyset \vdash \{P\} \rightleftharpoons^* \longrightarrow \rightleftharpoons^* \emptyset \vdash \{P'\}$$

References

[1] M. Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 611–638. Springer-Verlag, 1997.

[2] M. Abadi. Protection in programming-language translations. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, July 1998. To appear.

[3] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. Manuscript, full version of this paper, on the Web at `http://pauillac.inria.fr/join`, 1998.

[4] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. Technical Report 414, University of Cambridge Computer Laboratory, Jan. 1997. Extended version of both [5] and [6]. A revised version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998, and an abridged version will appear in *Information and Computation*.

[5] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 36–47, April 1997.

[6] M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the spi 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 59–73. Springer-Verlag, July 1997.

[7] 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`.

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

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

[10] E. Brinksma, A. Rensink, and W. Vogler. Fair testing. In I. Lee and S. A. Smolka, editors, *Proceedings of the 6th International Conference on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 313–327. Springer-Verlag, 1995.

[11] 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.

[12] *10th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1997. Proceedings.

[13] M. Dam. Proving trust in systems of second-order processes: Preliminary results. Draft, 1997.

[14] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Comput. Sci.*, 34:83–133, 1984.

[15] 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.

[16] 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.

[17] 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.

[18] C. Fournet and L. Maranget. The join-calculus language. Web pages at `http://pauillac.inria.fr/join`, June 1997.

[19] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[20] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Comput. Sci.*, 151:437–486, 1995.

[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] J. Linn. RFC 1508: Generic security service application program interface. Web page at `ftp://ds.internic.net/rfc/rfc1508.txt`, Sept. 1993.

[23] J. Linn. Generic interface to security services. *Computer Communications*, 17(7):476–482, July 1994.

[24] D. L. McDonald, B. G. Phan, and R. J. Atkinson. A socket-based key management API (and surrounding infrastructure). In *Proceedings of INET96*, 1996. On the Web at `http://www.isoc.org/isoc/whatis/conferences/inet/96/proceedings/`.

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

[26] Microsoft. CryptoAPI. Web pages at `http://www.microsoft.com/security/tech/`.

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

[28] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[29] R. Milner. The polyadic π-calculus: a tutorial. In Bauer, Brauer, and Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.

[30] 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.

[31] 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.

[32] D. Sangiorgi and R. Milner. The problem of "weak bisimulation up to". In W. R. Cleaveland, editor, *Proceedings of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1992.

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

[34] 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.

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