Secure Implementations of Typed Channel Abstractions

(Extended Abstract)

Michele Bugliesi
Dipartimento di Informatica
Università Ca' Foscari, Venezia, Italy
michele@dsi.unive.it

Marco Giunti
Dipartimento di Informatica
Università Ca' Foscari, Venezia, Italy
giunti@dsi.unive.it

Abstract

The challenges hidden in the implementation of high-level process calculi into low-level environments are well understood [3]. This paper develops a secure implementation of a typed pi calculus, in which capability types are employed to realize the policies for the access to communication channels. Our implementation compiles high-level processes of the pi-calculus into low-level principals of a cryptographic process calculus based on the applied-pi calculus [1]. In this translation, the high-level type capabilities are implemented as term capabilities protected by encryption keys only known to the intended receivers. As such, the implementation is effective even when the compiled, low-level principals are deployed in open contexts for which no assumption on trust and behavior may be made. Our technique and results draw on, and extend, previous work on secure implementation of channel abstractions in a dialect of the join calculus [2]. In particular, our translation preserves the forward secrecy of communications in a calculus that includes matching and supports the dynamic exchange of write *and* read access-rights among processes. We establish the adequacy and full abstraction of the implementation by contrasting the untyped equivalences of the low-level cryptographic calculus, with the typed equivalences of the high-level source calculus.

Categories and Subject Descriptors F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.3 [*Studies of Program Constructs*]: Type structure; D.1.3 [*Programming Techniques*]: Concurrent Programming; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

General Terms Security, Languages, Verification

Keywords Typed Behavioral Theories, Language Encodings, Full Abstraction

1. Introduction

The use of types for resource access control is a long established technique in the literature on process calculi [17], and so is the application of typed equivalences to reason on the behavior of typed processes [14, 18]. Resource control is achieved by predicating the access to communication channels on the possession of certain type capabilities, and by having a static typing system ensure that the resulting policies are complied with by well-typed processes. Typed equational techniques, in turn, draw on judgments of the form $\mathrm{I} \models P \cong Q$ stating the indistinguishability (hence the equivalence) of two processes, P and Q, in enclosing contexts that have access to the names in the type environment I via the type capabilities assigned to them by I.

Typed equations of this kind are very effective whenever we have control on the structure of the contexts observing our processes, i.e., whenever may assume that such contexts are well-typed. The question we address in this paper is whether the same kind of reasoning can still be relied upon when we extend the class of observing contexts to arbitrary, potentially ill-typed contexts. Stated more explicitly: can we deploy our typed processes as low-level agents to run in distributed environments, in a fully abstract manner, i.e. preserving the typed behavioral congruences we have established? This appears to be an important question, as it constitutes a fundamental prerequisite to the use of typed process calculi as an abstract specification tool for concurrent computations in distributed, open systems.

In [10] we argue that the desired correspondence may hardly be achieved for high-level process calculi that rely on static typing alone. The solution we envision in that paper is based on a new typing discipline that combines static and dynamic typing. Specifically, we introduce a typed variant of the pi-calculus in which the output construct, noted $a\langle v@T\rangle$, uses type coercion to enforce the delivery of v at the type T, regardless of the type of the communication channel a. A static typing system guarantees that v may indeed be assigned the coercion type T, while a mechanism of dynamically typed synchronization guarantees that v is received only at supertypes of T, so as to guarantee type soundness of each exchange.

By breaking the dependency between the types of the transmission channels and the types of the names transmitted, distinctive of the approaches to typing in the pi calculus tradition [17, 14], we can safely reduce the capability types to the simplest, flat structure that only exhibits the read/write access rights on channels, regardless of the types of the values transmitted. Furthermore, and more interestingly for our present concerns, the combination of type coercion and dynamically typed synchronization allows us to gain further control on the interactions among processes as well as between processes and their enclosing context. Based on that, we are able to recover fully abstract implementations of the high-level specifications, i.e. implementations that preserve the typed congruences established for the specifications. The basic idea is rather simple, and suggested by the very structure of the types. Briefly, we represent a channel with a pair of asymmetric keys, an encryption key to

POPL'07 January 17–19, 2007, Nice, France.

transmit and a decryption key to receive data on n — noted n^+ and n^- respectively. Then, we establish the following correspondence between the cryptographic keys and the type-level capabilities attached to each name: $[\![n@\mathsf{w}]\!] = n^+$ and $[\![n@\mathsf{r}]\!] = n^-$. Based on this representation, we may implement a communication over a channel as an exchange over the network, as follows[1]:

$$[\![n\langle m@A\rangle]\!] = net\langle\{[\![m@A]\!]\}_{n^+}\rangle$$
$$[\![n(x).P]\!] = net(y).\mathsf{decrypt}\ y\ \mathsf{as}\ \{x\}_{n^+}\ \mathsf{with}\ n^-\ \mathsf{in}\ [\![P]\!]$$

While this idea is appealing in its simplicity, it suffers from a number of shortcomings, first made explicit by Abadi in [3]. In subsequent work, Abadi, Fournet and Gonthier have shown how to counter these problems and recover a fully abstract implementation for the join calculus [2].

The fundamental obstacle against using the solution of [2] for our present purposes is related to the so called problem of *forward secrecy*. To illustrate, consider the following example which we adapt from [3]. Let P and Q be the two processes defined as follows (we omit the type coercions whenever irrelevant):

$$P = (\mathsf{new}\ n)(n\langle m\rangle \mid n(x).p\langle n@\mathsf{r}\rangle)$$
$$Q = (\mathsf{new}\ n)(n\langle m'\rangle \mid n(x).p\langle n@\mathsf{r}\rangle)$$

It is not difficult to be convinced that P and Q are behaviorally equivalent (essentially under any typing assumption), as m and m' are sent over a secret channel and no high-level context may recover the contents of messages sent. On the other hand, a low-level context may tell $[\![P]\!]$ from $[\![Q]\!]$ by buffering the message sent on n and then deciphering it when n^- is published. In [2], this problem is avoided altogether, as the join calculus does not allow names to be communicated with read capabilities, a feature that instead constitutes one of the fundamental ingredients of our typed calculus.

