A Survey of Name-Passing Calculi and Crypto-Primitives*

Michele Bugliesi[1], Giuseppe Castagna[2], Silvia Crafa[1],
Riccardo Focardi[1], and Vladimiro Sassone[3]

[1] Università "Ca' Foscari", Venezia
[2] Ecole Normale Superieure, Paris
[3] University of Sussex

Abstract. The paper surveys the literature on high-level name-passing process calculi, and their extensions with cryptographic primitives. The survey is by no means exhaustive, for essentially two reasons. First, in trying to provide a coherent presentation of different ideas and techniques, one inevitably ends up leaving out the approaches that do not fit the intended roadmap. Secondly, the literature on the subject has been growing at very high rate over the years. As a consequence, we decided to concentrate on few papers that introduce the main ideas, in the hope that discussing them in some detail will provide sufficient insight for further reading.

Outline of the Paper

We start in Section 1 with a brief review of a polyadic version of Milner's π-calculus. Then we outline the foundational work by Pierce and Sangiorgi on typing systems for the π-calculus. Section 3 covers the Join Calculus, and a discussion on its type systems. The remaining sections cover security specific extensions of name-passing calculi. In Section 4 we review an extension of the π-calculus with a new construct for group creation, and study the impact of the new primitive in enforcing secrecy. In Section 5 we discuss the *security π-calculus*, a typed version of the asynchronous π-calculus, which applies type based techniques provide security resource access control and information flow security guarantees. Section 6 gives a brief outline of a value passing extension of CCS, known as CryptoSPA, with cryptographic primitives. Finally, Section 7 covers the spi-calculus, and its typing system(s) for secrecy. Each section includes pointers to further important work in the literature relevant to each of the topics.

1 The Pi Calculus

The π-calculus is a way of describing and analyzing systems consisting of agents which interact among each other, and whose configuration is continually changing. The π-calculus emerged as the canonical model of concurrent computation, in much the same way as the λ-calculus has established itself as the canonical model of functional computation.

* Research supported by 'MyThS: Models and Types for Security in Mobile Distributed Systems', EU FET-GC IST-2001-32617.

R. Focardi and R. Gorrieri (Eds.): FOSAD 2001/2002, LNCS 2946, pp. 91–138, 2004.

The λ-calculus emphasizes the view of computation as the process of taking arguments and yielding results. In the λ-calculus everything is a function, and computation is, essentially, the result of function application. *Concurrent* computation cannot be forced into this functional metaphor of computation without severe distortions: if anything, functional computation is a special case of concurrent computation, and one should reasonably expect to find the functional model represented within a general enough model of concurrency.

In the π-calculus, every term denotes a process – a computational activity running in parallel with other processes and possibly containing several independent subprocesses. Computation arises as a result of process interaction, which in turns is based on communication on named channels. Naming is, in fact, the pervasive notion of the calculus, for various reasons. Naming presupposes *independence*: one naturally assumes that the *namer* and the *named* are independent (concurrent) entities. Further, using a name, or address, is a prerequisite to the act of *communicating*, and of locating and modifying data.

Based on these observations, the π-calculus seeks ways to treat data-access and communication as the same thing: in doing so, it presupposes that naming of *channels* is primitive, while naming of *agents* is not. As we shall see, departing from this view, and extending the concept of naming to agents and *locations* is what led to the development of models of mobility on top of the π-calculus. As of now, however, we start looking at the π-calculus in itself.

1.1 Syntax and Operational Semantics

There are in fact several versions of the π-calculus. Here, we will concentrate on a very basic one, although polyadic: the differences with other versions are mostly orthogonal to our concerns. The syntax is given in Table 1.

We assume an infinite set of *names* to be used for values and communication channels, and an infinite set of variables. We let $a,b-p,q$ range over names and $x-z$ range over variables. In addition, we often reserve u and v to denote names or variables indistinguishably, whenever the distinction between the two notions may safely be disregarded.

We use a number of notation conventions: $\tilde{x} : \tilde{T}$ stands for $x_1 : T_1, \ldots, x_k : T_k$, and we omit trailing dead processes, writing $\overline{u}\langle N \rangle$ for $\overline{u}\langle N \rangle.\mathbf{0}$ and $u(\tilde{x} : \tilde{T})$ for $u(\tilde{x} : \tilde{T}).\mathbf{0}$. The empty tuple plays the role of synchronization messages. The input prefix and the restriction operator are binders: the notations $fn(P)$ and $fv(P)$ indicate, respectively, the set of free names and free variables of the process P: these notions are defined as usual. We assume identity for α-convertible terms throughout, and we often omit type annotations on the two binders whenever irrelevant to the context in question.

The syntactic form $\mathbf{0}$ denotes the inert process, which does nothing. $u(x : T)P$ is a process that waits to read a value on the channel u: having received a value, say M, it behaves as P with every free occurrence of x substituted by M. Dually, $\overline{u}\langle M \rangle.P$ is a process that sends a value M on channel u and then behaves as P. The syntax suggests that output, as input, is *synchronous*, hence blocking: before continuing as P

Table 1 Pi calculus (typed) syntax

Expressions	M,N	$::=$	bv	basic value
		$\vert$	$a,\dots,p$	name
		$\vert$	$x,\dots,z$	variable
		$\vert$	$(M_1,\dots,M_k)$	tuple, $k \geqslant 0$
Processes	P,Q,R	$::=$	$\mathbf{0}$	stop
		$\vert$	$\overline{u}\langle N\rangle.P$	output
		$\vert$	$u(\tilde{x}:\tilde{T}).P$	input
		$\vert$	$(\nu a:T)P$	restriction
		$\vert$	$P \mid P$	composition
		$\vert$	$!P$	replication

the process the output $\overline{u}\langle M\rangle$ must be consumed by another process running in parallel[1]. The restriction form $(\nu a:T)P$ declares a new, *fresh* name a local to P. $P \mid Q$ denotes the parallel composition of two subprocesses P and Q. Finally, $!P$ stands for an infinite number of (parallel) copies of P.

The operational semantics of the π-calculus is defined in terms of two relations: a *structural equivalence* relation on process terms that allows the rearrangement of parallel compositions, replications and restrictions so that the participants in a communication can be brought into immediate proximity; and a *reduction* relation that describes the act of communication itself.

Structural Congruence is defined as the least congruence relation that is closed under the following rules:

1. $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$, $P \mid \mathbf{0} \equiv P$
2. $(\nu a)\mathbf{0} \equiv \mathbf{0}$, $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$
3. $(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q$ if $a \notin fn(P)$
4. $!P \equiv !P \mid P$

The one-step reduction relation $P \longrightarrow Q$ is the least relation closed under rules in Table 2.

The notation $P\{x_1 := M_1, \dots x_k := M_k\}$ indicates the simultaneous substitution of M_i for each free occurrence of the variable x_i in P, for $i \in [1..k]$. We assume that substitution maps variables to names (or else unstructured values). In other words, the substitution $\{x_1 := M_1, \dots x_k := M_k\}$ is only defined when each of the M_i is either a name or a basic value. In all other cases it is undefined.

The rule (COMM) is the core of reduction relation, as it defines the effect of synchronization between two processes on a channel. The rules (STRUCT) complete the definition. Notice that reduction is possible under a restriction, but not under either

[1] There exists an *asynchronous* variant of the calculus in which output is non-blocking. We will discuss it briefly below, and return on it in later sections, when discussing some of the derivative calculi.

Table 2 Reduction Relation

(COMMUNICATION)

$$n(x_1 : T_1,\dots,x_k : T_k)P \mid \overline{n}\langle M_1,\dots,M_k\rangle.Q \;\longrightarrow\; P\{x_1 := M_1,\dots,x_k := M_k\} \mid Q$$

(STRUCTURAL RULES)

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \frac{P \longrightarrow P'}{(\nu a)P \longrightarrow (\nu a)P'} \qquad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

of the two input and output prefix forms. It is also instructive to comment on the last structural rule for reduction, that connects the relations of reduction and structural congruence, and specifically on the interplay between the reduction rule (COMM) and the structural rule $(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q$ if $a \notin fn(P)$, known as the rule of *scope extrusion*. If we read the equivalence from left to right there is nothing surprising: since the name a does not occur free in P, restricting a on this process is vacuous, and we may safely move the restriction to Q without changing (at least intuitively) the meaning of the term. When used from right to left, instead, the equivalence enables the communication of private names. Consider the term $c(x).P \mid (\nu a)\overline{c}\langle a\rangle.Q$. In their current form, the two parallel processes may not communicate. However, we may use the congruence rules to rearrange the term as in $(\nu a)(c(x).P \mid \overline{c}\langle a\rangle.Q)$, and then use (COMM) to reduce it to $P\{x := a\} \mid Q$. By effect of the reduction, the name a, which was private to Q, has now been communicated to P. Interestingly, the name a may very well be the name of a channel, which implies that the reduction has the effect of establishing a new communication link between the two processes P and Q. Also note that the new link is now *private* to P and Q, and will remain so as long as the two processes do not communicate it to third parties.

This simple example shows that the combination of scope extrusion and communication provides a very powerful mechanism for:

- dynamically changing the topological structure of a system of processes, by creating new, fresh, communication links.
- establishing *private*, hence *secure* communication links among the principals of the system.

The ability to represent dynamically changing system topologies is the distinctive feature of the π-calculus with respect to previous CCS-like calculi for concurrency. The possibility of establishing private channels, in turn, makes the π-calculus a good foundation for studying formal models of security protocols. We briefly illustrate this potential of the π-calculus with a simplified version of the protocol known as the *Wide Mouthed Frog* protocol. In this version, we have two principals A and B (the outfamous Alice and Bob), willing to exchange secret data M, and a server S, that mediates their communication:

Message 1: $A \rightarrow S\;\; c_{AB}$ on c_{AS}

Message 2: $S \rightarrow B\;\; c_{AB}$ on c_{BS}

Message 3: $A \rightarrow B\;\; M$ on c_{AB}

Initially, each one of A and B shares a channel with S. A sends to S a secret channel that it wishes to use for communicate with B; S sends this channel to B and then A and B may communicate. The π-calculus formulation of the protocol is just as direct, but now formal:

$$\begin{aligned}
A &\triangleq (\nu\, c_{AB})\overline{c_{AS}}\langle c_{AB}\rangle.\overline{c_{AB}}\langle M\rangle \\
S &\triangleq c_{AS}(x).\overline{c_{BS}}\langle x\rangle \\
B &\triangleq c_{BS}(x).x(y).P\{y\}
\end{aligned}$$

The notation $P\{y\}$ is used here simply to emphasize that P will do something with the message it receives on the input channel x. The example shows how a secret channel may be established for communication, and relies critically on scope extrusion: the scoping rules guarantee that the context in which the protocol is executed (i.e. any process running in parallel with A, B and S) will not be able to access the secret channel c_{AB}, unless of course any of the principals involved in the protocol gives it away.

This use of private channels for secrecy is suggestive and effective in its simplicity. On the other hand, a problem with the π-calculus formulation of the protocol arises when we consider its implementation in a distributed environment. In that case, it is not realistic to rely only on the scope rules to ensure secrecy of names, as one also needs to prevent the context from having free access public channels over which private names are communicated. In our example, the name c_{AB} is secret, but to guarantee that secrecy is preserved through the protocol we should also envisage a mechanism for preventing the context from reading the name c_{AB} while it is communicated over the public channels c_{AS} and c_{BS}. Unfortunately, the π-calculus does not allow one to express the cryptographic operations that would typically be used for that purpose. This observation motivated the design of the cryptographic extension of the π-calculus known as the *spi* calculus [5, 10].

We conclude the description of the untyped π-calculus with a more complex example that illustrates the reduction semantics and the computational flavor of the calculus.

Example 1 (Memory Cells). A memory cell can abstractly be thought of as an object with private store s holding the cell value, and two methods *get* and *put* for reading and writing the contents of the cell. In the π-calculus, this can be represented as a process consisting of three parallel subprocesses like the ones displayed below:

$$\begin{aligned}
\mathsf{cell}(n) ::= (\nu s)(&\overline{s}\langle n\rangle \\
&\mid\ !\,get(y).s(x).(\overline{s}\langle x\rangle \mid \overline{y}\langle x\rangle) \\
&\mid\ !\,put(y,v).s(x).(\overline{s}\langle v\rangle \mid \overline{y}\langle\rangle))
\end{aligned}$$

$\mathsf{cell}(n)$ declares the private name s representing the physical location holding the value n, and provides the two handlers for serving the "get" and "put" requests on its contents. Both the handlers are implemented as replicated processes, to make it possible to serve multiple requests. Each request is served by first spawning a fresh copy of the corresponding handler by means of the congruence rule $!P \equiv P \mid !P$.

The intuition is as follows. To read the cell contents, a user sends a "get" request by transmitting, over the channel *get*, the name of a channel where it waits for the result of the request. Upon receiving the channel name, the "get" handler inside the cell consumes the current cell value, and then reinstates it while also copying it to the channel it received from the user. The protocol for serving a “put” request is similar. The cell’s “put” handler waits for a for value v: once v is received, the handler consumes the current cell value and writes v on the private channel s. There is a further subtlety, however, in that the “put” handler inside the cell also expects an “ack” channel from the user, which it uses to signal the completion of the protocol to the user. This may be required by a user that, say, increments the cell value and then reads the new value to print it: before reading the value, the user may use the ack channel to make sure it prints the new cell value, the one resulting from the increment.

Here, we illustrate the reduction semantics with a simpler (and less realistic) user:

$$\mathsf{user}(v) ::= (\nu\, ack)(\overline{put}\langle ack, v\rangle.ack().(\nu\, ret)\overline{get}\langle ret\rangle.ret(x).\overline{print}\langle x\rangle)$$

The user first writes a new value and then reads the cell contents to print the returned value. Now consider the system $\mathsf{cell}(0) \mid \mathsf{user}(v)$. An initial phase of structural rearrangements brings the system in the form $(\nu s)(\nu\, ack)(\nu\, ret)(\dots)_{cell} \mid (\dots)_{user}$. Then the system $(\dots)_{cell} \mid (\dots)_{user}$ evolves as follows: we omit the application of congruence rules and, at each reduction step, we only display the subterms that are relevant to the reduction in question:

$$\begin{aligned}
&(\overline{s}\langle 0\rangle \mid (\mathbf{put}(\mathbf{y},\mathbf{v}).s(x).\dots \mid \dots))_{cell} \mid (\overline{\mathbf{put}}\langle \mathbf{ack}, \mathbf{1}\rangle.\dots)_{user}\\
\longrightarrow\ &(\overline{\mathbf{s}}\langle \mathbf{0}\rangle \mid \mathbf{s}(\mathbf{x}).(\overline{s}\langle 1\rangle \mid \overline{ack}\langle\rangle) \mid \dots))_{cell} \mid (ack().\dots)_{user}\\
\longrightarrow\ &(\overline{s}\langle 1\rangle \mid \overline{\mathbf{ack}}\langle\rangle \mid \dots)_{cell} \mid (\mathbf{ack}().ret(x).\dots)_{user}\\
\longrightarrow\ &(\overline{s}\langle 1\rangle \mid (\mathbf{get}(\mathbf{y}).s(x).\dots))_{cell} \mid (\overline{\mathbf{get}}\langle \mathbf{ret}\rangle.\dots)_{user}\\
\longrightarrow\ &(\overline{\mathbf{s}}\langle \mathbf{1}\rangle \mid (\mathbf{s}(\mathbf{x}).(\overline{s}\langle x\rangle \mid \overline{ret}\langle x\rangle)\dots))_{cell} \mid ret(x).\overline{print}\langle x\rangle\\
\longrightarrow\ &(\overline{s}\langle 1\rangle \mid \overline{\mathbf{ret}}\langle \mathbf{1}\rangle\dots)_{cell} \mid \mathbf{ret}(\mathbf{x}).\overline{print}\langle x\rangle\\
\longrightarrow\ &(\overline{s}\langle 1\rangle \mid \dots)_{cell} \mid \overline{print}\langle 1\rangle \qquad \square
\end{aligned}$$

1.2 Further Reading

Starting with the original presentation [46], there is by now an extensive literature on the π-calculus, also in the form of introductory [45], and advanced [54]. Most versions of the π-calculus, including the one we have outlined here, are first-order in that they allow only names to be transmitted over channels. Higher-order versions of the calculus have been extensively studied by Sangiorgi [54].

2 Typing and Subtyping for the Pi Calculus

We have so far ignored the typing annotations occurring in the input and restriction binders. Now we take them more seriously, and look at the rôle of types in the calculus. There are in fact several reasons why types are useful for process calculi in general, and for the π-calculus in particular.

- The theory of the pure (untyped) π-calculus is often insufficient to prove some "expected" properties of processes. These properties arise typically from the programmer using names according to some intended principle or logical discipline which, however, does not appear anywhere in the terms of the pure π-calculus, and therefore cannot be used in proofs. Types bring the intended structure back into light, and therefore enhance formal reasoning on process terms: for instance, typed behavioral equivalences are easier to prove, based on the fact that only typed contexts need to be considered.
- types may be employed to ensure that process interaction happens only in type-consistent ways, and hence to enable static detection of run-time type errors. To exemplify, consider the following two terms:

  $$\overline{a}\langle b,c\rangle.P \mid a(x).Q \qquad\qquad \overline{a}\langle true\rangle.P \mid a(x).x(y).Q$$

  Both terms are, at least intuitively, ill-formed. The first reduces to the non-sensical process $\langle b,c\rangle(x).Q$, while the second to the ill-formed term $true(y).Q$ A simple arity check would be enough to rule out the first term as ill-formed. This, however, is not true of the second term.
- types can be useful for resource control. In the π-calculus, resources are channels, and the way that resources can be protected from unintended use is by *hiding* their names by means of the restriction operator. However, this is often too coarse a policy to enable effective resources control. In the untyped calculus, resource protection is lost when the resource name is transmitted, as no assumption can be made on how the recipient of the name will use the resource. Types may come to the rescue, as they can be employed to express and enforce a restrictive use of channels by associating them with *read* and/or *write* capabilities.