To recover forward secrecy, one needs a more structured representation of type capabilities to make sure that distributing a read capability does not correspond to leaking any decryption key. The solution we envision in the present paper is based on the representation of a channel as a process that serves input and output requests, so that each message exchange is the result of two separate protocols with writer and reader clients. All channels are associated with two separate key-pairs. The decryption keys are always stored securely at the channel, and never leaked; the encryption keys, in turn, are available to the clients that have read and/or write access to the channel. In the write protocol, the client sends data and the channel buffers it on a private queue; in the read protocol, the client sends a session key and the server returns data encrypted with that key. As a result, publishing a read (write) capability on a channel corresponds to publishing the read (write) encryption keys associated with the channel: decryption keys are never leaked, as desired.

Under appropriate, mostly standard, hypothesis on the properties of the underlying network, we show that a translation based on these ideas is sound. Full abstraction, instead, is harder to achieve as one needs to build safeguards against attacks that exploit malformed data or rely on malicious channels that intentionally leak their associated decryption keys. To account for that, we complement the translation with a proxy-service mechanism to ensure that all communication protocols take place via system generated (hence secure) channels. We prove that the resulting implementation is fully abstract by showing that the untyped equivalences of the low-level cryptographic calculus coincide, via the translation, with the typed equivalences of the high-level calculus. As a byproduct, since our high-level process calculus is a conservative extension of the untyped pi calculus, we also have a direct fully abstract implementation of the pi calculus. Finally, we devise a distributed implementation of the proxy service for a variant of our high-level process calculus in which processes are assigned to different domains, each one providing a local proxy. We extend the full abstraction result to this enhanced, more realistic model.

Plan of the paper. Section 2 reviews the typed pi calculus of [10], that we employ as the high-level language. Section 3 introduces the dialect of the applied-pi calculus that we use as the implementation language. Section 4 formalizes the low-level model of the network that underpins our implementation. Sections 5 to 7 detail the translations we have outlined and establishes the main results. Section 8 concludes with final remarks.

2. A pi calculus with dynamic typing

Given its intended use as a specification language for distributed systems, our pi-calculus is asynchronous. We remark, however, that the same technique and results would apply, mutatis mutandis, to the synchronous case.

We presuppose countable sets of names and variables, ranged over by $a - n$ and $x, y, \ldots$ respectively. We use u, v to range over names and variables, when the distinction does not matter. We denote tuples of values and types with $\tilde{u}$ and $\tilde{A}$, respectively. The syntax of processes is given below.

A, B	::=	$\mathsf{ch}(\mathsf{rw}) \mid \mathsf{ch}(\mathsf{r}) \mid \mathsf{ch}(\mathsf{w}) \mid \top$	
P, Q	::=	$\mathbf{0}$	*inaction*
		$P \mid Q$	*composition*
		$(\mathsf{new}\ n : A)P$	*restriction*
		$!\,P$	*replication*
		$[u = v]\,P; Q$	*matching*
		$u\langle\tilde{v}@\tilde{A}\rangle$	*type-coerced output*
		$u(\tilde{x}@\tilde{A}).P$	*typed input*

The first four constructs are standard from the pi-calculus. The matching process $[u = v]\,P; Q$ is a conditional that proceeds as P if $u = v$, and as Q otherwise. In the input and output forms, the notation $\tilde{v}@\tilde{A}$ (respectively $\tilde{x}@\tilde{A}$) is short for $v_1@A_1, \ldots, v_n@A_n$ (resp. $x_1@A_1, \ldots, x_n@A_n$). Restrictions and input prefixes bind names and variables, respectively. The notions of free and bound names/variables arise as expected.

The types $\mathsf{ch}(\cdot)$ are the types of channels, built around the capabilities r, w and rw which provide *read*, *write* and *full fledged* access to the channel, respectively. To ease the notation, we henceforth denote channel types by only mentioning the associated capabilities. $\top$ is the type of all values and may be used to export a name without providing any capability for synchronization or exchange.

The subtyping relation $<:$ is the lattice defined as follows (with A any type): $\mathsf{rw} <: \mathsf{r}$, $\mathsf{rw} <: \mathsf{w}$, $A <: \top$, with a meet operation arising as expected, namely:

$$\mathsf{rw} \sqcap \mathsf{r} \triangleq \mathsf{rw} \quad \mathsf{rw} \sqcap \mathsf{w} \triangleq \mathsf{rw} \quad \mathsf{w} \sqcap \mathsf{r} = \mathsf{rw} \quad A \sqcap \top \triangleq A$$

Type environments, ranged over by Γ, Δ, are finite mappings from names and variables, to types. Following [14], we extend type environments by using the meet operator: $\Gamma \sqcap u : A = \Gamma, u : A$ if $u \notin dom(\Gamma)$, otherwise $\Gamma \sqcap u : A = \Gamma'$ with Γ' differing from Γ only at u: $\Gamma'(u) = \Gamma(u) \sqcap A$. Subtyping is extended to type environments as expected: $\Gamma <: \Delta$ whenever $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Delta)$ and for all $a : \mathrm{dom}(\Gamma)$ we have $\Gamma(a) <: \Delta(a)$. If $\Gamma(n) <: \mathsf{r}$, we say that $\Gamma^{\mathsf{r}}(n)$ is defined, written $\Gamma^{\mathsf{r}}(n)\downarrow$. $\Gamma^{\mathsf{r}}(n)\uparrow$ indicates that $\Gamma(n)$ is undefined or $\Gamma(n) \not<: \mathsf{r}$. Corresponding notation is employed for the write capability.

[1] Here net is a public channel representing the network. Now, since all exchanges are assumed to occur over net, the input process should also be instrumented with a recovery mechanism to handle all those cases in which it picks up "wrong packets". We omit this and other details at this stage, for ease of readability.

Table 1 Typing and Dynamics for the pi calculus

Typing rules The typing rules for name and variable projection, and for the process forms of parallel composition, replication, restriction and nil are entirely standard cf. [17, 19]). The remaining rules are given below.

(T-Match)
$$\frac{\Gamma \vdash Q \quad \Gamma(u) = A \quad \Gamma(v) = B \quad \Gamma \sqcap u : B \sqcap v : A \vdash P}{\Gamma \vdash [u = v]\, P; Q}$$

(T-Out@)
$$\frac{\Gamma^{\mathsf{w}}(u) \downarrow \quad \Gamma \vdash \tilde{v} : \tilde{B}}{\Gamma \vdash u\langle \tilde{v} @ \tilde{B} \rangle}$$

(T-In@)
$$\frac{\Gamma^{\mathsf{r}}(u) \downarrow \quad \Gamma, \tilde{x} : \tilde{A} \vdash P}{\Gamma \vdash u(\tilde{x} @ \tilde{A}).P}$$

Labelled Transitions The transitions for parallel composition, restriction and replication are standard. The remaining transitions are given below. In rule (PI-CLOSE@), we tacitly assume that the tuples $\tilde{B}$ and $\tilde{B}'$ have the same arity, and the notation $\tilde{B} <: \tilde{B}'$ is short for the subtyping judgments on the components types.