The study of type systems for process calculi originated from ideas by Milner [42, 43], based on the observation that channels used in system of processes naturally obey a discipline in the values they carry, that reflects their intended use. For instance, in the cell example above, the *ret* channel is used to communicate integers, while the *get* channel is used to communicate another channel (in fact, the *ret* channel). In Milner's original formulation, the cell example could be described by the following *sorting*:

$$\begin{array}{ll} ret\ : S_i & S_i \mapsto \mathsf{int} \\ get\ : S_g & S_g \mapsto (S_i) \\ ack : S_a & S_a \mapsto () \\ put : S_p & S_p \mapsto (S_a,()) \end{array}$$

The key idea, in types systems for the π-calculus, is that sorts, or types, are assigned only to channels, whereas processes are either well typed under a particular set of assumptions for their bound and free names and variables, or they are not. As we shall see, a different approach is possible, based on assigning more informative types to processes to describe various forms of process behavior. For the time being, however, we look at typing systems where the rôle of types is essentially that of describing (and prescribing) the intended use of channels.

The foundational work on type system by Pierce and Sangiorgi [47] was inspired by Milner's initial idea, which they elaborate in two dimensions. First they replace matching of types "by-name" with a more direct notion of structural matching, a technical modification that enables a substantively more concise and elegant presentation. Secondly, and more importantly, they employ types in a prescriptive manner to control and restrict access to channels. Their technique is based on associating channels with capabilities, and on introducing a notion of subtyping to gain additional control over the use processes can make of channels. The rest of this section gives an overview of their work. The reader is referred to [47] for full details.

2.1 Types

The structure of types is described by the following productions.

$$\begin{array}{rcll} \textit{Types}\ S,T & ::= & \mathsf{B} & \text{types of basic values} \\ & \mid & (T_1,\ldots,T_k) & \text{tuple, } k \geqslant 0 \\ & \mid & \mathsf{r}(T) & \text{input channel} \\ & \mid & \mathsf{w}(T) & \text{output channel} \\ & \mid & \mathsf{rw}(T) & \text{input/output channel} \end{array}$$

The type of a channel not only describes the type T of the values it carries, but also the kind of access the channel offers to its users. In the untyped calculus every channel is available for input and output: types help distinguishing, and restricting, the use of channels by associating them with access capabilities, providing users with the right to read from and/or write to a channel. The distinction between the two forms of access is reminiscent of a corresponding distinction that is made for the *reference types* in some functional programming languages. Reference types, that is, the types of mutable cells, are modeled with two different types: one for use of cells as "sources" of values, from which values can be read, and the other for cells as "sinks" where values can be placed. The same intuition applies to channels: channels of type $\mathsf{r}(T)$ may only be used as sources (i.e. for input), channels of type $\mathsf{w}(T)$ as sinks (i.e. for output), whereas channels of type $\mathsf{rw}(T)$ are input-output channels behaving both as sources and sinks.

To exemplify, $\mathsf{r}(\mathsf{int})$ is a read-only channel carrying values of type int. Since channels themselves are values, one can define a typed channel $c : \mathsf{rw}(\mathsf{r}(\mathsf{int}))$, conferring c the capability of sending and receiving values which in turn are read-only channels carrying integers.

2.2 Typing Rules

The typing rules are given in Table 3. They derive judgments in two forms: $\Gamma \vdash M : T$ stating that term M has type T, and $\Gamma \vdash P$ which simply says the process P is well-typed in *context*, or *type environment* Γ. A type environment Γ contains a set of type assumptions for the free names and variables occurring in P: equivalently, one may think of Γ as a finite map from names and variables to types.

The rules (BASE) and (TUPLE) should be self-explained. The (NAME) rule depends on the subtype relation $S \leqslant T$ which we discuss below: if the name (or variable) u is assumed to have type S in Γ, then any occurrence of that name in a process may also be

Table 3 Typing Rules for the Pi Calculus

Typing of Terms

$$\text{(B{\small ASE})}\ \frac{}{\Gamma \vdash bv : \mathsf{B}} \qquad \text{(N{\small AME})}\ \frac{\Gamma(u) = S \quad S \leqslant T}{\Gamma \vdash u : T} \qquad \text{(T{\small UPLE})}\ \frac{\Gamma \vdash M_i : T_i \quad i \in [1..k]}{\Gamma \vdash (M_1, \dots, M_k) : (T_1, \dots, T_k)}$$

Typing of Processes

$$\text{(I{\small NPUT})}\ \frac{\Gamma \vdash u : \mathsf{r}(\tilde{T}) \quad \Gamma, \tilde{x} : \tilde{T} \vdash P \quad \tilde{x} \cap \mathsf{Dom}(\Gamma) = \varnothing}{\Gamma \vdash u(\tilde{x} : \tilde{T}).P} \qquad \text{(O{\small UTPUT})}\ \frac{\Gamma \vdash u : \mathsf{w}(T) \quad \Gamma \vdash M : T \quad \Gamma \vdash P}{\Gamma \vdash \overline{u}\langle M \rangle.P}$$

$$\text{(D{\small EAD})}\ \frac{}{\Gamma \vdash \mathbf{0}} \qquad \text{(P{\small AR})}\ \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \text{(R{\small EPL})}\ \frac{\Gamma \vdash P}{\Gamma \vdash !P} \qquad \text{(R{\small ESTR})}\ \frac{\Gamma, a : T \vdash P \quad a \notin \mathsf{Dom}(\Gamma)}{\Gamma \vdash (\nu a : T)P}$$

Table 4 Core Subtype rules for channel types

$$\text{(S{\small UB} I{\small NPUT})}\ \frac{S \leqslant T}{\mathsf{r}(S) \leqslant \mathsf{r}(T)} \qquad \text{(S{\small UB} O{\small UTPUT})}\ \frac{T \leqslant S}{\mathsf{w}(S) \leqslant \mathsf{w}(T)} \qquad \text{(S{\small UB} IO/I)}\ \frac{}{\mathsf{rw}(T) \leqslant \mathsf{r}(T)} \qquad \text{(S{\small UB} IO/O)}\ \frac{}{\mathsf{rw}(T) \leqslant \mathsf{w}(T)}$$

typed at T provided that T is a super-type of S. The (INPUT) and (OUTPUT) rules ensure that channels are used consistently with their types. In the (INPUT) rule, the first premise requires that the channel from which input is requested provide a read capability and that the type of the input variables of the channel be consistent with the channel type. In addition, in order for the process $u(\tilde{x} : \tilde{T}).P$ to be well typed, the continuation P must also be well typed under the additional assumptions that the input variables $\tilde{x}$ are of the declared types. The rule (OUTPUT) has a similar reading. The remaining rules are easily explained: (PAR) and (REPL) are purely structural, (DEAD) states that the inert process is well typed, and (RESTR) is standard.

2.3 Subtyping

The subtype relation is central to the use of the type system to enforce access control over channels. The core subtyping rules are defined in Table 4. The subtype relation is the least reflexive and transitive relation that is closed under these rules, and a rule that extends the subtype relation homomorphically to tuples: $(S_1, \dots, S_k) \leqslant (T_1, \dots, T_k)$ if $S_i \leqslant T_i$ for all $i \in [1..k]$.

The two rules (SUB INPUT) and (SUB OUTPUT) are readily understood by analogy between channel types and reference types. Alternatively, one may think of a channel as a function: in its role as a source the channel returns a value, in its role as a sink it receives argument. Now, the two rules reflect the subtype relation for function types: covariant in their return type and contra-variant in their input. The rules (SUB IO/I) and (SUB IO/O) enable access control: any channel (which in the untyped calculus is always available for both input and output) may be associated with a more restrictive type (read-only or write-only) to protect it from misuse in certain situations. To illustrate the power of subtyping for resource access control, consider the following example from [47].

Example 2 (Access to a Printer). Suppose we have a system with a printer P and two clients C_1 and C_2. The printer provides a request channel p carrying values of some type T representing data to be printed on behalf of the clients. The system can be represented by the π-calculus process $(\nu p : \mathsf{rw}(T))(P \mid C_1 \mid C_2)$.

If we take, say, $C_1 \triangleq \overline{p}\langle j_1\rangle.\overline{p}\langle j_2\rangle.\ldots$, one would expect that the jobs $j_1, j_2, \ldots$ are received and processed, in that order, by the printer P. This is not necessarily the case, however, as C_2 might be not be willing to comply with the rules of the protocol. For instance, it competes with P to "steal" the jobs sent by C_1 and throws them away: $C_2 \triangleq\, !\,p(j : T).\mathbf{0}$.

One can prevent this kind of misbehavior by constraining the capabilities offered to C_1 and C_2 on the channel p: in the end, the clients should only write on p, whereas the printer should only read from it. We may therefore extend the system with an initialization phase that enforces this intended behavior on all the participants in the protocol. The initialization phase uses two channels, a and b, to communicate the name p to the printer and to the two clients, restricting the respective capabilities on p.

$$(\nu p : \mathsf{rw}(T))\ (\overline{a}\langle p\rangle.\overline{b}\langle p\rangle \mid a(x : \mathsf{r}(T)).P \mid b(y : \mathsf{w}(T)).(C_1 \mid C_2))$$

Notice that now p is a read-only channel within P and a write-only channel within C_1 and C_2. Assuming appropriate definitions for the processes P, C_1 and C_2, the system type checks, under the assumption $a,b : \mathsf{rw}(\mathsf{rw}(T))$, as the subtype relation ensures that $p : \mathsf{rw}(T)$ may legally be substituted for any $x : \mathsf{r}(T)$ or $y : \mathsf{w}(T)$.

2.4 Properties of the Type System

The type system satisfies the standard properties one expects: subject reduction and type safety. In functional languages, subject reduction guarantees that types are preserved during the computation. The result for the π-calculus is similar, and ensures that well-typedness is preserved by all the non-deterministic reductions of a process.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash P$ *and* $P \longrightarrow Q$*, then* $\Gamma \vdash Q$.

The proof of this result requires two auxiliary results. The first is the so-called subject-congruence theorem, stating that well-typedness is preserved by the relation of structural congruence.

Theorem 2 (Subject Congruence). *If* $\Gamma \vdash P$ *and* $P \equiv Q$*, then* $\Gamma \vdash Q$*. Dually, if* $\Gamma \vdash P$ *and* $Q \equiv P$*, then* $\Gamma \vdash Q$.

The second is the π-calculus version of the familiar substitution lemma from type system for the λ-calculus.

Theorem 3 (Substitution). *If* $\Gamma \vdash u : T$ *and* $\Gamma, x : T \vdash P$*, then* $\Gamma \vdash P\{x := u\}$.

Type safety is a more subtle issue. In functional calculi, proving type safety amounts to proving the so-called *absence of stuck states*, i.e. to show that the evaluation of well-typed terms either does not terminate, or returns a *value*, for a suitable notion of value. In the π-calculus, there is no notion of value, as computation is entirely based on interaction between processes, that do not return values. A notion of "stuck state" may nevertheless be formulated, and taken as the basic common denominator to different notions of type safety.

Theorem 4 (Basic Type Safety). *Assume* $\Gamma \vdash P$*, and* $P \longrightarrow Q$*. If* Q *contains a subterm*

$$c(x_1 : T_1, \ldots, x_n : T_n).Q_1 \mid \overline{c}\langle M_1, \ldots, M_k \rangle.Q_2$$

then all of the following hold true: c is a name or variable (i.e. not a constant of basic type), $k = n$ *and each of the* M_i *is a non-structured value.*

The theorem says essentially that process interaction happens in type-consistent ways, and never generates undefined substitutions. In addition, one may wish to prove other properties for reduction, and consequently richer notions of type safety. For instance, for the type system we have presented in the previous section, it can be proved that reduction of well-typed processes preserves guarantees that access to channels by processes is always consistent with the capabilities conferred to the channels by their types. We will discuss type-safety more formally in some of the calculi presented in later sections. Presently, we content ourselves with this informal formulation, and refer the interested reader to [47] for details on this richer notion of type safety.

2.5 Further Reading

The study of type systems for the π-calculus is currently very active, and has produced a large body of literature. Besides the work by Pierce and Sangiorgi we have reviewed in this section, and those we will discuss later on, an interesting pointer is to the work of Igarashi and Kobayashi [37] where a generic framework is proposed in which to understand several previous systems.

3 The Join Calculus

The *Join calculus* [29,30] is a variant of the asynchronous π-calculus [12,36] which combines restriction, reception, and replication in one construct, the *join receptor*: $J \triangleright P$. For example the definition

$$\texttt{def apply}\langle \texttt{f,x} \rangle \triangleright \texttt{f}\langle \texttt{x} \rangle \qquad (1)$$

defines a new name `apply` that receives two arguments and apply the first to the second. More precisely it receives a channel name that it bounds to f and a name that it bounds to x and sends the latter over the former. This is more formally shown by the following reduction:

$$\texttt{def apply}\langle \texttt{f,x} \rangle \triangleright \texttt{f}\langle \texttt{x} \rangle \texttt{ in apply}\langle \texttt{g,y} \rangle \quad \longrightarrow \quad \texttt{def apply}\langle \texttt{f,x} \rangle \triangleright \texttt{f}\langle \texttt{x} \rangle \texttt{ in g}\langle \texttt{y} \rangle$$

Table 5 The Join calculus

Processes	P,Q	$::=$	$x\langle\widetilde{V}\rangle$	Asynchronous message on x
			$\texttt{def}\ D\ \texttt{in}\ P$	Definition of D in P
			$P \mid Q$	Parallel Composition
			$\mathbf{0}$	Empty Process
Join patterns	J,J'	$::=$	$x\langle\tilde{y}\rangle$	Asynchronous reception on x
			$J \mid J'$	Joining messages
Definition	D,E	$::=$	$J \triangleright P$	Elementary clause
			$D \wedge E$	Simultaneous definition
Values	V,V'	$::=$	$\tilde{x}$	Names

Table 6 Received, defined and free variables

$dv(J \triangleright P)$	$= dv(J)$	$dv(D \wedge E)$	$= dv(D) \cup dv(E)$
$dv(T)$	$= \emptyset$	$dv(J \mid J')$	$= dv(J) \cup dv(J')$
$dv(x\langle\tilde{v}\rangle)$	$= \{x\}$		
$rv(x\langle\tilde{v}\rangle)$	$= \{u \mid u \in \tilde{v}\}$	$rv(J \mid J')$	$= rv(J) \uplus rv(J')$
$fv(J \triangleright P)$	$= dv(J) \cup (fv(P) - rv(J))$	$fv(D \wedge E)$	$= fv(D) \cup fv(E)$
$fv(\varepsilon)$	$= \emptyset$		
$fv(x\langle\tilde{v}\rangle)$	$= \{x\} \cup \{u \in \tilde{v}\}$		
$fv(\texttt{def}\ D\ \texttt{in}\ P)$	$= (fv(P) \cup fv(D)) - dv(D)$		
$fv(P \mid Q)$	$= fv(P) \cup fv(Q)$		
$fv(0)$	$= \emptyset$		

The syntax of the calculus is given in Table 5, where we assume names $x, y, \ldots$ to be drafted from an infinite set $\mathcal{N}$.

The only binding mechanism is the join pattern: the formal parameters which are received are bound in the guarded process. The *received* variables, $rv(J)$, are the names to which the messages sent are bound; the *defined* variables in a join pattern or a definition, $dv(J)$ and $dv(D)$, are the names which are bound by the definition. The *free* variables, $fv(P)$ and $fv(D)$, are all the names which are not bound. Received, defined and free variables can be easily defined as expected by structural induction (see Table 6).

It is important to notice that there is no linearity condition on the channel names in a composed join pattern: however, elementary join patterns are required to be linear, i.e. received variables are supposed to be pairwise distinct. A name is said to be *fresh* in a process when it is not free in it. In the following discussions a consistent use of names is assumed.

The operational semantics of the Join calculus is given using the chemical paradigm (structural rules $\rightleftharpoons$ plus reduction $\rightarrow$) in terms of the so called Reflexive Chemical Ab-

Table 7 The RCHAM

(str-join)	$\models P \mid Q$	$\rightleftharpoons$	$\models P, Q$
(str-def)	$\models \texttt{def}\ D\ \texttt{in}\ P$	$\rightleftharpoons$	$D_{\sigma_{dv}} \models P_{\sigma_{dv}}$
(red)	$\ldots \wedge J \triangleright P \wedge \ldots \models J_{\sigma_{rv}}$	$\rightarrow$	$\ldots \wedge J \triangleright P \wedge \ldots \models P_{\sigma_{rv}}$

Side conditions: in *(str-def)* σ_{dv} instantiates the names in $dv(D)$ to distinct fresh names; in *(red)* σ_{rv} substitutes the received variables $rv(J)$ with the values actually received

stract Machine (RCHAM) [29, 14] (see Table 7). States of the RCHAM are expression of the form $D \models P$, where P are the running processes and D are the (chemical) reactions.

Note that join patterns can be the parallel composition of different receptions, and that reduction takes place only when all the receptions synchronize. So for example the following receptor

$$\texttt{def ready}\langle\texttt{printer}\rangle \mid \texttt{print}\langle\texttt{file}\rangle \triangleright \texttt{printer}\langle\texttt{file}\rangle\ \texttt{in}\ P$$

reduces only when in P two (unbound) outputs on `ready` and `print` occur in parallel as for

$$\texttt{def ready}\langle\texttt{printer}\rangle \mid \texttt{print}\langle\texttt{file}\rangle \triangleright \texttt{printer}\langle\texttt{file}\rangle\ \texttt{in ready}\langle\texttt{gutenberg}\rangle \\ \mid \texttt{print}\langle\texttt{myths.ps}\rangle \mid Q$$

which reduces to

$$\texttt{def ready}\langle\texttt{printer}\rangle \mid \texttt{print}\langle\texttt{file}\rangle \triangleright \texttt{printer}\langle\texttt{file}\rangle\ \texttt{in gutenberg}\langle\texttt{myths.ps}\rangle \mid Q$$

The same behavior could be obtained by composing this definition with the definition (1):

$$\texttt{def apply}\langle\texttt{f,x}\rangle \triangleright \texttt{f}\langle\texttt{x}\rangle\ \wedge\ \texttt{ready}\langle p\rangle \mid \texttt{print}\langle f\rangle \triangleright \texttt{apply}\langle p, f\rangle$$

3.1 Typing

Let us again consider the definition of the expression (1). If we use $\langle T\rangle$ to denote the type of channels transporting values of type T, then `apply` has type $\langle\langle T\rangle, T\rangle$ for every type T. In words `apply` is a channel that transports pairs formed by a channel and a value that can be sent over that channel.

Note the polymorphic nature of the type of `apply`. This can be formally expressed by generalizing the type of `apply` into the following type schema: $\forall\alpha.\langle\langle\alpha\rangle,\alpha\rangle$. We saw before that join calculus provides synchronization between join patterns. Thus for instance a variant of `apply` that receives `f` and `x` from different sources can be defined as follows

$$\texttt{def fun}\langle\texttt{f}\rangle \mid \texttt{arg}\langle\texttt{x}\rangle \triangleright \texttt{f}\langle\texttt{x}\rangle$$

Table 8 Typing rules for the Join Calculus

$$\text{(INST)}\quad \frac{x : \forall\tilde{\alpha}.T \in A}{A \vdash x : T\{\tilde{\alpha} := \tilde{T}'\}} \qquad\qquad \text{(PAR)}\quad \frac{A \vdash P \quad A \vdash Q}{A \vdash P \mid Q}$$

$$\text{(MESSAGE)}\quad \frac{A \vdash x : \langle T_1,\dots,T_n\rangle \quad A \vdash v_i : T_i \quad (i = 1..n)}{A \vdash x\langle v_1,\dots,v_n\rangle} \qquad \text{(DEF)}\quad \frac{A,B \vdash D :: B \quad A, Gen(B,A) \vdash P}{A \vdash \mathtt{def}\ D\ \mathtt{in}\ P}$$

$$\text{(JOIN)}\quad \frac{A, y_{ij} : T_{ij}^{i=1..n, j=1..m_i} \vdash P}{A \vdash x_1\langle y_{1j}^{j=1..m_1}\rangle \mid \dots \mid x_n\langle y_{nj}^{j=1..m_n}\rangle \triangleright P :: x_i : \langle T_{i1},\dots T_{im_i}\rangle^{i=1..n}}$$

$$\text{(AND)}\quad \frac{A \vdash D_1 :: B_1 \quad A \vdash D_2 :: B_2}{A \vdash D_1 \wedge D_2 :: B_1, B_2}\ (B_1|_{\mathsf{Dom}(B_2)} = B_2|_{\mathsf{Dom}(B_1)})$$

According to what we said before `fun` and `arg` can be respectively typed as $\langle\langle\alpha\rangle\rangle$ and $\langle\alpha\rangle$. Observe, however, that `fun` and `arg` are correlated in their types as they must share the same type variable α. This forbids to generalize their types separately: if we assigned them the types $\forall\alpha.\langle\langle\alpha\rangle\rangle$ and $\forall\alpha.\langle\alpha\rangle$, then the correlation of the types of the two names defined in the same join pattern would be lost. In [14] this problem is handled by the definition of the generalization rule that forbids the generalization of type variables that appear free in the type of more than one co-defined name.

The type system of [14] is defined as follows:

$$\begin{array}{llll} \textit{Types} & T ::= \alpha \;\big|\; \langle T,\dots,T\rangle & \qquad \textit{Type Envs} & B ::= \varnothing \;\big|\; B, x : T \\ \textit{Schemas} & \sigma ::= T \;\big|\; \forall\alpha.\sigma & \qquad \textit{Schema Envs} & A ::= \varnothing \;\big|\; A, x : \sigma \end{array}$$

The type system includes three kinds of typing judgments:

$A \vdash u{:}T$ the name u has type T in A
$A \vdash P$ the process P is well typed in A
$A \vdash D :: B$ the definition D is well-typed in A with types B for its defined types

which are deduced by the typing rules in Table 8.

In that table, $Gen(B,A)$ is the generalization of the type environment B of the form $(x_i : T_i)^{i=1..n}$ with respect to the schema environment A: let $fv(A)$ be the set $\cup_{(s:\sigma)\in A} vars(\sigma)$ with $vars(\sigma)$ is the set of variables occurring in σ; let $B \setminus x$ be the environment B without the binding for x; then $Gen(B,A)$ is $(x_i : \forall(fv(T_i) - fv(A,(B \setminus x_i))).T_i)^{i=1..n}$.

With the exception of (DEF) all the rules are straightforward, insofar as they are almost directly inspired by the typing rules for polymorphic (poliadic) λ-calculus. (INST) assigns to a variable x any type that is an instance of the type schema associated to the x in A; (PAR) is straightforward; (MESSAGE) checks that the types of the actual parameters match the type of the channel the parameters are sent over; (JOIN) checks that the guarded process P is typable under the assumption that the types of the formal parameters of the join patterns match those of the corresponding channels, and associates to the definition the type environment of its declared names; (AND) associates to the composition of two definitions the composition of the type environments of their declared names, provided that the two definitions do not declare a common name. Finally (DEF) is the most technical rule: first it checks the typing of the definition D under the type environment produced by D. This allows recursive definitions; second it checks the well typing of P under the generalization of the types of the new definition with respect to A. In particular the generalization takes into account the problem of sharing we hinted in the beginning of the section. Therefore for every constraint $x{:}T \in B$ the generalization does not generalize all the free type variables of T but, instead, only those free variables that are not shared with a previous definition or with a parameter of the actual join pattern.

3.2 Properties of the Type System

Soundness. The soundness of the type system is obtained by proving subject reduction and basic type safety (corresponding to Theorem 4 for π-calculus.)

Theorem 5 (Subject Reduction). *If $A \vdash P$ and $P \longrightarrow Q$, then $A \vdash Q$.*

Definition 1. *A process of the form* `def` $D \wedge J \triangleright$ `in` $Q \mid x\langle\tilde{v}\rangle$ *is wrong if J contains a message $x\langle\tilde{y}\rangle$ where $\tilde{y}$ and $\tilde{v}$ have different arities.*

Theorem 6 (Basic Type Safety). *If $A \vdash P$ then P is not wrong.*

The composition of the previous two theorems ensures that well typed processes never go wrong.

Type Inference. Finally, there exists an algorithm that for every typable process returns the most general schema environment under which the process can be typed, while it fails if it is applied to a process that is not typable.

3.3 Further Reading

In [7] Abadi, Fournet, and Gonthier define the sjoin-calculus, that extends the join calculus with constructs for encryption and decryption and with names that can be used as keys, nonces, or other tags. This extension is very reminiscent of the the way the spi-calculus (see section 7) extends the π-calculus: as a matter of fact, the name sjoin was chosen in analogy with spi. The authors also show how to translate sjoin into a lower-level language that includes cryptographic primitives mapping communication on secure channels into encrypted communication on public channels. A correctness theorem for the translation ensures that one can reason about programs in sjoin without mentioning the cryptographic protocols used to implement them in the lower-level implementation.

In [17] Conchon and Pottier advocate that the the type system of [14], that forbids the generalization of any type variable that is shared between two jointly defined names (such as `fun` and `arg`), is overly restrictive when one wants to use types in a descriptive – rather then prescriptive – way. To that end they switch from the system of [14] in which the generalization is performed on syntactic criteria, to a richer type system based on constraints and where the generalization is more "semantic" (polymorphic types are interpreted as particular sets of monomorphic types) and fairly natural. However, rather surprisingly, the new generalization criterion hinders type inference as it results very difficult (perhaps impossible) to infer a most general type. As a result they propose a more restrictive (and syntactic) criterion that, while it allows type inference, it is closer to the original system of [14].

In his PhD. thesis [15] Conchon extends the type system of JOIN(X) with information-flow annotations that ensure a noninterference property based on bisimilarity equivalences. The new systems thus obtained can detect, for instance, information flow caused by contentions on distributed resources, which are not detected, in a satisfactory way, when using testing equivalences. The achievement is however limited by the fact that equivalences, rather than congruences, are considered.

A more in depth study of bisimulation for the join calculus can be found in [11].

In all these variants, join remains a concurrent calculus. In [31] the authors define the Distributed Join Calculus that extends join calculus essential with locations, migration, and failure. The new calculus allows one to express mobile agents roaming on the net, that is, that autonomously move from some node to a different node where they resume their current execution. Distributed join is also the core of the distributed language *jocaml*[16].

4 The Pi Calculus with Groups

In Section 1 (and we will see it also in Section 7) we discussed the importance of scope extrusion for secrecy. However, inattentive use of scope extrusion may cause secrets to be leaked. Consider a process P that wants to create a private name x. In the pi-calculus this can be done by letting P evolve into a configuration $(\nu x)P'$, where the channel x is intended to remain private to P'. This privacy policy is going to be violated if the system then evolves into a situation such as the following, where p is a public channel known to an hostile process (opponent) running in parallel with P.

$$p(y).O \mid (\nu x)(\overline{p}\langle x\rangle \mid P') \tag{2}$$

In this situation, the name x is about to be sent by P over the public channel p and received by the opponent. In order for this communication to happen, the rules of the pi-calculus, described in Section 1, require first an enlargement of the scope of x. After extrusion we have:

$$(\nu x)(p(y).O \mid \overline{p}\langle x\rangle \mid P') \tag{3}$$

Now, x can be communicated over p, and the opponent acquires the secret.

The private name x has been leaked to the opponent by a combination of two mechanisms: the output instruction $\overline{p}\langle x\rangle$ and the extrusion of (νx). It seems that we need to restrict either communication or extrusion. Since names are dynamic data in the pi-calculus, it is not easy to say that a situation such as $\overline{p}\langle x\rangle$ (sending x on a channel known to the opponent) should not arise, because p may be dynamically obtained from some other channel, and may not occur at all in the code of P.

The other possibility is to prevent extrusion, which is a necessary step when leaking names outside their initial scope. However, extrusion is a fundamental mechanism in the pi-calculus: blocking it completely would also block innocent communications over p.

A natural question is whether one could somehow declare x to be private, and have this assertion statically checked so that the privacy policy of x cannot be violated. To this end, in [13] authors add an operation of group creation to the typed pi-calculus, where a group is a type for channels. Group creation is a natural extension of the sort-based type systems developed for the pi-calculus (see Section 1). However, group creation has an interesting and subtle connection with secrecy. Creation of fresh groups has the effect of statically preventing certain communications, and can block the accidental or malicious leakage of secrets.

Intuitively, no channel belonging to a fresh group can be received by processes outside the initial scope of the group, even if those processes are untyped. Crucially, groups are not values, and cannot be communicated; otherwise, this secrecy property would fail.

Starting from the typed pi-calculus, we can classify channels into different groups (usually called sorts). We could have a group G for our private channels and write $(\nu x{:}G)P$ to declare x to be of sort G. However, if groups are global (as usually happens with sorts in standard pi-calculus type systems), they do not offer any protection because an opponent could very well mention G in an input instruction, and leakage can thus be made to typecheck:

$$p(y : G).O \mid (\nu x{:}G)(\overline{p}\langle x\rangle \mid P') \tag{4}$$

In order to guarantee secrecy, the group G itself should be secret, so that no opponent can input names of group G, and that no part of the process P can output G information on public channels.

In general we want the ability to create fresh groups on demand, and then to create fresh elements of those groups. To this end, we extend the pi-calculus with an operator, $(\nu G)P$, to dynamically create a new group G in a scope P. Although group creation is dynamic, the group information can be tracked statically to ensure that names of different groups are not confused. Moreover, dynamic group creation can be very useful: we can dynamically spawn subsystems that have their own pool of shared resources that cannot interfere with other subsystems.

Consider the following process, where $G[\,]$ is the type of a channel of group G:

$$(\nu p{:}U)\ (p(y{:}T).O \mid (\nu G)(\nu x{:}G[\,])\overline{p}\langle x\rangle) \tag{5}$$

Here an attempt is made again to send the channel x over the public channel p. Fortunately, this process cannot be typed: the type T would have to mention G, in order to receive a channel of group G, but this is impossible because G is not known in the global scope where p has been declared.

The construct (νG) has extrusion properties similar to (νx), which are needed to permit legal communications over channels unrelated to G channels, but these extrusion rules prevent G from being confused with any groups mentioned in T.

Untyped Opponents. Let us now consider the case where the opponent process is untyped or, equivalently, not well-typed. This is intended to cover the situation where an opponent can execute any instruction without being restricted by static checks such as type checking or bytecode verification. For example, the opponent could be running on a separate, untrusted, machine. Let consider again the previous process, where we remove typing information from the code of the opponent, since an opponent does not necessarily respect the typing rules. The opponent now attempts to read any message transmitted over the public channel, no matter what its type is.

$$(\nu p{:}U)(p(y).O \mid (\nu G)(\nu x{:}G[\,])\overline{p}\langle x\rangle) \tag{6}$$

The untyped opponent will not acquire secret information by cheating on the type of the public channel. The fact that the process P is well typed is sufficient to ensure secrecy, even in the presence of untyped opponents. This is because, in order for P to leak information over a public channel p, the output operation $\overline{p}\langle x\rangle$ must be well typed. The name x can be communicated only on channels whose type mentions G. So the output $\overline{p}\langle x\rangle$ cannot be well typed, because then the type U of p would have to mention the group G, but U is not in the scope of G.

We have thus established, informally, that a process creating a fresh group G can never communicate channels of group G to an opponent outside the initial scope of G, either because a (well typed) opponent cannot name G to receive the message, or, in any case, because a well typed process cannot use public channels to communicate G information to an (untyped) opponent. Thus, channels of group G are forever secret outside the initial scope of (νG). So, secrecy is reduced in a certain sense to scoping and typing restrictions. As we have seen, the scope of channels can be extruded too far, perhaps inadvertently, and cause leakage, while the scope of groups offers protection against accidental or malicious leakage, even though it can be extruded as well.

4.1 Syntax and Operational Semantics

We start showing the syntax of an asynchronous, polyadic, typed pi-calculus with groups and group creation. Types specify, for each channel, its group and the type of the values that can be exchanged on that channel.

$$\textit{Types} \quad T ::= G[T_1,\ldots,T_n] \qquad \text{polyadic channel in group } G$$

As usual, in a restriction $(\nu x{:}T)P$ the name x is bound in P, and in an input $x(\tilde{y} : \tilde{T}).P$, the names $y_1,\ldots,y_k$ are bound in P. In a group creation $(\nu G)P$, the group G is bound with scope P. Let $fn(P)$ be the set of free names in a process P, and let $fg(P)$,$fg(T)$ be the sets of groups free in a process P and a type T, respectively.

The operational semantics of the calculus is similar to that of the typed pi-calculus described in Section 1. Group creation is handled by the following new rules of structural equivalence and reduction:

Table 9 Typed pi-calculus with Groups

Expressions M,N	$::=$	$a,\dots,p$	name
	$\mid$	$x,\dots,z$	variable
Processes P,Q,R	$::=$	$\mathbf{0}$	stop
	$\mid$	$\overline{u}\langle\tilde{M}\rangle.P$	polyadic output
	$\mid$	$u(\tilde{x}{:}\tilde{T}).P$	polyadic input
	$\mid$	$(\nu G)P$	group creation
	$\mid$	$(\nu a{:}T)P$	restriction
	$\mid$	$P \mid P$	composition
	$\mid$	$!P$	replication

(Struct GRes GRes) $(\nu G_1)(\nu G_2)P \equiv (\nu G_2)(\nu G_1)P$

(Struct GRes Res) $(\nu G)(\nu x{:}T)P \equiv (\nu x{:}T)(\nu G)P$ if $G \notin fg(T)$

(Struct GRes Par) $(\nu G)(P \mid Q) \equiv P \mid (\nu G)P$ if $G \notin fg(P)$

(Red GRes) $(\nu G)P \longrightarrow (\nu G)Q$ if $P \longrightarrow Q$

Note that rule (Struct Gres Res) is crucial: it implements a sort of “barrier” between processes knowing a group name and processes that do not know it.

4.2 The Type System

Environments declare names and groups in scope during type-checking; we define environments, Γ, by $\Gamma ::= \emptyset \mid \Gamma,G \mid \Gamma,x{:}T$. We define four typing judgments: first, $\vdash \Gamma$ means that Γ is well formed; second $\Gamma \vdash T$ means that T is well formed in Γ; third, $\Gamma \vdash x : T$ means that $x : T$ is in Γ, and that Γ is well formed; and, fourth, $\Gamma \vdash P$ means that P is well formed in the environment Γ. Typing rules are collected in Table 10.