(PI-MATCH)
$$\frac{P \xrightarrow{\alpha} P'}{[a = a]\, P; Q \xrightarrow{\alpha} P'}$$

(PI-MISMATCH)
$$\frac{a \neq b \qquad Q \xrightarrow{\alpha} Q'}{[a = b]\, P; Q \xrightarrow{\alpha} Q'}$$

(PI-OUTPUT@)
$$\frac{}{a\langle \tilde{v} @ \tilde{B} \rangle \xrightarrow{a\langle \tilde{v} @ \tilde{B} \rangle} \mathbf{0}}$$

(PI-INPUT@)
$$\frac{}{a(\tilde{x} @ \tilde{B}).P \xrightarrow{a(\tilde{v} @ \tilde{B})} P\{\tilde{v}/\tilde{x}\}}$$

(PI-OPEN@)
$$\frac{P \xrightarrow{(\tilde{c})a\langle \tilde{v} @ \tilde{B} \rangle} P' \qquad b \neq a,\ b \in fn(\tilde{v})}{(\mathsf{new}\ b)P \xrightarrow{(b,\ \tilde{c})\, a\langle \tilde{v} @ \tilde{B} \rangle} P'}$$

(PI-CLOSE@)
$$\frac{P \xrightarrow{(\tilde{c})a\langle \tilde{v} @ \tilde{B} \rangle} P' \quad Q \xrightarrow{a(\tilde{v} @ \tilde{B}')} Q' \quad \tilde{B} <: \tilde{B}' \quad \tilde{c} \cap fn(Q) = \varnothing}{P \mid Q \xrightarrow{\tau} (\mathsf{new}\ \tilde{c})(P' \mid Q')}$$

Typing System. The typing rules, in Table 1, are largely standard, but with important specificities. The typing of matching is inherited directly from [14]: as in that case it allows a more liberal typing of the *then* branch of the conditional, based on the knowledge that the two values tested are indeed the same. To illustrate, as a result of that rule, the following judgment is derivable: $x : \mathsf{r}, y : \mathsf{w} \vdash [x = y]\, x\langle\rangle; \mathbf{0}$. The rules for input and output are distinctive of our typing system. In (T-Out@) the tuple $\tilde{v}$, output at channel u, is tagged with a type B with the intention to force its delivery at the type $\tilde{B}$ (or at a supertype). As we show below, a dynamic typecheck performed upon synchronization ensures that values delivered at a given type will only be received at the same or higher types.

Operational Semantics. The dynamics of the calculus is defined in terms of a labelled transition system built around the actions $\alpha \in \{\tau, u(\tilde{v} @ \tilde{A}), (\tilde{c})\, u\langle \tilde{v} @ \tilde{A} \rangle\}$. The output transition $(\tilde{c})\, u\langle \tilde{v} @ \tilde{A} \rangle$ carries a type tag along with the output value: it represents the output of (a tuple, possibly including fresh) values $\tilde{v}$ at the types $\tilde{A}$; dually, the input action $u(\tilde{v} @ \tilde{A})$ represents input of $\tilde{v}$ at the types $\tilde{A}$. The τ-transition represents internal synchronization, as usual. The novelty, with respect to companion process calculi is that it is dynamically typed: complementary labels synchronize only when they agree on the type of the value exchanged. As proved in [10, 13], the dynamic check ensures that well-typing is preserved by reduction.

Observational Equivalence. The notion of observational equivalence, based on weak bisimulation, is inherited almost directly from [14]. As usual in typed equivalences, we observe the behavior of processes by means of contexts that have a certain knowledge of the processes, represented by a set of type assumptions contained in a type environment. However, following [14], we take the view that the typing information available to the context may be different (less informative) than the information available to the system. Thus, while the system processes may perform certain actions because they possess the required (type) capabilities, the same may not be true of the context. We formalize these intuitions below, following [14].

Given two type environments Γ and I, we say that Γ is *compatible* with I if and only if $\mathrm{dom}(\Gamma) = \mathrm{dom}(\mathrm{I})$ and $\Gamma <: \mathrm{I}$. Furthermore, we say that a type environment is *closed* if it has only names (not variables) in its domain.

Definition 1. *A type-indexed relation $\mathscr{R}$ is a family of binary relations between processes indexed by closed type environments. We write $\mathrm{I} \models P \mathscr{R} Q$ to mean that (i) P and Q are related by $\mathscr{R}$ at I and (ii) there exist Γ and Δ compatible with I such that $\Gamma \vdash P$ and $\Delta \vdash Q$. We often write $\mathrm{I} \models P \mathscr{R} Q$ as $P\ \mathscr{R}_{\mathrm{I}}\ Q$.*

Definition 2 (Contextuality). *A type-indexed relation $\mathscr{R}$ is contextual whenever (i) $\mathrm{I} \vDash P \mathscr{R} Q$ and $a \notin \mathrm{dom}(\mathrm{I})$ implies $\mathrm{I}, a : A \vDash P \mathscr{R} Q$, (ii) $\mathrm{I} \vDash P \mathscr{R} Q$ and $\mathrm{I} \vdash R$ imply $\mathrm{I} \vDash (P \mid R) \mathscr{R} (Q \mid R)$, and (iii) $\mathrm{I}, a : A \vDash P \mathscr{R} Q$ implies $\mathrm{I} \vDash ((\mathsf{new}\ a{:}A)P) \mathscr{R} ((\mathsf{new}\ a{:}A)Q)$.*

Given a type environment I and a closed process P, we define the barb predicate relative to the environment I as follows. First define $P \downarrow_a \triangleq \exists P'$ such that $P \xrightarrow{(\tilde{c})a\langle \tilde{b} @ \tilde{B} \rangle} P'$ and $P \Downarrow_a$ iff $\exists P'$ such that $P \Longrightarrow P' \wedge P' \downarrow_a$, where $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$. Then we define the desired predicate as follows: $\mathrm{I} \vDash P \downarrow_a \triangleq \mathrm{I}(a)^{\mathsf{r}} \downarrow \wedge P \downarrow_a$, and $\mathrm{I} \vDash P \Downarrow_a \triangleq \mathrm{I}(a)^{\mathsf{r}} \downarrow \wedge P \Downarrow_a$.