Properties of the Type System. A consequence of our typing discipline is the ability to preserve secrets. In particular, the subject reduction property, together with the proper application of extrusion rules, has the effect of preventing certain communications that would leak secrets. For example, consider the process (4) at the beginning of this section:

$$(\nu p{:}U)(p(y{:}T).O \mid (\nu G)(\nu x{:}G[\,])\overline{p}\langle x\rangle)$$

In order to communicate the name x (the secret) on the public channel p, we would need to reduce the initial process to the following configuration:

$$(\nu p{:}U)(\nu G)(\nu x{:}G[\,])(p(y{:}T).O \mid \overline{p}\langle x\rangle)$$

If subject reduction holds then this reduced term has to be well-typed, which is true only if $p : H[T]$ for some H, and $T = G[\,]$. However, in order to get to the point of

Table 10 Typing rules for the pi-calculus with groups

$$\text{(Env Empty)}\quad \frac{}{\vdash \emptyset} \qquad \text{(Env } u\text{)}\quad \frac{\Gamma \vdash T \quad u \notin dom(\Gamma)}{\vdash \Gamma, u : T} \qquad \text{(Env Group)}\quad \frac{\vdash \Gamma \quad G \notin dom(\Gamma)}{\vdash \Gamma, G}$$

$$\text{(Type Chan)}\quad \frac{G \in dom(\Gamma) \quad \Gamma \vdash T_1 \ldots \Gamma \vdash T_n}{\Gamma \vdash G[T_1, \ldots, T_n]} \qquad \text{(Project)}\quad \frac{\vdash \Gamma', x : T, \Gamma''}{\Gamma', x : T, \Gamma'' \vdash x : T} \qquad \text{(GRes)}\quad \frac{\Gamma, G \vdash P}{\Gamma \vdash (\nu G)P}$$

$$\text{(Input)}\quad \frac{\Gamma \vdash M : G[T_1, \ldots, T_n] \quad \Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash P}{\Gamma \vdash M(x_1 : T_1, \ldots, x_n : T_n)P} \qquad \text{(Res)}\quad \frac{\Gamma, n : T \vdash P}{\Gamma \vdash (\nu n : T)P} \qquad \text{(Dead)}\quad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{0}}$$

$$\text{(Output)}\quad \frac{\Gamma \vdash M : G[T_1, \ldots, T_n] \quad \Gamma \vdash N_1 : T_1 \ldots \Gamma \vdash N_n : T_n}{\Gamma \vdash \overline{M}\langle N_1, \ldots, N_n \rangle} \qquad \text{(Par)}\quad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \text{(Repl)}\quad \frac{\Gamma \vdash P}{\Gamma \vdash !P}$$

bringing the input operation of the opponent next to the output operation, we must have extruded the (νG) and the $(\nu x{:}G[\,])$ binders outward. The rule (Struct Gres Res), used to extrude (νG) past $p(y{:}T).O$, requires that $G \notin fg(T)$. This contradicts the requirement that $T = G[\,]$.

Proposition 1 (Subject Congruence). *If* $\Gamma \vdash P$ *and* $P \equiv Q$*, then* $\Gamma \vdash Q$.

Proposition 2 (Subject Reduction). *If* $\Gamma \vdash P$ *and* $P \longrightarrow Q$*, then* $\Gamma \vdash Q$.

The formalization of secrecy is inspired by Abadi's definition [2]: a name is kept secret from an opponent if after no series of interactions is the name transmitted to the opponent. We model the external opponent simply by the finite set of names S known to it. A complete formalization of this notion of security can be found in [13], here we only overview the main theorem and its proof. The following theorem expresses the idea that in the process $(\nu G)(\nu x{:}G[T_1, \ldots, T_n])P$, the name x of the new group G is known only within P (the scope of G) and hence is kept secret from any opponent able to communicate with the process (whether or not the opponent respects our type system). Let $erase(P)$ be the process obtained from P by erasing type annotations and new-group creations. Let S be a set of names, we say that a process P *preserves the secrecy of* x *from* S if P will never communicate the name x to an opponent initially knowing the names in S.

Theorem 7 (Secrecy). *Suppose that* $\Gamma \vdash (\nu G)(\nu x{:}T)P$ *where* $G \in fg(T)$*. Let* S *be the names occurring in* $dom(\Gamma)$*. Then the erasure* $(\nu x)erase(P)$ *of* $(\nu G)(\nu x{:}T)P$ *preserves the secrecy of the restricted name* x *from* S.

The proof of the secrecy theorem (see [13]) is based on an auxiliary type system that partitions channels into untrusted channels , with type Un and trusted ones, with type $Ch[T_1,\dots,T_n]$, where each T_i is either a trusted or untrusted type. The type system insists that names are bound to variables with the same trust level (that is, the same type), and that no trusted name is ever transmitted on an untrusted channel. Hence an opponent knowing only untrusted channel names will never receive any trusted name.

In particular, for any group G, we can translate group-based types into the auxiliary type system as follows: any type that does not contain G free becomes Un, while a type $H[T_1,\dots,T_n]$ that contains G free is mapped onto $Ch[\langle\!\langle T_1 \rangle\!\rangle_G,\dots,\langle\!\langle T_n \rangle\!\rangle_G]$. This translation is proved to preserve typability. This implies that an opponent knowing only names whose type does not contain G free, will never be able to learn any name whose type contains G free. This is the key step in proving the secrecy theorem.

Finally, note that the typing rules constrain only the principals that want to protect their secrets from attackers. On the contrary, there are no restrictions on the code the attackers may run; we have in fact that any untrusted opponent may be type-checked as follows.

Lemma 1. *For all P, if $fn(P) = \{x_1,\dots,x_n\}$ then $x_1 : Un,\dots,x_n : Un \vdash P$.*

This is a distinctive property of the approach we discussed in this section, since it makes the type system suitable for reasoning about processes containing both trusted and untrusted subprocesses.

5 The Security Pi Calculus

The security π-calculus is an extension of the π calculus defined by Hennessy and Riely [33] to study properties of *resource access* and *information flow* control in systems with *multilevel* security. Before discussing the security π-calculus, we first give a very brief overview of the underlying models of multilevel security.

5.1 Multilevel Security

Traditional models of security are centered around notions of *subjects* and objects, with the former performing accesses on the latter by *read* and *write* (as well as *append*, *execute*, ..., etc. in certain models) operations. Multilevel security presupposes a lattice of security levels, and every subject and object in the system is assigned a level in this lattice. Based on these levels, access to objects by subjects are classified as *read-up* (resp. *read-down*) when a subject attempts to read an object of higher (resp. lower) level, and similarly for write accesses. Relying on this classification, *security policies* are defined to control access to objects by subjects and, more generally flow of information among the subjects and objects of the system.

An important class of security policies are the so-called *Mandatory Access Control* (MAC) policies, among which notable examples are *defense* security and *business* security. Defense security aims at protecting confidentiality of data by preventing flow of information from *high*, privileged, subjects to *low*, users. This is accomplished by forbidding read-up's and write-down's to objects: low-level users may not read confidential information held in high-level documents, and high-level principals may not write

Table 11 Syntax: The Security pi calculus

Expressions	M, N	$::=$	$\ldots$	as in Table 1
Processes	P, Q, R	$::=$	$\mathbf{0}$	stop
		$\mid$	$\overline{u}\langle N\rangle$	asynchronous output
		$\mid$	$u(\tilde{x} : \tilde{T}).P$	input
		$\mid$	$(\nu a : T)P$	restriction
		$\mid$	$P \mid P$	composition
		$\mid$	$!P$	replication
		$\mid$	$[P]_\sigma$	process at clearance σ

information on low-level objects that may be available to low-level users. Business security, on the other hand, centers around integrity, and a weaker form of confidentiality, and provides guarantees that low-level users have no direct access to secret high-level data, either in read or write mode.

Enforcing confidentiality and integrity often requires further constraints to prevent flow of sensitive information to non-authorized subjects arising from subtle and hidden ways of transmitting information, viz. *covert channels*: these may be established in several ways, via system-wide side effects on shared system resources. The prototypical example of covert channel is realized by means of the “file system full” exception. Suppose that a process fills the file system, and then deletes a 1-bit file: further attempts by that process to write that file will inform it of any two (high-level) users exchanging 1-bit information via the file system.

5.2 Syntax of the Security Pi-Calculus

The security π-calculus is based on the *asynchronous* variant of the π calculus. The choice of asynchronous output is motivated by security reasons, as synchronous output is more prone to covert channels and implicit flow of information. We will return to this point later: as of now, we proceed with our discussion on the asynchronous π-calculus and its extension with security.

There are different ways that the asynchronous π-calculus can be defined: for instance, one may define it by relying on the same syntax given in Table 1, and by extending the relation of structural congruence with the new rule: $a\langle M\rangle.P \equiv a\langle M\rangle.\mathbf{0} \mid P$. This rule effectively leads to an asynchronous version of the output operation, as it allows the process P to reduce, hence evolve, independently of the presence of an input process consuming the value M sent over the channel a.

Here, however, we will adhere to the more standard practice and use a different syntax in which output on a channel is defined as a process rather than a prefix. The syntax of the security π-calculus results from the syntax of the π-calculus from this change and from introducing a new construct for processes.
As anticipated, the output construct is now a process rather than a prefix: this is all that is needed to account for asynchrony. The new syntactic form $[P]_\sigma$ denotes a process P running at security level σ; it has no real computational meaning, as the notion of reduction is not affected by this construct. It is, however, relevant to the definition of

the instrumented semantics that we will introduce to capture a notion of run-time error resulting from *security violations.*

In the instrumented semantics, we view processes as playing the rôle of subjects, while channels are naturally associated with the rôle of objects that processes access in read and write mode. Security levels are associated with channels by enriching the structure of channel types: besides associating input-output capabilities with each name, channel types also include a security level.

The structure of the types is defined in terms of a lattice *SL* of security levels. We let Greek letters like $\delta,\sigma,\rho,\ldots$ range over the elements of this lattice: the top and bottom elements are denoted by $\top$ and $\bot$ as usual. To enhance flexibility, the structure of types allows different security levels to be associated with the input and output capabilities for a channel. Thus, if S and T are types, channels types may be structured as shown in the following to examples:

- $\{\mathsf{w}_\bot(S),\mathsf{r}_\top(T)\}$: the type of channels where low processes can write (values of type S), and only high processes can read (values of type T). This typing is appropriate for a mailbox, where everybody should be allowed to write but only the owner should be granted permission to read.
- $\{\mathsf{w}_\top(S),\mathsf{r}_\bot(S)\}$: the type of channels where anybody can read, but only high processes can write. This typing is typical of an information channel, where privileged users write information for everyone to read.

We give a formal definition of types in Section 5.5. Before that, we define the operational semantics and formalize a notion of security error

5.3 Reduction Semantics

The operational semantics is given, as for the π-calculus, in terms of the two relations of structural congruence and reduction. Structural congruence is defined by the following extension of the corresponding relation for the π-calculus:

Table 12 Structural congruence

π-Calculus Rules for Structural Equivalence.

1. $P \mid Q \equiv Q \mid P,\, P \mid (Q \mid R) \equiv (P \mid Q) \mid R,\, P \mid \mathbf{0} \equiv P$
2. $(\nu a)\mathbf{0} \equiv \mathbf{0}$, $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$
3. $(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q$ if $a \notin fn(P)$
4. $!P \equiv !P \mid P$

Security π-Calculus Specific Rules.

5. $[P \mid Q]_\sigma \equiv [P]_\sigma \mid [Q]_\sigma$
6. $[(\nu x : T)P]_\sigma \equiv (\nu x : T)[P]_\sigma$
7. $[[P]_\rho]_\sigma \equiv [P]_{\sigma\sqcap\rho}$

In addition, as for the π-calculus, the definition includes the structural rules that make $\equiv$ a congruence. The rules 5 and 6 are not surprising. In rule 7, the notation $\rho \sqcap \sigma$ indicates the *greatest lower bound* between ρ and σ in the lattice of security levels. Based on this relation, the reduction relation is defined as follows:

Table 13 Reduction Relation

(COMM)	$\overline{a}\langle\tilde{M}\rangle \mid a(\tilde{x}:\tilde{T})P \longrightarrow P\{x_1 := M_1,\ldots,x_k := M_k\}$
(COMM$_\rho$)	$[\overline{a}\langle\tilde{M}\rangle]_\sigma \mid [a(\tilde{x}:\tilde{T}).P]_\rho \longrightarrow [P\{x_1 := M_1,\ldots,x_k := M_k\}]_\rho$
(STRUCT)	$\dfrac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \quad \dfrac{P \longrightarrow P'}{[P]_\sigma \longrightarrow [P']_\sigma} \quad \dfrac{P \longrightarrow P'}{(\nu a:T)P \longrightarrow (\nu a:T)P'}$ $\dfrac{P \equiv P' \quad P' \longrightarrow Q \quad Q' \equiv Q}{P \longrightarrow Q}$

The rule (COMM) is the asynchronous variant of the reduction rule for communications from the π-calculus. The rule (COMM$_\rho$) is the corresponding rule for processes with a clearance: as we noted, the presence of the security level does not affect the computational behavior of processes. On the other hand, it is the basis for the formalization of run-time security error.

5.4 Security as Resource Access Control

Security violations occur against a given *security policy*, which is formalized in the calculus in terms of (i) a mapping from resources (i.e. names and values) to their types, and of (ii) an auxiliary reduction relation that underlines the import of the policy by defining what it means to violate it. As a first example, given a mapping Γ, one may enforce a policy for resource access control by stating that processes at clearance σ should only have access to channels and values at security level up to (and including) σ. This can be formalized by the following additional reductions:

Table 14 Security Violation

(E-INPUT)	$[n(\tilde{x}:\tilde{T}).P]_\rho \xrightarrow{\Gamma} \mathsf{err}$ if $\mathsf{r}_\sigma(\tilde{T}) \in \Gamma(n) \Longrightarrow \sigma \not\preceq \rho$
(E-OUTPUT)	$[\overline{n}\langle M\rangle]_\rho \xrightarrow{\Gamma} \mathsf{err}$ if $\mathsf{w}_\sigma(T) \in \Gamma(n) \Longrightarrow \sigma \not\preceq \rho$
(E-OUTVAL)	$[\overline{n}\langle M\rangle]_\rho \xrightarrow{\Gamma} \mathsf{err}$ if $M : \mathsf{B}_\sigma$ and $\sigma \not\preceq \rho$
(E-STRUCT)	$\dfrac{P \xrightarrow{\Gamma} \mathsf{err}}{P \mid Q \xrightarrow{\Gamma} \mathsf{err}} \quad \dfrac{P \xrightarrow{\Gamma} \mathsf{err}}{[P]_\sigma \xrightarrow{\Gamma} \mathsf{err}} \quad \dfrac{P \xrightarrow{\Gamma} \mathsf{err}}{(\nu a:A)P \xrightarrow{\Gamma} \mathsf{err}}$

The rule (E-INPUT) states that a process with clearance ρ can not read from channels that are not qualified by the security policy Γ, or that have security level higher than ρ. The rule (E-OUTPUT) defines dual conditions for errors resulting from an attempt to write on restricted channels. The rule (E-OUTVAL) states that a process with clearance ρ may only communicate a value along a channel if that value is not restricted from σ-level processes. In all three cases, the security violation is signalled by a reduction to the distinguished process term err. The remaining rules are purely structural, and propagate errors from a process to its enclosing terms.

We give two examples that illustrate the import of different security policies on the reduction semantics.

Example 3 (Resource Access Violations). Consider the process

$$P \triangleq [\overline{c}\langle a\rangle]_\top \mid [c(x).\overline{x}\langle 1\rangle]_\bot .$$

consisting of a high-level and a low-level processes communicating over a channel c, for which we define the security policy Γ as follows: $\Gamma(a) = A$, $\Gamma(c) = C$. First, assume that the two types A and C are defined as follows:

$$A = \{\mathsf{w}_\top(\mathsf{int}), \mathsf{r}_\bot(\mathsf{int})\} \quad \text{and} \quad C = \{\mathsf{w}_\bot(A), \mathsf{r}_\bot(A)\}$$

By one reduction step, P reduces to the process $[\overline{a}\langle 1\rangle]_\bot$, and the latter reduces to err as a result of a low-level process attempting to write on the high-level channel a. While the violation shows up after one reduction step, it originates earlier, from the fact that the value a of "high" level type A is written to channel $c : C$ with "low" write capability. Upgrading the write capability on C does not.

Consider then defining the types A and C differently, giving C high-level write capability:

$$A = \{\mathsf{w}_\top(\mathsf{int}), \mathsf{r}_\bot(\mathsf{int})\} \quad \text{and} \quad C = \{\mathsf{w}_\top(A), \mathsf{r}_\bot(A)\}$$

Again, the reduction of P to $[\overline{a}\langle 1\rangle]_\bot$ causes a security violation (i.e. a reduction to err) because the low-level process $[\overline{a}\langle 1\rangle]_\bot$ does not have the right to write on the channel a for which the write capability is "high". Here the problem originates from the high value a being written to a channel $c : C$ with "low" read capability.

The examples give a flavor of the inherent complexity of statically enforcing a security policy. Most of this complexity is determined by "indirect" flow of information arising as a result of processes dynamically acquiring new capabilities. In our case, the intuitive and direct measures represented by the "no read-up, no write-up" slogan are not enough to guarantee the desired effects of the access control policy. Further constraints must be imposed to prevent unauthorized access: the purpose of the type system we discuss next is to provide provably sufficient conditions for absence of security violations during reduction.

5.5 Types and Subtypes

The formal definition of types is somewhat complex, as it includes well-formedness rules ensuring that types are formed according to certain consistency conditions that provide the desired security guarantees. We start defining sets of *pre-capabilities* and *pre-types*.

Pre-Capabilities

$$\begin{array}{rcll} cap & ::= & \mathsf{r}_\sigma(T) & \sigma\text{-level input channel carrying values of type } T \\ & \mid & \mathsf{w}_\sigma(T) & \sigma\text{-level output channel carrying values of type } T \end{array}$$

Pre-Types

$$\begin{array}{rcll} S,T & ::= & \mathsf{B}_\sigma & \text{base type of level } \sigma \\ & \mid & \{cap_1,\ldots,cap_k\} & \text{channel type, } k \geqslant 0 \\ & \mid & (T_1,\ldots,T_k) & \text{tuple type, } k \geqslant 0 \end{array}$$

Next, we introduce the consistency conditions that single out the legal set of types. The consistency conditions are formulated in terms of ordering relations over pre-capabilities and pre-types induced by the ordering on security levels. Both the subtype relations are denoted by the symbol $\leqslant$, and are the least reflexive and transitive relations that are closed under the rules below.