Definition 3 (Typed behavioral equivalence). *Typed behavioral equivalence, noted $\cong^{\pi}$, is the largest symmetric and contextual type-indexed equivalence relation $\mathscr{R}$ such $\mathrm{I} \models P \mathscr{R} Q$ implies (i) if $\mathrm{I} \vDash P \downarrow_n$ then $\mathrm{I} \vDash Q \Downarrow_n$, and (ii) if $P \xrightarrow{\tau} P'$ then $Q \Longrightarrow Q'$ and $\mathrm{I} \models P' \mathscr{R} Q'$ for some Q'.*

A fundamental difference between our notion of equivalence and that of [14] arises as a consequence of the different typing disciplines in the two systems. In our system, the rule for synchronization guarantees that any name emitted on a public channel is received by the context at the static coercion type used upon output. Conversely, in [14], the type at which the context acquires the new

name is determined by the typing assumptions available to the context on the channel used for output. To illustrate, under the typing[2] $n : \mathsf{r}\langle\mathsf{r}\langle T\rangle\rangle$, a context interacting with the process $n\langle m\rangle$ will acquire m as a channel of type $\mathsf{r}\langle T\rangle$, regardless of the actual type of m, hence even though m was sent on n as a fully-fledged channel. Clearly, this kind of behavior presupposes that the context plays by the rules and will not try to acquire from the value emitted more than allowed by the read type of the transmission channel. As a consequence, reasoning on access control policies in system like [14] can hardly be carried out without assuming that the context is well-typed, with the same type system. In our solution, instead, an appropriate use of coercion may be put in place to prevent the context from acquiring, upon output by the system, more information than it was intended to. In fact, we may code the system's output simply as $n\langle m@\mathsf{r}\rangle$ and have the synchronization rule ensure that the context will not have more than the intended capabilities on m. Interestingly, a corresponding mechanism can be realized in terms of a low-level construction in the applied pi-calculus to provide such guarantees even in the presence of hostile contexts.

3. The applied pi calculus

The applied pi-calculus we use is an asynchronous version of the original calculus of [1], in which we assume that destructors are only used in let-expressions and may not occur in arbitrary terms. This is becoming common practice in the presentations of the applied pi-calculus [6, 7, 5].

As for the high-level calculus, we presuppose countable sets of names and variables, under the same notational conventions. In addition the calculus is characterized by a finite set of function symbols Σ from which terms may be formed. As in [7], we distinguish constructors and destructors, and use the former to build terms and the latter to take terms apart. Constructors are typically ranged over by f, destructors by d. Terms are built around variables and constructors, expressions correspond to destructor application:

$$\begin{array}{rcll} M, N & ::= & a, b, ... & \textit{channel names} \\ & & x, y, ... & \textit{variables} \\ & & f(M_1, ..., M_n) & \textit{constructor application} \\ E & ::= & d(\tilde{M}) & \textit{expression} \end{array}$$

A value is a term without variables. We always assume that constructors are applied consistently with their arity. Processes are defined as follows:

$$\begin{array}{rcl} P, Q, & ::= & \mathbf{0} \mid M\langle\tilde{N}\rangle \mid M(\tilde{x}).P \mid P \mid Q \mid (\mathsf{new}\ n)P \\ & & \mid \quad !\,P \mid \mathsf{let}\ x = E\ \mathsf{in}\ P\ \mathsf{else}\ Q \end{array}$$

Input prefixing, let, and restriction are binders: $M(x).P$ and $\mathsf{let}\ x = E\ \mathsf{in}\ P\ \mathsf{else}\ Q$ bind the variable x in P, $(\mathsf{new}\ n)P$ binds the name n in P. The notions of free and bound names/variables arise as expected. The process $\mathsf{let}\ x = E\ \mathsf{in}\ P\ \mathsf{else}\ Q$ tries to evaluate E; if that succeeds x is bound to the resulting term and the process continues as P (with the substitution in place). Otherwise the process reduces to Q. The evaluation of E is governed by a set of definitions which give semantics to the destructor. Each definition has the form $d(\tilde{M}) \doteq N$ where the terms $\tilde{M}$ and N have no free names and $fv(N) \subseteq fv(\tilde{M})$. Then $d(\tilde{M})$ is defined only if there is a definition $d(\tilde{M'}) \doteq N$ and a substitution σ such that $\tilde{M} = \tilde{M'}\sigma$, in which case $d(\tilde{M})$ evaluates to the term $N\sigma = N^*$, noted $d(\tilde{M}) \to N^*$. Conversely, we note $d(\tilde{M}) \not\to$ whenever there is no $\tilde{M'}\sigma = \tilde{M}$ with a defining equation. We say that a constructor is *one-way* if no destructor application ever returns any of the constructor's arguments.

[2] Here $\mathsf{r}\langle\mathsf{r}\langle T\rangle\rangle$ indicates the type of a read-only channel carrying values that are themselves read-only channels with payload of type T.

We always omit trailing $\mathbf{0}$ processes and write $\mathsf{let}\ x = E\ \mathsf{in}\ P$ instead of $\mathsf{let}\ x = E\ \mathsf{in}\ P\ \mathsf{else}\ \mathbf{0}$. We also combine multiple let-bindings in a single let instead of writing the corresponding nested definitions. Finally, we define:

$$\mathsf{rec}\ X.P \triangleq (\mathsf{new}\ a)\ (a\langle\rangle \mid !a().P\{a\langle\rangle/X\}) \quad a \notin fn(P)$$

Our applied pi-calculus includes destructors to project the elements of tuples (noted π_i), as well as two constructors for lists, :: (cons) and $\emptyset$ (nil), together with the the standard destructors hd and tl:

$$hd(x :: y) \doteq x \quad tl(x :: y) \doteq y$$

In addition, we introduce the one-way constructors $hash$, ek, dk, sk, pub, $priv$ that we employ as the basic building blocks of the cryptosystem used by the implementation. The constructor $hash$ generates a hash from the seed M, and we assume that the hash function is collision-free: $hash(M) = hash(N)$ implies $M = N$. The two unary constructors ek and dk generate encryption and decryption keys $ek(M)$ and $dk(M)$ from a seed M; as in [1] we essentially view M as a generator of unguessable seeds. We often abbreviate $ek(M)$ to M^+ and $dk(M)$ to M^-. Finally, sk generates a shared key $sk(M)$ from the seed M, while pub and $priv$ generate a pair of public/private keys $pub(M)$ and $priv(M)$ associated with the seed M. We often abbreviate $pub(M)$ with M_{ID}.

We may now define our cryptosystem in terms of two further constructors, $sign$, and $cipher$, and the destructors $decipher$, $verify$, defined by the following equations:

$$\begin{array}{rcl} decipher(cipher(x, ek(y)), dk(y)) & \doteq & x \\ decipher(cipher(x, sk(y)), sk(y)) & \doteq & x \\ verify(sign(x, priv(y)), pub(y)) & \doteq & x \end{array}$$

Signatures are built using the binary constructor $sign$ and checked by using the destructor $verify$ [6, 7]; encrypted packets, in turn, are formed around the binary constructor $cipher$, and taken apart by using the destructor $decipher$. In the implementation, we use two further constructors, rd and wr, to construct different sets of keys associated with the same seed (cf. Section 4). We often use the conventional spi-calculus notation $\{\tilde{M}\}_N$ for the encrypted packet $cipher(\tilde{M}, N)$. Following [7], we define an $\mathsf{if} - \mathsf{then} - \mathsf{else}$ construct in terms of let as shown below:

$$\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleq \mathsf{let}\ x = equals(M, N)\ \mathsf{in}\ P\ \mathsf{else}\ Q$$

Here we assume $x \notin fv(P, Q)$, and $equals$ is a further destructor with defining equation $equals(x, x) \doteq x$. Finally, we introduce an explicit form of decryption to bind multiple variables as in the original spi-calculus (again, we assume x fresh in P and Q)

$$\begin{array}{l} \mathsf{decrypt}\ M\ \mathsf{as}\ \{\tilde{y}\}_N\ \mathsf{in}\ P\ \mathsf{else}\ Q \triangleq \\ \quad \mathsf{let}\ x = decipher(M, N)\ \mathsf{in} \\ \quad\quad \mathsf{let}\ y_1 = \pi_1(x), \ldots, y_n = \pi_n(x)\ \mathsf{in}\ P\ \mathsf{else}\ Q \end{array}$$

Operational semantics. The dynamics of the calculus is defined by means of a labelled transition system. Following an increasingly common practice [7, 5], the transitions do not use the active substitutions introduced in the original formulation of [1].

The core of the LTS is given in In Table 2: we omit the remaining transitions (for parallel composition, replication and restriction) which are entirely standard. For simplicity, we require that all synchronizations occur on channel names, rather then arbitrary terms. The treatment of let is taken from [7]. The transitions are only defined over closed processes (with no free variables). Given and such process P, we define: $P \downarrow_a$ if and only if $\exists P'$ such that $P \xrightarrow{(\tilde{n})a\langle\tilde{M}\rangle} P'$, $P \Downarrow_a$ if and only if $\exists P'$ such that $P \Longrightarrow P' \wedge P' \downarrow_a$.

Behavioral equivalence. As in Section 2, we rely on a notion of behavioral equivalence based on weak bisimulation, and relative

Table 2 Labelled transitions for the applied pi-calculus

$$\text{(OUT)}\quad \frac{}{a\langle \tilde{M}\rangle \xrightarrow{a\langle \tilde{M}\rangle} \mathbf{0}} \qquad \text{(IN)}\quad \frac{}{a(\tilde{x}).P \xrightarrow{a(\tilde{M})} P\{\tilde{M}/\tilde{x}\}}$$

$$\text{(OPEN)}\quad \frac{P \xrightarrow{(\tilde{b})\, a\langle \tilde{M}\rangle} P' \qquad c \neq a,\ c \in fn(\tilde{M})}{(\mathsf{new}\ c)P \xrightarrow{(\tilde{b},c)\, a\langle \tilde{M}\rangle} P'}$$

$$\text{(CLOSE)}\quad \frac{P \xrightarrow{(\tilde{b})\, a\langle \tilde{M}\rangle} P' \quad Q \xrightarrow{a(\tilde{M})} Q' \qquad \tilde{b} \notin fn(Q)}{P \mid Q \xrightarrow{\tau} (\mathsf{new}\ \tilde{b})(P' \mid Q')}$$

$$\text{(LET)}\quad \frac{d(\tilde{M}) \to N \quad P\{N/x\} \xrightarrow{\alpha} P'}{\mathsf{let}\ x = d(\tilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q \xrightarrow{\alpha} P'} \qquad \text{(LET-ELSE)}\quad \frac{d(\tilde{M}) \not\to \quad Q \xrightarrow{\alpha} Q'}{\mathsf{let}\ x = d(\tilde{M})\ \mathsf{in}\ P\ \mathsf{else}\ Q \xrightarrow{\alpha} Q'}$$

to contexts with a certain knowledge about names and terms. The formal definitions are mostly based on the work of Boreale, De Nicola and Pugliese in [8].

A *term environment* ρ is a finite substitution from variables to values. We write $fn(\rho)$ to mean $fn(\mathrm{Range}(\rho))$. Substitutions may only be extended with new bindings for fresh variables: $\rho, M/x$ indicates the extension of ρ with $x \notin \mathrm{dom}(\rho)$. Given a term environment ρ, we let $\mathscr{A}(\rho)$ be the analysis of ρ, that is, the environment obtained by extending ρ with new bindings for the terms resulting from the application of destructors to the range of ρ. Formally:

Definition 4. *The analysis $\mathscr{A}(\rho)$ of ρ is the smallest substitution σ extending ρ that is closed by the following rule:*

$$\frac{d(\tilde{N}) \doteq N \quad \exists \sigma' \,.\, \tilde{N}\sigma' \subseteq \mathrm{Range}(\sigma) \quad (z \notin \mathrm{dom}(\sigma))}{N\sigma'/z \in \sigma}$$

Abusing the notation we often write $N \in \mathscr{A}(\rho)$ to mean $N \in \mathrm{Range}(\mathscr{A}(\rho))$. Given a process P we say that ρ defines P, written $\rho \vdash P$, if $fv(P) \subseteq \mathrm{dom}(\rho)$ and $fn(P) \cap fn(\rho) = \emptyset$. We use the same convention and notation for terms.