Table 15 Subtyping

$$\text{(SUB OUTPUT)}\ \frac{T \leqslant S \quad \sigma \preceq \rho}{\mathsf{w}_\sigma(S) \leqslant \mathsf{w}_\rho(T)} \qquad \text{(SUB INPUT)}\ \frac{S \leqslant T \quad \sigma \preceq \rho}{\mathsf{r}_\sigma(S) \leqslant \mathsf{r}_\rho(T)}$$

$$\text{(SUB BASE)}\ \frac{\sigma \preceq \rho}{\mathsf{B}_\sigma \leqslant \mathsf{B}_\rho} \qquad \text{(SUB TYPE)}\ \frac{(\forall j \in J)(\exists i \in I) cap_i \leqslant cap'_j}{\{cap_i\}_{i\in I} \leqslant \{cap'_j\}_{j\in J}} \qquad \text{(SUB TUPLE)}\ \frac{S_i \leqslant T_i \quad i \in [1..k]}{(S_1,\ldots,S_k) \leqslant (T_1,\ldots,T_k)}$$

The two relations are mutually recursive, following the mutually inductive definition of pre-types and pre-capabilities. The rules (SUB INPUT) and (SUB OUTPUT) are the direct generalization of the corresponding rules in Table 4. The remaining rules define the subtype relation over basic, channel and tuple pre-types, respectively. Note that the resulting subtype relation on pre-types generalizes the subtype relation by Pierce and Sangiorgi we discussed in Section 2.

Now the set of types (as opposed to the previously introduced pre-types) is defined by a kinding system that identifies the legal pre-types at each security level. Formally, for each level ρ, the set $Type_\rho$ is the least set closed under the following rules:

Table 16 Type Formation

$$\text{(T-BASE)}\ \frac{\sigma \preceq \rho}{\mathsf{B}_\sigma \in Type_\rho} \qquad \text{(T-TUPLE)}\ \frac{T_i \in Type_\rho}{(T_1,\ldots,T_k) \in Type_\rho} \qquad \text{(T-RD)}\ \frac{T \in Type_\sigma \quad \sigma \preceq \rho}{\{\mathsf{r}_\sigma(T)\} \in Type_\rho}$$

$$\text{(T-WR)}\ \frac{T \in Type_\sigma \quad \sigma \preceq \rho}{\{\mathsf{w}_\sigma(T)\} \in Type_\rho} \qquad \text{(T-WRRD)}\ \frac{S \in Type_\sigma \quad T \in Type_{\sigma'} \quad \sigma,\sigma' \preceq \rho \quad S \leqslant T}{\{\mathsf{w}_\sigma(S), \mathsf{r}_{\sigma'}(T)\} \in Type_\rho}$$

There are a number of interesting consequences of the definition that are worth pointing out. First note that if $\sigma \preceq \rho$ then $RType_\sigma \subseteq RType_\rho$. This follows by a straightforward inductive reasoning on the generation of types at each kind. The second thing to note is the compatibility requirements between the read and write capabilities in the assumptions of the rule (T-WRRD). The condition $\sigma, \sigma' \preceq \rho$ contributes to the property that $RType_\sigma \subseteq RType_\rho$ for every $\sigma \preceq \rho$. The assumption $S \leqslant T$, in turn, is a standard condition required for soundness of channel communication: any value that is written on a channel can be read from that channel at a super-type of the value's true type. Interestingly, however, the combination of this condition with the security constraints imposed by the (T-WRRD) and the other rules has also security implications.

To see them, we first state the following proposition, which can be proved by induction on the derivation of $S \in Type_\sigma$.

Proposition 3. *If $S \in Type_\sigma$, and $S \leqslant T$, then there exists ρ with $T \in Type_\rho$ and $\sigma \preceq \rho$.*

We illustrate the security implication we just mentioned with and example. Consider the type $T = \{\mathsf{w}_\top(S), \mathsf{r}_\bot(S')\}$, and a channel $a : T$. A priory, high-level processes (with clearance $\top$) may write to this channel, while low-level processes, (with clearance $\bot$) may read from it. But then, it would seem, the channel may be used to leak sensitive information, for low-level processes may read values written by high-level processes. In particular, a high-level process could write on a the name of a high-level channel: low processes could then read that name and gain access to the channel, thus resulting in a violation of the security policy induced by our instrumented semantics.

A closer look at the type formation and subtyping rules shows that this cannot happen. To see why, assume that the type T is legal, that is $T \in Type_\rho$ for some security level ρ. The hypotheses of the (T-WRRD) rule imply that $\top \preceq \rho$, hence $\rho = \top$; furthermore, the two types S and S' must be such that $S \in Type_\sigma$ with $\sigma \preceq \top$, and $S' \in Type_{\sigma'}$ with $\sigma' \preceq \bot$ and $S \leqslant S'$. From these conditions, by the above proposition, it follows that $\sigma \preceq \sigma'$, and this, together with $\sigma' \preceq \bot$, implies that $\sigma = \bot$. In other words, the formation rules require that $S \in Type_\bot$, which implies that only low values (and channels) can be written to any channel of type T. But then, even though high-level processes can write on channels of type T, they may only write low-level values: thus only low-level information may flow from high to low processes.

In their present form, the type formation rules limit types to contain at most one read and one write capabilities: this clearly results in a loss of expressive power, but there is no fundamental difficulty in extending the formalization to handle types in the general form.

5.6 Typing Rules

The typing rules, given in Table 17, derive judgments in two forms: the usual form $\Gamma \vdash M : T$ stating that term M has type T, and the form $\Gamma \vdash^\sigma P$ which says that process P is well-typed in the context Γ, at security level σ (the rules for parallel composition and replication are standard, and omitted).

Table 17 Typing Rules for the Security π-calculus

$$\text{(NAME)}\ \frac{\Gamma(u) \leqslant A}{\Gamma \vdash u : A} \qquad \text{(RESTR)}\ \frac{\Gamma, a : T \vdash^{\sigma} P \quad T \in \mathit{Type}_{\sigma} \quad a \notin \mathsf{Dom}(\Gamma)}{\Gamma \vdash^{\sigma} (\nu a : T)P} \qquad \text{(PROC)}\ \frac{\Gamma \vdash^{\sigma \sqcap \rho} P}{\Gamma \vdash^{\sigma} [P]_{\rho}}$$

$$\text{(INPUT)}\ \frac{\Gamma, \tilde{x} : \tilde{T} \vdash^{\sigma} P \quad \Gamma \vdash u : \mathsf{r}_{\sigma}(\tilde{T}) \quad \tilde{x} \cap \mathsf{Dom}(\Gamma) = \varnothing}{\Gamma \vdash^{\sigma} u(\tilde{x} : \tilde{T}).P} \qquad \text{(OUTPUT)}\ \frac{\Gamma \vdash u : \mathsf{w}_{\sigma}(T) \quad \Gamma \vdash M : T}{\Gamma \vdash^{\sigma} \overline{u}\langle M \rangle}$$

The first three rules should be self-explanatory, but note, in the (RESTR) rule, that only names at level (at most) σ may legally be introduced by well-typed processes running at clearance σ. In the (INPUT) rule, the premises guarantee that the channel is used consistently with its associated capabilities and security level. For the latter, note that u offers a read capability at the same level σ at which the input process is currently running. From the definition of subtyping, and the rule (NAME), it follows that $\Gamma(u) = T$ for a type T that includes a read capability at level $\rho \preceq \sigma$: this guarantees that a process with clearance σ may read from any channel with security level up-to σ, as desired. The same reasoning applies to the (OUPUT) rule. The constraints imposed by the typing rules, together with the constraints imposed on the type formation rules provide static guarantees of type safety, that is absence of run-time violations for every well-typed process. Type safety is formalized as follows.

Theorem 8 (Type Safety for Resource Access). $\Gamma \vdash^{\sigma} P$ *implies* $[P]_{\sigma} \overset{\Gamma}{\not\longrightarrow} \mathsf{err}$

In other words, if a process P is well-typed at clearance σ, then neither P nor any of its derivatives will attempt a non-authorized access to a value or a channel restricted from level σ. That P is free of error reductions follows directly from the above theorem: that it is also true of the derivatives of P follows by the fact that well-typedness at any clearance level is preserved by reduction as stated by the following theorem.

Theorem 9 (Subject Reduction). *If* $\Gamma \vdash^{\sigma} P$ *and* $P \longrightarrow Q$*, then* $\Gamma \vdash^{\sigma} Q$

To exemplify the impact of the type system in enforcing our policy of resource access control, consider the process

$$P \triangleq (\nu a : A)(\nu c : C)[\overline{c}\langle a \rangle]_{\top} \mid [c(x).\overline{x}\langle 1 \rangle]_{\bot}.$$

In Example 3 we discussed two definitions for the types A and C: in both cases, the process is ill-typed independently of the clearance ($\top$ or $\bot$) at which we may type it. In fact, ill-typedness is a consequence of the type C being ill-formed, as $A \in \mathit{Type}_{\top}$ may not be read from channels of type C with $\bot$-level read capability.

We give more examples illustrating the rôle of types for security in the next section, where we discuss a variation of the type system that provides guarantees for the "no read-up, no write-down" constraints distinctive of the *defense* policy of MAC security.

5.7 MAC Policies: *Defense* Security

Few changes are required to type formation rules to account for this case of MAC security: the typing rules, instead, are unchanged. To understand and motivate the changes, we start with a simple examples.

Example 4 (Defense Security). Consider again the process P from example 3:

$$P \triangleq [\overline{c}\langle l\rangle]_\top \mid [c(x).\overline{x}\langle 1\rangle]_\bot.$$

where $\Gamma(c) = C$ and $\Gamma(l) = L$, and the two types C and L are defined as follows.

$$C = \{\mathsf{w}_\bot(L), \mathsf{r}_\bot(L)\} \quad \text{and} \quad L = \{\mathsf{w}_\bot(\mathsf{int}), \mathsf{r}_\bot(\mathsf{int})\}.$$

With the current type system, P is well typed in Γ, as the channel c has "low" type and offers read and write capabilities: hence both processes may legally access c. The same is true of the type assignment $l : L$, and $c : C$ with L as above, and C defined now as $C = \{\mathsf{w}_\top(L), \mathsf{r}_\bot(L)\}$. Indeed, it is not difficult to see that there is no violation of our resource access policy, as there is no P error reduction for P or any of its derivatives.

In both the above cases, the term P would be rejected as "unsafe" under defense security, as in both cases a high-level process ends-up writing a low-level object, hence establishing a high-to-low flow of information. It is, however, easy to identify the source of the problems, and change the type system to enforce the new constraints.

In the first case, the problem is a direct violation of the "no write-down" constraint, which results from the current definition of subtyping. The judgment $\Gamma \vdash^\top \overline{c}\langle l\rangle$ is derivable by an application of the (OUTPUT) from the premise $\Gamma \vdash c : \mathsf{w}_\top(L)$, as $\Gamma(c) \leqslant \mathsf{w}_\top(L)$. In particular, the subtype relation holds because so does $\mathsf{w}_\bot(L) \leqslant \mathsf{w}_\top(L)$: to prevent the write-down, it is thus enough to rule out the latter relation.

In the second case, instead, the problem results from the channel c offering a write capability to processes running at high clearance, and read capability to low processes. As a result, a low process can "read up" information written by high-level processes on the same channel. To prevent such situations, it is enough to refine the type formation rules by requiring that a read capability on a channel type not be lower than the write capability (if any).

The new set of types may thus be defined as follows:

Definition 2 (Types for defense security). *For any security level* ρ*, let* $Type_\rho$ *be the least set of types that is closed under the subtype and kind rules of Section 5.5, where*

- *rule (*SUB OUTPUT*) is replaced by:* $\dfrac{T \leqslant S}{\mathsf{w}_\sigma(S) \leqslant \mathsf{w}_\sigma(T)}$
- *rule (*T-WRRD*) is replaced by:* $\dfrac{S \in Type_\sigma \quad T \in Type_{\sigma'} \quad \sigma \preceq \sigma' \preceq \rho \quad S \leqslant T}{\{\mathsf{w}_\sigma(S), \mathsf{r}_{\sigma'}(T)\} \in Type_\rho}$

Given the new definition of types, and the typing rules of Table 17, it is possible to show that well-typed processes do not cause any defense security violation. Of course, this requires a new definition of error reductions, to reflect the desired notion of violation under defense security.

5.8 Information Flow Security

Having outlined a solution to defense security, we conclude our discussion on the security π-calculus with a few observations on information-flow security.

As we already mentioned, information flow security aims at protecting confidentiality and integrity of information by preventing ‘implicit’ flow of information via covert channels. Examples of covert channels may naturally formalized in the security π-calculus.

As a first example, consider the system:

$$[h(x).\mathsf{if}\ x = 0\ \mathsf{then}\ \overline{hl}\langle 0\rangle\ \mathsf{else}\ \overline{hl}\langle 1\rangle]_{\top} \mid [hl(z).Q]_{\bot}$$

where one has $hl : HL$ and $h : H$, and the two types in question are defined as follows:

$$HL = \{\mathsf{w}_{\top}(\mathsf{int}), \mathsf{r}_{\bot}(\mathsf{int})\}, \quad H = \{\mathsf{w}_{\top}(\mathsf{int}), \mathsf{r}_{\top}(\mathsf{int})\}.$$

We have already noticed the presence of information flow in a similar process in Example 4, resulting from a low process reading on a channel that is written by a high process. Here the case of information flow is more interesting, however, as the low process gains additional information on the value x transmitted over the high-channel h. Indeed, the example is not problematic, as the definition of types for defense security rules out this system as insecure. Consider however, the new system:

$$[h(x)\,\mathsf{if}\ x = 0\ \mathsf{then}\ [\bar{l}\langle 0\rangle]_{\bot}\ \mathsf{else}\ [\bar{l}\langle 1\rangle]_{\bot}]_{\top} \mid [l(z).Q]_{\bot}$$

where now $h : H$, $l : L$ and the two types are defined as follows:

$$H = \{\mathsf{w}_{\top}(\mathsf{int}), \mathsf{r}_{\top}(\mathsf{int})\}, \quad L = \{\mathsf{w}_{\bot}(\mathsf{int}), \mathsf{r}_{\bot}(\mathsf{int})\}$$

This system is well-typed, even with the type system of Section 5.7, as the high-level process downgrades itself prior to writing on the low-level channel l. Still, the system exhibits the same high-to-low flow of information as before.

As a final example, it is instructive to look at the impact of synchronous communication over information flow. Assuming synchronous communication the following has the same problems as the previous one. Consider

$$[\overline{l_1}\langle\rangle.Q_1 \mid \overline{l_2}\langle\rangle.Q_2]_{\bot} \quad \mid \quad [\mathsf{if}\ x = 0\ \mathsf{then}\ l_1()\ \mathsf{else}\ l_2()]_{\top}$$

Assuming $L = \{\mathsf{w}_{\bot}(), \mathsf{r}_{\bot}()\}$, and $l_1, l_2 : L$, the system is well-typed, and yet there is an implicit flow of information arising purely from synchronization: information on the value of x may be assumed by both the continuations Q_1 and Q_2 of the low process.

5.9 Further Reading

The work on information-flow security for the π-calculus is well developed in [34] and subsequent work by Hennessy[2]. A related approach is discussed by Honda and Vasconcelos in [35].

[2] (see http://www.cogs.susx.ac.uk/users/matthewh/research.html).

Information flow analysis based on non-interference originated with the seminal idea of Goguen and Meseguer [32]. In process calculi, the first formalizations of non-interference were proposed in [51, 24, 52], based on suitable trace-based notions of behavioral process equivalence for CCS-like calculi.

Information flow analyses based on typing techniques were first discussed in the pioneering work D. and P. Dennings [19], in which a type system detecting direct and indirect flows among program variables in imperative languages was devised. This initial idea was refined and formalized some twenty years later in work on type systems providing guarantees of non-interference in multi-threaded programming languages both in nondeterministic [56, 55] and probabilistic settings [53].

Type systems for secure information flow and non-interference in process have also been applied to enforce secrecy of cryptographic protocols. The most notable applications of typing techniques for analysis of security protocols have been developed for Abadi and Gordon's *spi* calculus [5, 10], that we discuss in the next section.

6 The CryptoSPA Calculus

In this section we report from [27] the *Cryptographic Security Process Algebra* (CryptoSPA for short). It is basically a variant of value-passing CCS [41], where the processes are provided with some primitives for manipulating messages. In particular, processes can perform message encryption and decryption, and also construct complex messages by composing together simpler ones.

6.1 Syntax of the Calculus

CryptoSPA syntax is based on the following elements:

- A set $I = \{a, b, \ldots\}$ of *input* channels, a set $O = \{\bar{a}, \bar{b}, \ldots\}$ of *output* ones;
- A set M of basic messages and a set K of encryption keys with a function $\cdot^{-1} : K \to K$ such that $(k^{-1})^{-1} = k$. The set $\mathcal{M}$ of all messages is defined as the least set such that $M \cup K \in \mathcal{M}$ and $\forall m \in \mathcal{M}$, $\forall k \in K$ we have that (m, m') and $\{m\}_k$ also belong to $\mathcal{M}$;
- A set C of *public* channels; these channels represent the insecure network where the enemy can intercept and fake messages;
- A family $\mathcal{U}$ of sets of messages and a function $Msg(c) : I \cup O \longrightarrow \mathcal{U}$ which maps every channel c into the set of possible messages that can be sent and received along such a channel. Msg is such that $Msg(c) = Msg(\bar{c})$.
- A set $Act = \{c(m) \mid c \in I, m \in Msg(c)\} \cup \{\bar{c}\langle m\rangle \mid c \in O, m \in Msg(c)\} \cup \{\tau\}$ of actions (τ is the internal, invisible action), ranged over by a; we also have a function $chan(a)$ which returns c if a is either $c(m)$ or $\bar{c}\langle m\rangle$, and the special channel *void* when $a = \tau$; we assume that *void* is never used within a restriction operator (see below).
- A set *Const* of constants, ranged over by A.

The syntax of CryptoSPA agents is defined as follows:

$$E ::= \mathbf{0} \bigm| c(x).E \bigm| \bar{c}\langle e\rangle.E \bigm| \tau.E \bigm| E+E \bigm| E \| E \bigm| E \backslash L \bigm| E[f] \bigm|$$
$$\bigm| A(m_1, \ldots, m_n) \bigm| [e = e']E; E \bigm| [\langle e_1 \ldots e_r\rangle \vdash_{rule} x]E; E$$

where x is a variable, $m_1,\ldots,m_n$ are messages, $e,e_1,\ldots,e_r$ are messages (possibly containing variables) and L is a set of input channels. Both the operators $c(x).E$ and $[\langle e_1 \ldots e_r\rangle \vdash_{rule} x]E;E'$ bind the variable x in E. It is also necessary to define constants as follows: $A(x_1,\ldots,x_n) \triangleq E$ where E is a CryptoSPA agent which may contain no free variables except $x_1,\ldots,x_n$, which must be distinct.

Besides the standard value-passing CCS operators, we have an additional one that has been introduced in order to model message handling and cryptography. Informally, the $[\langle m_1 \ldots m_r\rangle \vdash_{rule} x]E_1;E_2$ process tries to deduce an information z from the tuple of messages $\langle m_1 \ldots m_r\rangle$ through one application of rule $\vdash_{rule}$; if it succeeds then it behaves like $E_1[z/x]$, otherwise it behaves like E_2; for example, given a rule $\vdash_{dec}$ for decryption, process $[\langle \{m\}_k, k^{-1}\rangle \vdash_{dec} x]E_1;E_2$ decrypts message $\{m\}_k$ through key k^{-1} and behaves like $E_1[m/x]$ while $[\langle \{m\}_k, k'\rangle \vdash_{dec} x]E_1;E_2$ (with $k' \neq k^{-1}$) tries to decrypt the same message with the wrong inverse key k' and (since it is not permitted by $\vdash_{dec}$) it behaves like E_2.

We call $\mathcal{E}$ the set of all the CryptoSPA terms, and we define $sort(E)$ to be the set of all the channels syntactically occurring in the term E.

6.2 The Operational Semantics of CryptoSPA

In order to model message handling and cryptography, in Table 18 we define an inference system which formalizes the way messages may be manipulated by processes.

Table 18 Inference System for message manipulation

Let $m, m' \in \mathcal{M}$ and $k, k^{-1} \in K$.

$$\frac{\text{m \& m'}}{\text{(m,m')}}\ (\vdash_{pair}) \qquad \frac{\text{(m,m')}}{\text{m}}\ (\vdash_{fst}) \qquad \frac{\text{(m,m')}}{\text{m'}}\ (\vdash_{snd})$$

$$\frac{\text{m \& k}}{\{m\}_k}\ (\vdash_{enc}) \qquad \frac{\{m\}_k \ \& \ k^{-1}}{\text{m}}\ (\vdash_{dec})$$

It is indeed quite similar to those used by many authors (see, e.g., [38, 39]). In particular it can combine two messages obtaining a pair (rule $\vdash_{pair}$); it can extract one message from a pair (rules $\vdash_{fst}$ and $\vdash_{snd}$); it can encrypt a message m with a key k obtaining $\{m\}_k$ and finally decrypt a message of the form $\{m\}_k$ only if it has the corresponding (inverse) key k^{-1} (rules $\vdash_{enc}$ and $\vdash_{dec}$). We denote with $\mathcal{D}(\phi)$ the set of messages that can be deduced by applying the inference rules on the messages in ϕ. Note that we are assuming encryption as completely reliable. Indeed we do not allow any kind of cryptographic attack, e.g., the guessing of secret keys. This permits to observe the attacks that can be carried out even if cryptography is completely reliable.

The formal behavior of a CryptoSPA term is described by means of the *labelled transition system* $< \mathcal{E}, Act, \{\xrightarrow{a}\}_{a \in A} >$, where $\xrightarrow{a}_{a \in A}$ is the least relation between CryptoSPA terms induced by axioms and inference rules of Table 19.

Table 19 Operational semantics

$$(input)\frac{m \in Msg(c)}{c(x).E \xrightarrow{c(m)} E[m/x]} \qquad (output)\frac{m \in Msg(c)}{\overline{c}\langle m\rangle.E \xrightarrow{\overline{c}\langle m\rangle} E} \qquad (internal)\frac{}{\tau.E \xrightarrow{\tau} E}$$

$$(\|_1)\frac{E \xrightarrow{a} E'}{E \,\|\, E_1 \xrightarrow{a} E' \,\|\, E_1} \qquad (\|_2)\frac{E \xrightarrow{c(m)} E' \quad E_1 \xrightarrow{\overline{c}\langle m\rangle} E_1'}{E \,\|\, E_1 \xrightarrow{\tau} E' \,\|\, E_1'}$$

$$(+_1)\frac{E \xrightarrow{a} E'}{E + E_1 \xrightarrow{a} E'} \qquad ([f])\frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]} \qquad (\backslash L)\frac{E \xrightarrow{a} E' \quad chan(a) \notin L}{E\backslash L \xrightarrow{a} E'\backslash L}$$

$$(=_1)\frac{m \neq m' \quad E_2 \xrightarrow{a} E_2'}{[m = m']E_1; E_2 \xrightarrow{a} E_2'} \qquad (=_2)\frac{m = m' \quad E_1 \xrightarrow{a} E_1'}{[m = m']E_1; E_2 \xrightarrow{a} E_1'}$$

$$(def)\frac{E[m_1/x_1, \dots, m_n/x_n] \xrightarrow{a} E' \quad A(x_1, \dots, x_n) \triangleq E}{A(m_1, \dots, m_n) \xrightarrow{a} E'}$$

$$(\mathcal{D}_1)\frac{\langle m_1 \dots m_r\rangle \vdash_{rule} m \quad E_1[m/x] \xrightarrow{a} E_1'}{[\langle m_1 \dots m_r\rangle \vdash_{rule} x]E_1; E_2 \xrightarrow{a} E_1'} \qquad (\mathcal{D}_2)\frac{\nexists m : \langle m_1 \dots m_r\rangle \vdash_{rule} m \quad E_2 \xrightarrow{a} E_2'}{[\langle m_1 \dots m_r\rangle \vdash_{rule} x]E_1; E_2 \xrightarrow{a} E_2'}$$

Plus symmetric rules for $+_1$, $\|_1$ and $\|_2$are omitted

Example. We present a very simple example of a protocol where A sends a message m_A to B encrypted with a key k_{AB} shared between A and B[3]. We define it as $P \triangleq A(m_A, k_{AB}) \,\|\, B(k_{AB})$ where $A(m,k) \triangleq \overline{c}\langle\{m\}_k\rangle$[4] and $B(k) \triangleq c(y).[\langle y,k\rangle \vdash_{dec} z]\overline{out}\langle z\rangle$. Moreover, $k_{AB}^{-1} = k_{AB}$ (symmetric encryption) and $Msg(c) = \{\{m\}_k \mid m \in M, k \in K\}$. We want to analyze the execution of P with no intrusions, we thus consider $P \setminus \{c\}$, since the restriction guarantees that c can be used only inside P. We obtain a system which can only execute action $\overline{out}\langle m_A\rangle$ that represents the correct transmission of m_A from A to B. In particular, the only possible execution is the one where A sends to B message $\{m_A\}_{k_{AB}}$ and then $\overline{out}\langle m_A\rangle$ is executed:

$$P \setminus \{c\} \xrightarrow{\tau} (\mathbf{0} \,\|\, [\langle\{m_A\}_{k_{AB}}, k_{AB}\rangle \vdash_{dec} z]\overline{out}\langle z\rangle) \setminus \{c\} \xrightarrow{\overline{out}\langle m_A\rangle} (\mathbf{0} \,\|\, \mathbf{0}) \setminus \{c\}$$

The calculus of CryptoSPA has been successfully applied to the automatic specification and the verification of security protocols, see [20,26,27,21,25,23,48–50,22].

[3] For the sake of readability, we omit the termination $\mathbf{0}$ at the end of every agent specifications, e.g., we write a in place of $a.\mathbf{0}$. We also write $[m = m']E$ in place of $[m = m']E; \mathbf{0}$ and analogously for $[\langle m_1 \dots m_r\rangle \vdash_{rule} x]E; \mathbf{0}$.

[4] Note that this process could be also written as $A(m,k) \triangleq [\langle m,k\rangle \vdash_{enc} x]\overline{c}\langle x\rangle$.

7 The Spi-Calculus

The spi calculus is an extension of the pi calculus with cryptographic primitives that has been introduced by Abadi and Gordon in [5, 10], The spi calculus is designed for describing and analyzing security protocols, such as those for authentication and for electronic commerce. These protocols rely on cryptography and on communication channels with properties like authenticity and privacy. Accordingly, cryptographic operations and communication through channels, are the main ingredients of the spi calculus.

As we discussed in Section 1, some abstract security protocol can be expressed in the pi calculus, thanks to its simple but powerful primitives for channels. Moreover, the scoping rules of the pi calculus guarantee that the environment of a protocol (the attacker) cannot access a channel that is not explicitly given; scoping is thus the basis of security. However, as we pointed out, when considering a distributed environment, it is not realistic to rely only on the scope rules, we also have to prevent the context from having free access to public channels over which private names are communicated. In a distributed environment such a channel protection relies on the use of cryptography. With shared-key cryptography, secrecy can be achieved by communication on public channels under secret keys.

The spi calculus is thus an extension of the pi calculus that consider cryptographic issues in more detail. Its features can be summarized as follows:

- it permits an explicit representation of the use of cryptography in protocols, while it does not seem easy to represent encryption and decryption within the pi calculus;
- it relies on the powerful scoping constructs of the pi calculus;
- within the spi calculus, the environment can be defined as an arbitrary spi calculus process instead of giving an explicit model;
- security properties, both integrity and secrecy, can be represented as equivalences and analyzed by means of static techniques.

7.1 Syntax and Semantics

The syntax of the spi calculus extends a particular version of the pi calculus with constructs for encrypting and decrypting messages (see Table 20) . In standard pi calculus names are the only terms. For convenience, the syntax of spi calculus also contains constructs for paring and numbers, namely (M,N),0 and $succ(M)$. Furthermore, the term $\{M_1,\dots,M_k\}_N$ represents the ciphertext obtained by encrypting $M_1,\dots,M_k$ under the key N using a shared-key cryptosystem such as DES. The key is an arbitrary term; typically, names are used as keys because in the spi calculus names are unguessable capabilities.

Intuitively, the new constructs of spi calculus have the following meanings: a match $[M\ is\ N]P$ behaves as P provided that terms M and N are the same, otherwise it is stuck. A pair splitting process $let\ (x,y) = M\ in\ P$, where x and y are bound in P, behaves as $P\{x := N, y := L\}$ if the term M is the pair (N,L). An integer case process $case\ M\ of\ 0 : P\ succ(x) : Q$, where x is bound in Q, behaves as P if term M is 0, as $Q\{x := N\}$ if M is $succ(N)$. Finally the process $case\ L\ of\ \{x_1,\dots,x_k\}_N\ in\ P$, where $x_1,\dots,x_k$ are bound

Table 20 Spi calculus syntax

Expressions L,M,N	$::=$	bv	basic value
	$\mid$	$a,\dots,p$	name
	$\mid$	$x,\dots,z$	variable
	$\mid$	(M,N)	pair
	$\mid$	0	zero
	$\mid$	$succ(M)$	successor
	$\mid$	$\{M_1,\dots,M_k\}_N$	shared-key encryption ($k \geq 0$)
Processes P,Q,R	$::=$	$\mathbf{0}$	stop
	$\mid$	$\overline{u}\langle N_1,\dots,N_k\rangle.P$	output ($k \geq 0$)
	$\mid$	$u(x_1,\dots,x_k).P$	input ($k \geq 0$)
	$\mid$	$(\nu a)P$	restriction
	$\mid$	$P \mid P$	composition
	$\mid$	$!P$	replication
	$\mid$	$[M\ is\ N]P$	match
	$\mid$	$let\ (x,y) = M\ in\ P$	pair splitting
	$\mid$	$case\ M\ of\ 0 : P\ succ(x) : Q$	integer case
	$\mid$	$case\ L\ of\ \{x_1,\dots,x_k\}_N\ in\ P$	shared-key decryption ($k \geq 0$)

in P, attempts to decrypt the term L with the key N; if L is a ciphertext of the form $\{M_1,\dots,M_k\}_N$, then the process behaves as $P\{x_1 := M_1,\dots,x_k := M_k\}$, and otherwise the process is stuck.

Implicit in the definition of the spi calculus syntax are some standard but significant assumptions about cryptography: (*i*) the only way to decrypt an encrypted packet is to know the corresponding key; (*ii*) an encrypted packet does not reveal the key that was used to encrypt it; (*iii*) there is sufficient redundancy in messages so that the decryption algorithm can detect whether a ciphertext was encrypted with the expected key.

Operational Semantics. The operational semantics of spi calculus can be defined in terms of a structural congruence and a reduction relation, extending the corresponding relations defined in Section 1 for the π calculus. In particular, structural congruence is defined as the least congruence relation closed under rules 1.-4. of Section 1.1 plus the following rules:

$$\begin{array}{lll}
\text{(Red Repl)} & !P & \equiv P \mid !P \\
\text{(Red Match)} & [M\ is\ M]P & \equiv P \\
\text{(Red Let)} & let\ (x,y) = (M,N)\ in\ P & \equiv P\{x := M, y := N\} \\
\text{(Red Zero)} & case\ 0\ of\ 0 : P\ succ(x) : Q & \equiv P \\
\text{(Red Succ)} & case\ succ(M)\ of\ 0 : P\ succ(x) : Q & \equiv Q\{x := M\} \\
\text{(Red Decrypt)} & case\ \{\tilde{M}\}_N\ of\ \{\tilde{x}\}_N\ in\ P & \equiv P\{\tilde{x} := \tilde{M}\}
\end{array}$$

The reduction relation is then the least relation closed under the following rules: In order to develop proof techniques for the spi calculus, we define an auxiliary, equivalent, operational semantics based on a commitment relation, in the style of Milner [44]. The

Table 21 Reduction Relation

$$(\textsc{Comm})\quad n(x_1,\ldots,x_k).P \mid \overline{n}\langle M_1,\ldots,M_k\rangle.Q \longrightarrow P\{x_1 := M_1,\ldots,x_k := M_k\} \mid Q$$

$$(\textsc{Struct})\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\qquad \frac{P \longrightarrow P'}{(\nu n)P \longrightarrow (\nu n)P'}\qquad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

definition of commitment depends on two new syntactic forms: *abstractions* and *concretions*. An abstraction is a term of the form $(\tilde{x})P$, where $x_1,\ldots,x_k$ are bound variables, and P is a process. A concretion is a term of the form $(\nu\tilde{m})\langle\tilde{M}\rangle P$ where $M_1,\ldots,M_k$ are expressions, P is a process, and the names $m_1,\ldots m_l$ are bound in $M_1,\ldots,M_k$ and P. Finally an *agent* is an abstraction, a process or a concretion. We use the metavariables A and B to stand for arbitrary agents, C for concretions, and F for abstractions. Restriction and parallel composition for abstractions and concretions are defined as follows:

$$\begin{array}{lll}
(\nu m)(\tilde{x})P & = (\tilde{x})(\nu m)P & \\
Q \mid (\tilde{x})P & = (\tilde{x})(Q \mid P) & \text{with } \{\tilde{x}\} \cap fv(Q) = \emptyset \\
(\nu m)(\nu\tilde{n})\langle\tilde{M}\rangle P & = (\nu m,\tilde{n})\langle\tilde{M}\rangle P & \text{with } m \notin \{\tilde{n}\} \\
Q \mid (\nu\tilde{n})\langle\tilde{M}\rangle P & = (\nu\tilde{n})\langle\tilde{M}\rangle Q \mid P & \text{with } \{\tilde{n}\} \cap fn(Q) = \emptyset
\end{array}$$

If F is the abstraction $(x_1,\ldots,x_k)P$ and C is the concretion $(\nu n_1,\ldots,n_l)\langle M_1,\ldots,M_k\rangle Q$, and if $\{n_1,\ldots,n_l\} \cap fn(P) = \emptyset$, we define the process $F@C$ and $C@F$ as follows:

$$\begin{array}{l}
F@C \triangleq (\nu n_1)\ldots(\nu n_l)(P\{x_1 := M_1,\ldots,x_k := M_k\} \mid Q) \\
C@F \triangleq (\nu n_1)\ldots(\nu n_l)(Q \mid P\{x_1 := M_1,\ldots,x_k := M_k\})
\end{array}$$

Let the *reduction relation* $>$ be the least relation on closed processes that satisfies the following axioms:

$$\begin{array}{lll}
\text{(Red Repl)} & !P & > P \mid !P \\
\text{(Red Match)} & [M \textit{ is } M]P & > P \\
\text{(Red Let)} & \textit{let } (x,y) = (M,N) \textit{ in } P & > P\{x := M, y := N\} \\
\text{(Red Zero)} & \textit{case } 0 \textit{ of } 0 : P \textit{ succ}(x) : Q & > P \\
\text{(Red Succ)} & \textit{case succ}(M) \textit{ of } 0 : P \textit{ succ}(x) : Q & > Q\{x := M\} \\
\text{(Red Decrypt)} & \textit{case } \{\tilde{M}\}_N \textit{ of } \{\tilde{x}\}_N \textit{ in } P & > P\{\tilde{x} := \tilde{M}\}
\end{array}$$

A *barb* β is a name m (representing input) or a co-name $\overline{m}$ (representing output). An *action* is a barb or a distinguished *silent action* τ. The commitment relation is written $P \xrightarrow{\alpha} A$ where P is a closed process, α is an action and A is a closed agent. The commitment relation is defined by rules in Table 22.