Definition 5 (Term-indexed relation). *A term-indexed relation $\mathscr{R}$ is a family of binary relations between closed processes indexed by term environments. We write $\rho \models P \mathscr{R} Q$ (or equivalently $P\ \mathscr{R}_\rho\ Q$) to mean that P and Q are related by $\mathscr{R}$ at ρ and that $fn(P, Q) \subseteq fn(\rho)$.*

We have a notion of contextuality corresponding to that given in Def. 2. We note $\rho \setminus n$ the term environment resulting from erasing from ρ all bindings M/x such that $n \in fn(M)$.

Definition 6 (Contextuality). *A term-indexed relation $\mathscr{R}$ is contextual whenever $\rho \models P \mathscr{R} Q$ implies (i) if $\rho \vdash R$ then $\rho \models (P \mid R\rho)\ \mathscr{R}(Q \mid R\rho)$, (ii) $\rho, n/x \models P \mathscr{R} Q$ with $n \notin fn(\rho)$, and (iii) $\rho \setminus n \models ((\mathsf{new}\ n)P)\mathscr{R}((\mathsf{new}\ n)Q)$.*

The barb predicate is defined relative to a term-environment, as expected: $\rho \models P \downarrow_a \triangleq\ a \in \mathscr{A}(\rho) \wedge\ P \downarrow_a$, and $\rho \models P \Downarrow_a \triangleq a \in \mathscr{A}(\rho) \wedge\ P \Downarrow_a$.

Definition 7 (Behavioral equivalence). *Behavioral equivalence, noted $\cong^{A\pi}$, is the largest symmetric and contextual term-indexed relation $\mathscr{R}$ such that $\rho \models P \mathscr{R} Q$ implies (i) if $\rho \models P \downarrow_n$ then $\rho \models Q \Downarrow_n$, and (ii) if $P \xrightarrow{\tau} P'$ then $\exists Q'.\ Q \Longrightarrow Q'$ and $\rho \models P' \mathscr{R} Q'$.*

4. The implementation framework

As anticipated, our implementation is based on a compilation of the high-level processes of the pi-calculus into corresponding processes of the spi-calculus representing low-level principals running in an open network.

4.1 Network model

Our assumptions about the low-level communication infrastructure of such network are the same as those of [2]. In particular, we presuppose a Needham-Schroeder model, in which an intruder can interpose a computer in all communication paths and thus alter or copy parts of messages, replay messages or forge new ones. On the other hand, we assume that the intruder cannot gain control of the whole network, and intercept all messages: consequently, message delivery may always be achieved with an adequate degree of redundancy. To model such eavesdropper we assume that principals may only send and receive messages through a public network interface consisting of a communication channel net (typically all the exchanges over this channel are encrypted). On the other hand, each principal has a secure environment in which to compute and store private data.

4.2 Data structures for names

The design of the implementation is centered around the choice of an appropriate data structure to represent the names and the capability types of the high-level calculus. As we mentioned at the outset, the idea is to represent the capability types with corresponding term capabilities, implemented as encryption keys. Specifically, the representation of the name n as a fully fledged channel includes the name identity and two encryption keys corresponding to the read and write capabilities: $(pub(n), ek(wr(n)), ek(rd(n)))$. The application of the one-way constructors rd and wr provides a way to construct the read and write seeds associated with the key-pairs that implement the capability types attached to n.

The representation of a name is now obtained by introducing self-signed certificates [6, 4] of the form:

$$Cert(n) \triangleq \{pub(n), hash(ek(wr(n))), hash(ek(rd(n)))\}_{priv(n)}$$

The certificates help determine the "type" of the names they are attached to. To illustrate, suppose we receive a tuple formed as $(M_0, M_1, M_2, Cert(n))$, with M_i arbitrary terms. One may first ensure that the certificate corresponds to the correct identity by using the public key $pub(n)$ to verify the certificate, and then match M_0 with the public key. As this stage, one may decide whether or not M_1 and M_2 are valid capabilities for the identity M_0 by calculating $hash(M_1)$ and $hash(M_2)$ and by checking whether they match the two values $hash(ek(wr(n)))$ and $hash(ek(rd(n)))$ respectively: in the former case we certify a write capability, in the latter a read capability.

We henceforth let $\underline{n}$ denote the representation of n together with the associated certificate:

$$\underline{n} \;=\; (pub(n), ek(wr(n)), ek(rd(n)), Cert(n)).$$

All the low-level protocols that implement the synchronization steps in the high-level calculus exchange and manipulate quadruples and expect the components of such tuples to adhere to the format above. For uniformity, we write $\underline{x}$ to note a quadruple of variables used to store name representations. Further, we introduce the following mnemonic notation for the components of 4-tuple terms:

$$M_{ID} = \pi_1(M) \quad M_w^+ = \pi_2(M)$$
$$M_r^+ = \pi_3(M) \quad M_{CERT} = \pi_4(M)$$

Notice that we make no assumption on the format of the 4-tuple: hence, only when $M = \underline{n}$ for some name n the projections select the identity, the keys and the certificate associated with n. Abusing the notation, we often write $\underline{n}/\underline{x}$ (and more generally $M/\underline{x}$, when M is a 4-tuple) to indicate the substitution of the components of $\underline{n}$ (or M) for the corresponding variables composing $\underline{x}$.

Given this representation of names, the effect of casting a name at a higher type in the high-level calculus is realized, in the implementation, with a mechanism that masks away the encryption keys corresponding to the type capabilities "lost" in the type cast. This is accomplished by introducing four unary destructors $\Uparrow\mathsf{r}, \Uparrow\mathsf{w}, \Uparrow\mathsf{rw}, \Uparrow\top$ corresponding to the four possible type coercions of the source calculus. We use postfix notation in the defining equations below, and in the rest of the presentation, writing $M\Uparrow T$ instead of $\Uparrow T(M)$.

$$\begin{array}{rcl}(x,y,z,w)\Uparrow \mathsf{rw} & \doteq & (x,y,z,w)\\ (x,y,z,w)\Uparrow \mathsf{r} & \doteq & (x,\top,z,w)\\ (x,y,z,w)\Uparrow \mathsf{w} & \doteq & (x,y,\top,w)\\ (x,y,z,w)\Uparrow \top & \doteq & (x,\top,\top,w)\end{array}$$

When applied to the representation of a name n, the application $\underline{n}\Uparrow T$ implements the coercion of "run-time" type of $\underline{n}$ corresponding to the type-coercion $n@T$: in the implementation, the coercion simply amounts to erasing the appropriate encryption keys.