The following proposition asserts that the two operational semantics for spi calculus, the one based on reduction relation, and the other one based on commitment relation, are equivalent.

Proposition 4. *$P \longrightarrow Q$ if and only if $P \xrightarrow{\tau} \equiv Q$.*

Table 22 Commitment Relation

(COMM OUT)
$$\frac{}{\overline{m}\langle\tilde{M}\rangle.P \xrightarrow{\overline{m}} (\nu)\langle\tilde{M}\rangle P}$$

(COMM IN)
$$\frac{}{m(\tilde{x}).P \xrightarrow{m} (\tilde{x})P}$$

(COMM INTER 1)
$$\frac{P \xrightarrow{m} F \quad Q \xrightarrow{\overline{m}} C}{P \mid Q \xrightarrow{\tau} F@C}$$

(COMM INTER 2)
$$\frac{P \xrightarrow{\overline{m}} C \quad Q \xrightarrow{m} F}{P \mid Q \xrightarrow{\tau} C@F}$$

(COMM PAR 1)
$$\frac{P \xrightarrow{\alpha} A}{P \mid Q \xrightarrow{\alpha} A \mid Q}$$

(COMM PAR 2)
$$\frac{Q \xrightarrow{\alpha} A}{P \mid Q \xrightarrow{\alpha} P \mid A}$$

(COMM RES)
$$\frac{P \xrightarrow{\alpha} A \quad \alpha \notin \{m, \overline{m}\}}{(\nu m)P \xrightarrow{\alpha} (\nu m)A}$$

(COMM RED)
$$\frac{P > Q \quad Q \xrightarrow{\alpha} A}{P \xrightarrow{\alpha} A}$$

Testing Equivalence. Testing equivalence is useful to compare process behaviors and to define security properties such as secrecy and authentication.

Let a *test* be a pair (Q, β) consisting of a closed process Q and a barb β. We say that P *passes* a test (Q, β) if and only if

$$(P \mid Q) \xrightarrow{\tau} Q_1 \ldots \xrightarrow{\tau} Q_n \xrightarrow{\beta} A$$

for some $n \geq 0$, some processes $Q_1, \ldots, Q_n$ and some agent A. We obtain a testing preorder $\sqsubseteq$ and a testing equivalence $\simeq$ on closed processes:

$$P \sqsubseteq P' \triangleq \text{for any test } (Q,\beta)\text{, if } P \text{ passes } (Q,\beta) \text{ then } P' \text{ passes } (Q,\beta)$$
$$P \simeq P' \triangleq P \sqsubseteq P' \text{ and } P' \sqsubseteq P$$

The idea of testing equivalence comes from the work of De Nicola and Hennessy [18]. A test neatly formalizes the idea of a generic experiment or observation that another process (such as an attacker) might perform on a process. Thus testing equivalence concisely captures the concept of equivalence in an arbitrary environment. Furthermore, testing equivalence is a congruence; more precisely, if $P \simeq Q$ then P and Q may be used interchangeably in any context, that is $\mathcal{C}[P] \simeq \mathcal{C}[Q]$ for any closed context $\mathcal{C}$.

7.2 Secrecy by Typing in the Spi Calculus

In this section we describe rules that Abadi proposed in [1] for achieving secrecy properties in security protocols expressed in the spi calculus. The rules have the form of typing rules; they guarantee that, if a protocol typechecks, then it does not leak its secret inputs. Before starting the formalization of the type system, we recall from [1] some informal security principle we adopt in the following.

First, our rules should constrain only the principals that want to protect their secrets form the attacker. That is since in some situations we may assume that the attacker cannot guess certain keys, but we cannot expect to restrict the code that the attacker runs.

We then consider only three classes of data: *Public* data, which can be communicated to anyone, *Secret* data, which should not be leaked, *Any* data, that is, arbitrary data. We refer to *Secret, Public* and *Any* as levels or types. We then assume that

> The result of encrypting data with a public key has the same classification as the data, while the result of encrypting data with a secret key may be made public.
>
> Only public data can be sent on public channels, while all kinds of data may be sent on secret channels.

Because a piece of data of level *Any* could be of level *Secret*, it should not be leaked. On the other hand, a piece of data of level *Any* could be of level *Public*, so it cannot be used as a secret. Thus

> if all we know about a piece of data is that it has level *Any*, then we should protect it as if it had level *Secret*, but we can exploit it only if it had level *Public.*

In our rules we adopt a standard format for all messages on secret channels or under secret keys. Each message on a secret channel has three components, the first of which has level *Secret*, the second *Any*, and the third *Public*, plus a confounder component. This schema implements the following principle:

> Upon receipt of a message, it should be easy to decide which part of the contents are sensitive information, if any. This decision is least error-prone when it does not depend on implicit context.

For the use of confounders, note that if each encrypted message of a protocol includes a freshly generated confounder in a standard position, then the protocol will not generate the same ciphertext more than once.

Types and Typing Rules. The syntax of types corresponds to the three classes of data:

$$\mathit{Types} \quad S,T \quad ::= \quad \mathit{Public} \mid \mathit{Secret} \mid \mathit{Any}$$

There is also a subtyping relation between types: $T <: S$ holds if T equals S or if S is *Any.* The typing system contains three forms of judgments: $\vdash E$ stating that the environment E is well formed, $E \vdash M : T$ stating that the term M is of level T in E, and $E \vdash P$ stating that the process P typechecks in E.

An environment is a list of distinct names and variables with associated levels. In addition, each name n has an associated term of the form $\{M_1,\ldots,M_k,n\}_N$. This association means that the name n may be used as a confounder only in the term $\{M_1,\ldots,M_k,n\}_N$. We write $x : T$ for variable x with level T, and $n : T :: \{M_1,\ldots, M_k,n\}_N$. The rules for environments are in Table 23.

The hypotheses of rule (ENV NAME) imply that if a variable x occurs in $\{M_1,\ldots, M_k,n\}_N$, then it is declared in E. This means that we cannot instantiate the variable x in several ways, obtaining several different terms with the same confounder, and thus defeating the purpose of confounders.

Table 23 Environment Formation

$$\text{(ENV } \emptyset\text{)}\quad \frac{}{\vdash \emptyset} \qquad \text{(ENV VAR)}\quad \frac{\vdash E \quad x \notin dom(E)}{\vdash E, x : T} \qquad \text{(ENV NAME)}\quad \frac{\vdash E \quad n \notin dom(E) \quad E \vdash M_i : T_i \;\; i = 1..k \quad E \vdash N : S}{\vdash E, n : T :: \{M_1, \ldots, M_k, n\}_N}$$

Table 24 Typing Rules for Terms

$$\text{(SUBSUM)}\quad \frac{E \vdash M : T \quad T <: S}{E \vdash M : S} \qquad \text{(VARIABLE)}\quad \frac{\vdash E \quad x : T \in E}{E \vdash x : T} \qquad \text{(NAME)}\quad \frac{\vdash E \quad n : T :: \{M_1, \ldots, M_k, n\}_N \text{ in } E}{E \vdash n : T}$$

$$\text{(ZERO)}\quad \frac{\vdash E}{E \vdash 0 : Public} \qquad \text{(SUCC)}\quad \frac{E \vdash M : T}{E \vdash succ(M) : T} \qquad \text{(PAIR)}\quad \frac{E \vdash M : T \quad E \vdash N : T}{E \vdash (M, N) : T}$$

$$\text{(ENCRYPT } \mathit{Secret}\text{)}\quad \frac{E \vdash M_1 : Secret \;\; E \vdash M_2 : Any \quad E \vdash M_3 : Public \qquad E \vdash N : Secret \quad n : T :: \{M_1, M_2, M_3, n\}_N \text{ in } E}{E \vdash \{M_1, M_2, M_3, n\}_N : Public}$$

$$\text{(ENCRYPT } \mathit{Public}\text{) } \mathit{with}\ T = \mathit{Public}\ \mathit{if}\ k = 0\quad \frac{E \vdash M_i : T \;\; i = 1..k \quad E \vdash N : Public}{E \vdash \{M_1, \ldots, M_k, n\}_N : T}$$

Rules (ZERO) and (SUCC) say that 0 is of level *Public* and that adding one preserves the level of a piece of data. Therefore, these classifications mean that the typing system works even against an attacker that may generate any number, starting from 0 and successively incrementing it. The rule (ENCRYPT *Public*) says that k pieces of data of the same level T can be encrypted under a key of level *Public*, with a resulting ciphertext of level T. The rule (ENCRYPT *Secret*) imposes more restrictions for encryption under keys of level *Secret*, because the resulting ciphertext is of level *Public*. These restrictions enforce a particular format for the contents and the use of a confounder: the ciphertext must contain a first component of level *Secret*, a second one of level *Any*, a third one of level *Public*, and an appropriate confounder as final component. Note that there is no rule for encryption for the case where N is a term of level *Any*.

Finally, typing rules for processes are collected in Table 25.

The first four rules handle input and output processes. Rule (OUTPUT *Public*) says that terms of level *Public* may be sent on a channel of level *Public*. Rule (OUTPUT *Secret*) says that terms of all levels may be sent on a channel of level *Secret*, provided this is done according to the correct format of a secret message. The two rules for input match these rules for output. Note that if M is a term of level *Any* and it is not known whether it is of level *Public* or *Secret*, then M cannot be used as a channel. The rule (PAIR SPLIT) breaks a term of level *Public* or *Secret* into two components, each assumed to be of the same level of the original term. The case where the origi-

Table 25 Typing rules for processes

(OUTPUT *Public*)
$$\frac{E \vdash M : \mathit{Public} \quad E \vdash M_i : \mathit{Public} \;\; i = 1..k \quad E \vdash P}{E \vdash \overline{M}\langle M_1, \ldots, M_k \rangle.P}$$

(DEAD)
$$\frac{\vdash E}{E \vdash \mathbf{0}}$$

(PAR)
$$\frac{E \vdash P \quad E \vdash Q}{E \vdash P \mid Q}$$

(OUTPUT *Secret*)
$$\frac{\begin{array}{l} E \vdash M : \mathit{Secret} \quad E \vdash P \\ E \vdash M_1 : \mathit{Secret} \; E \vdash M_2 : \mathit{Any} \; E \vdash M_3 : \mathit{Public} \end{array}}{E \vdash \overline{M}\langle M_1, M_2, M_3 \rangle.P}$$

(REPL)
$$\frac{E \vdash P}{E \vdash !P}$$

(NEW)
$$\frac{E, n : T :: L \vdash P}{E \vdash (\nu n)P}$$

(INPUT *Secret*)
$$\frac{E \vdash M : \mathit{Secret} \quad E, x_1 : \mathit{Secret}, x_2 : \mathit{Any}, x_3 : \mathit{Public} \vdash P}{E \vdash M(x_1, x_2, x_3).P}$$

(INPUT *Public*)
$$\frac{E \vdash M : \mathit{Public} \quad E, x_i : \mathit{Public} \vdash P \quad i = 1..k}{E \vdash M(x_1, \ldots, x_k).P}$$

(PAIR SPLIT) $T \in \{\mathit{Public}, \mathit{Secret}\}$
$$\frac{E \vdash M : T \quad E, x : T, y : T \vdash P}{E \vdash \mathit{let}\ (x, y) = M\ \mathit{in}\ P}$$

(INTEGER) $T \in \{\mathit{Public}, \mathit{Secret}\}$
$$\frac{E \vdash M : T \quad E \vdash P \quad E, x : T \vdash Q}{E \vdash \mathit{case}\ M\ \mathit{of}\ 0 : P\ \mathit{succ}(x) : Q}$$

(MATCH) $T, S \in \{\mathit{Public}, \mathit{Secret}\}$
$$\frac{E \vdash M : T \quad E \vdash N : S \quad E \vdash P}{E \vdash [M\ \mathit{is}\ N]P}$$

(DECRYPT *Public*) $T \in \{\mathit{Public}, \mathit{Secret}\}$
$$\frac{E \vdash L : T \quad E \vdash N : \mathit{Public} \quad E, x_i : T \vdash P \;\; i = 1..k}{E \vdash \mathit{case}\ L\ \mathit{of}\ \{x_1, \ldots, x_k\}_N\ \mathit{in}\ P}$$

(DECRYPT *Secret*) $T \in \{\mathit{Public}, \mathit{Secret}\}$
$$\frac{E \vdash L : T \quad E \vdash N : \mathit{Secret} \quad E, x_1 : \mathit{Secret}, x_2 : \mathit{Any}, x_3 : \mathit{Public}, x_4 : \mathit{Any} \vdash P}{E \vdash \mathit{case}\ L\ \mathit{of}\ \{x_1, x_2, x_3, x_4\}_N\ \mathit{in}\ P}$$

nal term is known only to be of level *Any* is disallowed; if it were allowed, this rule would permit leaking whether the term is in fact a pair. The same holds true for rules (MATCH),(INTEGER) and (DECRYPT). Rule (DECRYPT *Secret*) gives the level *Any* to the confounder in the message being decrypted, for lack of more accurate static information but with no significant loss. Finally, note that there is no rule for decryption with a key of level *Any*.

Properties of the Type System. The main property of the previous type system is that if a process *P* typechecks, then it does not leak the values of parameters of level *Any*.

The secrecy property of well typed processes is formalized in the following theorem, where the notion of leaking is expressed via testing equivalence.

Theorem 10 (Secrecy). *If only variables of level Any and only names of level Public are in the domain of the environment E, if σ and σ' are two substitutions of values for the variables in E, and if P typechecks, i.e. $E \vdash P$, then $P\sigma$ and $P\sigma'$ are testing equivalent, i.e. $P\sigma \simeq P\sigma'$.*

The conclusion of theorem 10 means that an observer cannot distinguish $P\sigma$ and $P\sigma'$, so it cannot detect the difference in the values for the variables. Despite their secrecy, none of these variables is declared with level *Secret* ; however, the process P may produce terms of level *Secret* during its execution using the restriction operator (e.g. it may construct fresh encryption keys). For instance, P may be the process $(\nu K)(\nu m)(\nu n)\overline{c}\langle\{m,x,0,n\}_K\rangle$ where x is of level *Any* and c is of level *Public*, and where we can assign the type *Secret* to the bound names K,m,n. Theorem 10 implies that P does not leak the value x, in the sense that $P\{x := M\}$ and $P\{x := N\}$ are testing equivalent for all closed terms M and N. Thus, the typing system is meant to protect parameters of level *Any* relying on dynamically generated names of level *Secret.*

7.3 An Example with Key Establishment

We argued that the spi calculus enables more detailed descriptions of security protocols than the pi calculus. While the pi calculus enables the representation of channels, the spi calculus also enables the representation of channel implementations in terms of cryptography.

As in the pi calculus, scoping is the basis of security in spi calculus. In particular, restriction can be used to model the creation of fresh, unguessable cryptographic keys. Restriction can also be used to model the creation of fresh nonces of the sort used in challenge-response exchanges.

In this section we refine the example shown in Section 1, where we presented an abstract and simplified version of the Wide Mouthed Frog protocol. The following example is the cryptographic version of that of Section 1. In this protocol, the principals A and B share keys K_{AS} and K_{SB} respectively with a server S. When A and B want to communicate securely, A creates a new key K_{AB}, sends it to the server under K_{AS}, and the server forwards it to B under K_{SB}. Since all communication is protected by encryption, communication can take place through public channels, which we write c_{AS}, c_{SB} and c_{AB} as in Section 1. In addition to the keys and the payload M, the protocol messages include the names of principals and confounders. Informally, a simplified version of this protocol is:

Message 1: $A \rightarrow S\ \ \{K_{AB},*,(A,B),C_A\}_{K_{AS}}$ on c_{AS}

Message 2: $S \rightarrow B\ \ \{K_{AB},*,(A,B),C_S\}_{K_{SB}}$ on c_{SB}

Message 3: $A \rightarrow B\ \ \{*,M,*,C'_A\}_{K_{AB}}$ on c_{AB}

The channels c_{AS}, c_{BS}, c_{AB} are public. The keys K_{AS}, K_{SB} are secret keys for communication with the server, while K_{AB} is the new secret key for communication from A to B. C_A, C'_A, C_S are confounders, and $*$ is an arbitrary message of appropriate level. In Message 1, A provides the key K_{AB} to S, which passes it on to B in Message 2. In Message 1 and Message 2, the pair (A,B) conveys the names of the users of the key. In Message 3, A uses K_{AB} for sending M.

In the spi calculus, we can express this message sequence as follows, where we assume that B, after receiving the message M from A, outputs an arbitrary message on a public cannel d:

$$A(M) \triangleq (\nu K_{AB})(\nu C_A)\overline{c_{AS}}\langle\{K_{AB},*,(a,b),C_A\}_{K_{AS}}\rangle.(\nu C'_A)\overline{c_{AB}}\langle\{*,M,*,C'_A\}_{K_{AB}}\rangle$$

$$S \triangleq c_{AS}(x).case\ x\ of\ \{x_{key},x_1,x_2,x_{cnf}\}_{K_{AS}}\ in\ (\nu C_S)\overline{c_{SB}}\langle\{x_{key},x_1,x_2,C_S\}_{K_{SB}}\rangle$$

$$\begin{aligned} B \triangleq\ & c_{BS}(x).case\ x\ of\{x_{key},x_1,x_2,y_{cnf}\}_{K_{SB}}\ in \\ & c_{AB}(z).case\ z\ of\{z_1,z_{cipher},z_2,z_{cfn}\}_{x_{key}}\ in\ \overline{d}\langle*\rangle\} \end{aligned}$$

$$Inst(M) \triangleq (\nu K_{AS})(\nu K_{SB})(A(M)\mid S\mid B)$$

Now, assuming that M is a term of type *Any*, and c_{AS},c_{BS},c_{AB},d are channels of type *Public*, it is easy to prove that the process $Inst(M)$ is well typed. As a consequence of the theorem 10, we have that the protocol above does not reveal the message M from A. In particular, we have $Inst(M') \simeq Inst(M'')$ for arbitrary terms M',M''.