In the translation, we will find it useful to compute, and manipulate the run-time type of terms. We give a low-level representation of the high-level types by simply treating the latter as nullary constructors: thus the terms $\mathsf{rw}, \mathsf{r}, \mathsf{w}, \top$ in the implementation represent the corresponding high-level types $\mathsf{ch}(\mathsf{rw}), \mathsf{ch}(\mathsf{r}), \mathsf{ch}(\mathsf{w}), \top$. Following the convention adopted for names, we write $\underline{T}$ for the representation of the type T in the low-level calculus. Also, we sometime write $\underline{S} <: \underline{T}$ when $S <: T$. Next, we introduce notation to express a number of useful operations on the representation of names and their (run-time) types. Specifically, we note:

- $\mathsf{if}\ WF(M,\underline{T})\ \mathsf{then}\ P\ \mathsf{else}\ Q$ a process that tests whether M is certified and has type $\underline{S}$, for some $S <: T$;
- $\mathsf{let}\ \tilde{x} = \mathsf{Cast}(M,\underline{T})\ \mathsf{in}\ P$ a process that casts the type of M to $\underline{T}$ and binds the resulting term and the certificate to $\tilde{x}$ in P (this is the dynamic counterpart of the $\Uparrow T$ destructors);
- $\mathsf{let}\ \tilde{x} = \mathsf{Meet}\{M^1,\ldots,M^n\}\ \mathsf{in}\ P$ a process that computes a new tuple representing the result of merging the capabilities of the M_i's and binds such tuple to $\tilde{x}$: the result of the meet is defined only if $M^i_{CERT} = M^j_{CERT}$ for all $i,j \in [1..n]$;
- $\mathsf{let}\ x = \mathsf{Typeof}(M)\ \mathsf{in}\ P$ a process that verifies that M represents capabilities certified by M_{CERT}, determines the type of M, and binds that type to x; in such case we informally say that M has such type. Again the operation is undefined if M fails to be certified.

All the operations above can be encoded by means of nested applications of the projection, equality, $verify$ and cast destructors: we omit all details and refer the reader to [13]. Instead, for future reference, we note that such operations will be invoked in our implementation only in the presence of precise invariants that guarantee that they (the operations) are well-defined. More precisely, the communication protocols guarantee the following conditions: (i) whenever we activate the process $\mathsf{let}\ \tilde{x} = \mathsf{Cast}(M,T)\ \mathsf{in}\ P$, M_{CERT} is indeed a certificate that certifies the remaining components of M, and the type of M is $S <: T$, (ii) whenever we evaluate $\mathsf{let}\ \tilde{x} = \mathsf{Meet}\{M^1,\ldots,M^n\}\ \mathsf{in}\ P$, M^i_{CERT} and M^j_{CERT} are the same certificate which certifies the remaining components of M^i and M^j at their respective types for all $i,j \in [1..n]$.

A further important remark is in order on the nature of the certificates and the guarantees they convey. On the one hand, as we argued, the certificates help extract the capabilities from the representation of names. On the other hand, being self-signed, a certificate does not by itself ensure that the capabilities extracted from a term $(M_{ID}, M_w^+, M_w^+, M_{CERT})$ are valid encryption keys. That is indeed the case for names generated by our implementation, but a term with this structure received from an arbitrary context may well be validated at type T without being the representation of any name at any type. To illustrate, the (fake) certificate $C = \{pub(n), hash(M), hash(N)\}_{priv(n)}$ will certify a write capability in $(pub(n), M, \top, C)$ regardless of the structure of M, which can be any term. Fortunately, one can build more structure in the implementation so as to protect against such threats and give guarantees that any term used as a key is indeed a valid key. We show how that can be accomplished in Section 6.

4.3 Message filtering and emission

As in [2], all the communication protocols underlying our implementation run on the public channel net, and hence require the ability for all protocol participants to filter messages, based on different mechanisms.

A first mechanism allows processes to filter replays of messages, based on nonces. We write $\mathsf{if}\ M \notin Set_n\ \mathsf{then}\ P$, for a process that adds M to the list carried on channel n and continues as P whenever M does not occur in the list.

$$\mathsf{if}\ M \notin Set_n\ \mathsf{then}\ P \triangleq \\ n(y).(\mathsf{if}\ M \notin y\ \mathsf{then}\ n\langle M :: y\rangle \mid P\ \mathsf{else}\ n\langle y\rangle)$$

The testing process $\mathsf{if}\ M \notin N\ \mathsf{then}\ P\ \mathsf{else}\ Q$ can be implemented by nested applications of head, tail and equal destructors: we omit all details and refer again to [13].

The input of a message uses two filtering protocols. The first picks a packet from the net and proceeds with the continuation only if the message is successfully decrypted; otherwise it re-emits the message and retries:

$$\mathsf{filter}\ \tilde{y}\ \mathsf{with}\ N\ \mathsf{in}\ P \triangleq \\ \mathsf{rec}\ X.net(x).\mathsf{decrypt}\ x\ \mathsf{as}\ \{\tilde{y}\}_N\ \mathsf{in}\ P\ \mathsf{else}\ (net\langle x\rangle \mid X)$$

The second protocol filters messages based on their types, and upcasts them at those types.

$$\mathsf{filter}\ \underline{x}@t\ \mathsf{from}\ c\ \mathsf{in}\ P \triangleq \\ \mathsf{rec}\ X.c(\underline{y}).\mathsf{if}\ WF(\underline{y},t)\ \mathsf{then}\ (\mathsf{let}\ \underline{x} = \mathsf{Cast}(\underline{y},t)\ \mathsf{in}\ P) \\ \mathsf{else}\ (c\langle\underline{x}\rangle \mid X)$$

The output of a message M on a network is realized by emitting M on the net channel, typically encrypted. To ensure delivery, the emission is replicated, and packaged with a fresh nonce, n, to protect against replay attacks. The nonce also acts as a confounder

for cryptanalysis attacks.

$$\mathsf{emit}(\{M\}_k) \triangleq (\mathsf{new}\ n)!\, net\langle\{M, n\}_k\rangle$$

5. A First Implementation

We are ready to give a formal definition of the two complementary components of the implementation: the channel infrastructure for communication, and the compilation of high-level processes. As anticipated, the idea is to set up appropriate channel servers to support write and read requests for synchronization and value exchange; the high-level processes of the pi calculus, in turn, are represented as clients running corresponding protocols with the servers.