Notice that also in this version of the Wide Mouthed Frog protocol, the use of scope extrusion is essential: A generates the key K_{AB} and sends it out of scope to B via S. In the example discussed so far, channel establishment and data communication happen only once. More sophisticated examples may be written to represent many protocol sessions between many principals. However, as the intricacy of the examples increases, so does the opportunity for errors. Note that many of the mistakes in authentication protocols arise from confusion between sessions. See [6] for further examples.

7.4 Secrecy Types for Asymmetric Communication

Although so far we have discussed only shared-key cryptography, other kinds of cryptography are also easy to treat within the spi calculus. Many security protocols use asymmetric communication primitives, namely communication channels with only one fixed end-point (the receiver) and particularly public-key encryption. Compared to shared-key encryption, these primitives present special difficulties, partly because they rely on pairs of related capabilities (e.g. “public” and “private” keys) with different level of secrecy and scopes.

In this section, we show a variant of spi calculus that focus on asymmetric communication primitives, especially public-key encryption. This process calculus has been proposed by Abadi and Blanchet in [3], where authors also show a type system in which types convey secrecy properties and such that well typed programs keep their secrets.

We consider a polyadic, asynchronous, variant of spi calculus that includes channels with only one fixed end-point (the receiver) and public-key encryption. Channels with fixed receivers can be used for transmitting secrets if the adversary cannot listen on those channels. On the other hand, the capability for sending on those channels may be published. Such channels may therefore convey not only secrets but also public data from the adversary. The type system will handle both cases.

In addition, in a public-key encryption scheme, the capabilities of encryption and decryption are separate, and can be handled separately. Typically, the capability for decryption (the “private” key) remains with one principal, while the capability for encryption (the “public” key) may be published. Our process calculus and type system treat public-key encryption and communication on channels with fixed receivers analogously.

Table 26 Syntax of the process calculus

Expressions L,M,N	$::=$	$a,\dots,p,k$	name
	$\mid$	$x,\dots,z$	variable
	$\mid$	$\{M_1,\dots,M_k\}_N$	encryption ($k \geq 0$)
Processes P,Q,R	$::=$	$\mathbf{0}$	stop
	$\mid$	$\overline{M}\langle N_1,\dots,N_k\rangle$	output ($k \geq 0$)
	$\mid$	$a(x_1,\dots,x_k).P$	input ($k \geq 0$)
	$\mid$	$(\nu a)P$	restriction
	$\mid$	$P \mid P$	composition
	$\mid$	$!P$	replication
	$\mid$	$case\ M\ of\ \{x_1,\dots,x_n\}_k : P\ else\ Q$	decryption ($n \geq 0$)
	$\mid$	$if\ M = N\ thenP\ else\ Q$	conditional

The syntax of the process calculus is shown in Table 26. In order to deal with asymmetric communication, Abadi and Blanchet in [3] propose to follow the same approach of the local pi calculus [40].

In the local pi calculus, input is possible only on channels that are syntactically represented by names (and not variables). Output is possible on channels represented by names or variables. Thus, the input capability for a channel a remains within the scope of the restriction $(\nu a)P$ where a is created, while the output capability can be transmitted outside. Further, this approach is extended to public-key encryption, as follows. Decryption is possible only with keys that are syntactically represented by names (and not variables). Encryption is possible with keys that are represented by names or variables. Thus we have a model where the encryption capability may be public while the decryption capability remains private, in the scope where it is generated.

Thus, when a name a refers to a channel, it represents both end-points of the channels, that is the capabilities for output and input on the channel. A variable can confer only the former capability, even if its run-time value is a. Similarly, a name k will not represent a single encryption or decryption key, but rather the pair of an encryption key and the corresponding decryption key. A variable can confer only the capability of encrypting, even if its value is k at run-time.

As an example, consider the following process:

$$(\nu k)(\overline{a}\langle k\rangle \mid b(x).case\ x\ of\ \{y\}_k : \overline{c}\langle y\rangle)$$

This process relies on three public channels, a,b,c. It generates a fresh key pair k; outputs the corresponding encryption key on a; and receives messages on b, filtering for one encrypted under k, of which it outputs the plaintext on c.

The operational semantics of the calculus can be defined in a standard way using a reduction relation and a structural congruence relation, see [3] for details.

Secrecy by Typing. In the following we show a type system such that well typed processes are proven to keep their secrets. In particular, we use a concept of secrecy similar to that we discussed for the spi calculus and in Section 4 for the pi calculus. We say that a process preserves the secrecy of a piece of data M if the process never publishes M, or anything that would permit the computation of M, even in interaction with an attacker. Moreover, we think of an attacker as any process Q of the calculus, represented by the sets of its initial capabilities (i.e. the set of names on which it is able to output, input, encrypt, and decrypt).

The types of our type system are defined by the following grammar:

$$\begin{aligned}\text{Types} ::= \mathit{Public} \mid \mathit{Secret} \ &\mid C^{\mathit{Public}}[T_1,\ldots,T_n] \mid C^{\mathit{Secret}}[T_1,\ldots,T_n] \\ &\mid K^{\mathit{Public}}[T_1,\ldots,T_n] \mid K^{\mathit{Secret}}[T_1,\ldots,T_n]\end{aligned}$$

Let L range over $\{\mathit{Public},\mathit{Secret}\}$, we will write $C^L[T_1,\ldots,T_n]$. We have a subtyping relation that is the least reflexive relation such that $C^L[T_1,\ldots,T_n] \leq L$ and $K^L[T_1,\ldots,T_n] \leq L$. Note that we do not have $\mathit{Secret} \leq \mathit{Public}$ or vice versa.

Public (resp. *Secret*) is the type of public (resp. secret) data. $C^{\mathit{Secret}}[T_1,\ldots,T_n]$ is the type of a channel on which the opponent cannot send messages, and which carries n-tuples with components of types T_i. Similarly, $K^{\mathit{Secret}}[T_1,\ldots,T_n]$ is the type of an encryption key that the adversary does not have, and which is used to encrypt n-tuples with components of types T_i. $C^{\mathit{Public}}[T_1,\ldots,T_n]$ is the type of a channel on which the opponent may send messages. The channel may be intended to carry n-tuples with components of types T_i, but the adversary may send *any* data it has (that is, any public data) along that channel. Similarly, $K^{\mathit{Public}}[T_1,\ldots,T_n]$ is the type of an encryption key that the opponent may have. This key is intended for encrypting n-tuples with components of types T_i, but the adversary may encrypt *any* data it has (that is, any public data) under this key.

We do not show the typing rules for this process calculus (see [3]), we only discuss the rationale of the type system.

- Any public data can be sent on a channel of type $C^{\mathit{Public}}[T_1,\ldots,T_n]$ or *Public*. This use of the channel may not seem to conform to its declared type. However, it is unavoidable, since we expect that an attacker can use the channel; moreover, it does not cause harm from the point of view of secrecy.
- Since channels of type $C^{\mathit{Secret}}[T_1,\ldots,T_n]$ may not be known by an attacker, we can guarantee that only tuples with types $T_1,\ldots,T_n$ can be sent on such a channel.
- When typing the process $a(x_1,..,x_n).P$ where a is a channel of type $C^{\mathit{Public}}[T_1,..,T_n]$, two cases arise. In the first case input values are of type *Public*; in the second case input values have the expected types $T_1,\ldots,T_n$. In order to typecheck the process $a(x_1,\ldots,x_n).P$, the type system thus checks that the process P executed after the input is well typed in both cases.
- When reading from a channel a of type $C^{\mathit{Secret}}[T_1,\ldots,T_n]$, the input values must be of the expected types $T_1,\ldots,T_n$ since the channel a cannot be known to the attacker.

- Rules for encryption are similar to those for output. Any public data can be encrypted under a public encryption key, and data of types $T_1, \ldots, T_n$ can be encrypted under a key of type $K^L[T_1, \ldots, T_n]$. Dually, rules for decryption are similar to those for input.
- Ciphertexts are always of type *Public*.

This type system reflects a binary view of secrecy, according to which the world is divided into system and attacker, and a secret is something that the attacker does not have. When we wish to express that a piece of data is a secret for a given set of principals, we define the system to include only the processes that represent those principals. Note that the mechanism of group creation we discussed in Section 4, directly supports a rich view of secrecy that does not simply divide the world in two parts. Even if that approach does not treat cryptography, we think that the type system with group creation can be extended to deal also with cryptographic primitives.

Properties of the Type System. We start with a lemma that says that every process is well-typed, at least in a fairly trivial way that makes its free names public. This lemma is important because it means that any process that represents an opponent is well-typed. It is a formal counterpart to the informal idea that the type system cannot constrain the adversary.

Lemma 2. *Let P be an untyped process. If $fn(P) \subseteq \{a_1, \ldots, a_n\}$,$fv(P) \subseteq \{x_1, \ldots, x_m\}$, and $T_i \leq Public$ for all $i = 1 \ldots m$, then $a_1 : Public, \ldots, a_n : Public, x_1 : T_1, \ldots, x_m : T_m \vdash P$.*

We end with an informal statement of the secrecy theorem, see [3] for a complete formalization.

Theorem 11 (Secrecy). *Let P be a well-typed, closed process. Then P preserves the secrecy of names of type Secret against adversaries that can input, output, encrypt, and decrypt on names declared Public, and output and encrypt on names declared $C^{Public}[\ldots]$ and $K^{Public}[\ldots]$.*

As an example, we can obtain $a{:}Public, s{:}Secret \vdash (\nu k)\overline{a}\langle\{s\}_k, k\rangle$ by letting $k : K^{Public}$ $[Secret]$. Then the theorem above implies that the process $(\nu k)\overline{a}\langle\{s\}_k, k\rangle$ preserves the secrecy of s form any opponent that can input, output, encrypt, and decrypt on a. In other words, if Q is a closed process and $fn(Q) \subseteq \{a\}$, then $Q \mid (\nu k)\overline{a}\langle\{s\}_k, k\rangle$ does not output s on a. Thus, assuming that Q does not have s in advance, Q cannot guess s or compute it from the message on a.

7.5 Further Reading

In [6], a final section shows how we could add to the syntax of pure spi calculus cryptographic operations such as hashing, public-key encryption and digital signature.

A more general approach is that of [4], where authors introduce and study the so called *applied pi calculus*, a uniform extension of the pi calculus that is parameterized on a finite set of function symbols. Such functions can be instantiated as data structures (e.g. pairs) but also as cryptographic functions as hashing, (a)symmetric encryption, probabilistic encryption, message authentication codes (MACs). The main advantage of applied pi calculus is that its semantics and proof techniques represent a common framework to reason about very different security protocols.

Beside secrecy, other security properties can be studied in the context of spi calculus. As an example, see [6] for a formalization of authenticity property with testing equivalence.

Finally, in [8, 7, 9] authors study the security properties of the join calculus (a variant of pi calculus with an emphasis on distributed programming [28]) enriched with cryptography.

References

1. M. Abadi. Secrecy by typing security protocols. *Journal of the ACM*, 46(5):749–786, 1999.
2. M. Abadi. Security protocols and specifications. In *Proceedings of FOSSACS'96*, pages 1–13. Springer-Verlag, LNCS 1578, 1999.
3. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Proceedings of FOSSACS'01*, pages 25–41. Springer-Verlag, 2001.
4. M. Abadi and C. Fournet. Mobile values, new names and secure communication. In *Proceedings of POPL'01*, pages 104–115. ACM Press, 2001.
5. M. Abadi and A. D. Gordon. A calculus of cryptographic protocols: the spi calculus. In *Fourth ACM Conference on Computer and Communication Security*, pages 36–47, 1997.
6. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: the spi calculus. Technical Report 149, Digital Equipment Corporation Systems Reasearch Center, January 1998.
7. Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.
8. Martín Abadi, Cédric Fournet, and Georges Gonthier. A top-down look at a secure message. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Proceedings of FSTTCS '99*, volume 1738 of *LNCS*, pages 122–141. Springer, December 1999.
9. Martín Abadi, Cédric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Proceedings of POPL '00*, pages 302–315. ACM, January 2000.
10. Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Journal of Information and Computation*, 148(1):1–70, 1999.
11. M. Boreale, C. Fournet, and C. Laneve. Bisimulations for the join-calculus. In David Gries and Willem-Paul de Roever, editors, *Proceedings of PROCOMET '98*. IFIP, 1998.
12. Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.
13. Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In Catuscia Palamidessi, editor, *Proceedings of CONCUR 2000*, volume 1877 of *LNCS*, pages 365–379. Springer, August 2000.
14. Luc Maranget Cédric Fournet, Cosimo Laneve and Didier Rḿy. Implicit typing à la ML for the join-calculus. In *Proc. of the 1997 8th International Conference on Concurrency Theory*. Springer, 1997.
15. Sylvain Conchon. *Analyse modulaire de flot d'information pour les calculs sq́uentiels et concurrents*. PhD thesis, École Polytechnique, 2002.
16. Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA'99)*, Palm Springs, California, 1999.
17. Sylvain Conchon and François Pottier. JOIN(X): Constraint-based type inference for the join-calculus. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 221–236. Springer Verlag, April 2001.

18. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
19. D.E. Denning and P.J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20:504–513, 1977.
20. A. Durante, R. Focardi, and R. Gorrieri. A compiler for analysing cryptographic protocols using non-interference. *ACM Transactions on Software Engineering and Methodology*, 9(4):489–530, 2000.
21. A. Durante, R. Focardi, and R. Gorrieri. CVS: A compiler for the analysis of cryptographic protocols. In *Proceedings of CSFW'99*, pages 203–212. IEEE press, 1999.
22. A. Durante, R. Focardi, and R. Gorrieri. CVS at Work: A report on new failures upon some cryptographic protocol. In *Proceedings of MMM-ACNS'01*. Springer LNCS 2052, 2001.
23. R. Focardi, A. Ghelli, and R. Gorrieri. Using non interference for the analysis of security protocols. In *Proceedings of DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.
24. R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.
25. R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, September 1997.
26. R. Focardi, R. Gorrieri, and F. Martinelli. "Non Interference for the Analysis of Cryptographic Protocols". In *Proceedings of ICALP'00*, number 1853 in LNCS, pages 744–755, July 2000.
27. R. Focardi and F. Martinelli. A uniform approach for the definition of security properties. In *Proceedings of World Congress on Formal Methods (FM'99)*, pages 794–813. Springer, LNCS 1708, 1999.
28. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer.
29. Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.
30. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer-Verlag. LNCS 1119.
31. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer-Verlag. LNCS 1119.
32. J.A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of Symposium on Secrecy and Privacy*, pages 11–20. IEEE Computer Society, April 1982.
33. M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous π-calculus (extended abstract). In *Proceedings of ICALP2000*, volume 1853 of *LNCS*, pages 415–427. Springer-Verlag, 2000.
34. Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. Computer Science Technical Report 2000:03, School of Cognitive and Computing Sciences, University of Sussex, 2000.
35. K. Honda, V.T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In Gert Smolka, editor, *Proceedings of ESOP 2000*, volume 1782 of *LNCS*, pages 180–199. Springer, 2000.
36. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P[ièerre] America, editor, *Proceedings of ECOOP '91*, volume 512 of *LNCS*, pages 133–147. Springer, July 1991.

37. A. Igarashi and Kobayashi N. A generic type system for the π-calculus. In *Proceedings of POPL '01*, pages 128–141. ACM Press, 2000.
38. G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *Proceedings of Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 146–166, Passau (Germany), March 1996. Springer-Verlag, LNCS 1055.
39. W. Marrero, E. Clarke, and S. Jha. A model checker for authentication protocols. In *Proc. of DIMACS Workshop on Design and Formal Verification of Security Protocols*. Rutgers University, Sep. 1997.
40. Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of ICALP '98*, volume 1443 of *LNCS*, pages 856–867. Springer, July 1998.
41. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
42. R. Milner. Sorts in the π-calculus (extended abstract). In E[ike] Best and G[rzegorz] Rozenberg, editors, *Proceedings of the 3rd Workshop on Concurrency and Compositionality*, volume 191 of *GMD-Studien*. GMD Bonn, St. Augustin, 1991. Also available as Report 6/91 from University of Hildesheim.
43. R. Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.
44. R. Milner. The π-calculus. Undergraduate lecture notes, Cambridge University, 1995.
45. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.
46. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.
47. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Compoter Sciences*, 6(5):409–454, 1996. An extended abstract in *Proc. LICS 93*, IEEE Computer Society Press.
48. R. Focardi and F. Martinelli. A uniform approach for the analysis of cryptographic protocols. In *Conference on Security in Communication Networks (SCN'99), (G. Persiano ed.)*, 1999.
49. R. Focardi, R. Gorrieri, and F. Martinelli. Secrecy in security protocols as non interference. In *Proceedings of DERA/RHUL Workshop on Secure Architectures and Information Flow (S. Schneider and P. Ryan ed.)*. Elsevier ENTCS, Volume 32, 1999.
50. R. Focardi, R. Gorrieri, and F. Martinelli. Message authentication through non interference. In *Proceedings of International Conference on Algebraic Methodology And Software Technology (AMAST 2000)*, pages 258–272. Springer LNCS 1816, 2000.
51. A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through determinism. In *Proceeding of European Symposium on Research in Computer Security 1994 (ESORICS'94)*, number 875 in LNCS, pages 33–53. Springer-Verlag, 1994.
52. P.Y.A. Ryan and S.A. Schneider. Process algebra and non-interference. In *Proceedings of 12th IEEE Computer Security Foundation Workshop*, pages 214–227. IEEE press, 1999.
53. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundation Workshop*, Cambridge (UK), 2000. IEEE press.
54. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
55. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of POPL'98*, pages 355–364. ACM Press, January 1988.
56. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.