5.1 Channels and communication protocols

Each channel server is associated with two pairs of asymmetric keys employed in the protocols for reading and for writing, respectively. The encryption keys are circulated among clients as components of the tuples that represent the high-level names; the decryption keys, in turn, are stored securely at the channel servers.

Write protocol. On the client side, writing on a channel is accomplished by emitting a packet encrypted under the channel's write encryption key. The message is replicated to ensure delivery, and packaged with a nonce to protect against replay attacks. The server, in turn, uses the write decryption key to receive the message, uses the nonce to filter multiple copies of the message (hardly realistic, of course: indeed, the solution from [2], based on a challenge response mechanisms, would work just as well here), and stores them into a private queue. It also filters based on the format of the messages received, requiring that they match the format expected of the encoding of names.

Read Protocol. On the client side, the reader process uses the channel's read encryption key to send a request to the server. The request takes the form of pair including a symmetric key that will be used as a session key to exchange the message with the server, and (the representation of) a type at which the client expects its input value. The client then waits for a message from the server encrypted under the session key: upon receiving the packet, it proceeds with its continuation. The server, in turn, uses the channel's read decryption key to receive a request from the client. To protect against replays, the server keeps track of the nonces received on a private channel. After checking the freshness of the request, the server uses the type to select one of the message from its private queue and then packages the message with the key. The nonce can be spared on the actual message sent by the server as the key expires at the completion of the protocol (the server may easily filter out replays of the session key).

The definition of the channel servers is reported in Table 3. We use the notation n_r^- and n_w^- to refer to $dk(rd(n))$ and $dk(wr(n))$ respectively. The two private names n° and n^* hold, respectively, the message queue and the nonce set associated with n. The definitions, as given, are adequate for monadic exchanges only, but the extension to the polyadic case is relatively straightforward, as one may rely on corresponding polyadic exchanges in the implementation language. Indeed, we could simply associate an arity with each channel, and parameterize the definition of the filters (on the server and client sides) on that arity.

5.2 Compilation of high-level processes

The translation of high-level processes is given by induction on the typing judgments of the pi-calculus, and is only defined for derivable judgments. We remark, however, that the output of the translation only depends on the structure of the processes, as the latter contain enough information to guide the generation of the low-level code. On the other hand, the typing information is useful in stating and proving the properties of the translation, whenever we need to draw precise connections between the static capability types of the high-level processes and the dynamic term capabilities of the low-level principals.

The compilation function takes an extra argument, noted μ, that collects information on the equalities corresponding to the matching prefixes encountered in the traversal of the process being compiled. This information is expressed as a set of bindings $u \leftrightarrow v$, with u and v syntactically different (if u and v are syntactically identical, the binding conveys no information). We note $\mu[u{\leftrightarrow}v]$ the extension of μ with a new binding. Also, we write $\mu \vdash u{\leftrightarrow}v$ to indicate that $u{\leftrightarrow}v$ is derivable from the equational theory generated by μ (notice that $\mu \vdash u{\leftrightarrow}v$ is decidable for finite $\mu's$).

The information conveyed by the argument μ is exploited by the compiler to generate code that reconstructs the run-time type of a name, gathering the encryption keys corresponding to the capability types available in the source calculus. More precisely, the term formed by the compiler-generated code represents the run-time counterpart of the meet taken on the types of all the names (or variables) that occur in an match enclosing the process being compiled. In Table 3, we express this calculation using the notation $\mathsf{let}\ \underline{x} = [\![\, u \,]\!]_\mu\ \mathsf{in}\ P$, which may be defined formally as the process $\mathsf{let}\ \underline{x} = \mathsf{Meet}\ \{\underline{w} \mid \mu \vdash u \leftrightarrow w\}\ \mathsf{in}\ P$, that computes the expected meet and binds it to the fresh tuple of variables $\underline{x}$. Notice that the set $\{\underline{w} \mid \mu \vdash u \leftrightarrow w\}$ is always non-empty (even for an empty μ) as $\mu \vdash u{\leftrightarrow}u$ may be derived by reflexivity, for all μ and u.

We are now ready to illustrate the clauses of the translation. An output in the source calculus is compiled into a corresponding emission: before that, however, the compiled code computes the appropriate representation for the term to be output and collects the cryptographic key required to form the packet emitted on the public network. Likewise, an input process is compiled into code that runs the read protocol after having collected the appropriate key. A high-level match is compiled into the corresponding conditional low-level process: the name-equality test in the pi-calculus is realized as a corresponding test on the public keys associated with the names involved. In addition, in case the test is successful, the continuation process is given access, via μ, to the new set of capabilities that corresponds to the meet of the two types associated with the names equated in the pi-process. As for restriction, the translation of a high-level restriction generates a corresponding restriction together with a new channel to be associated with the newly generated name. The remaining clauses are defined homeomorphically. The definition of the translation function is completed by a clause that introduces the top-level compilation map $[\![\, \cdot \,]\!]$.

5.3 Properties of the implementation

The format of the communication servers and the structure of the client code resulting from the translation ensure the following properties: (i) each message output by a writer will reach at most one reader, and dually, (ii) a legitimate reader client will complete the protocol provided that a type-compatible message on the same channel has been output by a writer. Thus, if appropriate channels are allocated for the free names of the high-level processes, one can prove that any pi-calculus synchronization on a name is simulated by a corresponding reduction sequence in the implementation, and conversely that the synchronizations on the channel queues in the implementation reflect the τ-reductions of the source calculus.

Given the relative complexity of the implementation, the proof of operational correspondence is elaborate. As a first, basic step, one needs a proof that the run-time flow and management of the cryptographic keys representing the high-level type capabilities

Table 3 Channels and a compositional translation of client processes

Channels

$$
\begin{array}{rcl}
WS_n & \triangleq & \mathbf{!}\,\mathsf{filter}\ (\underline{x}, z)\ \mathsf{with}\ n_w^-\ \mathsf{in\ if}\ z \notin Set_{n^*}\ \mathsf{then}\ n^\circ\langle \underline{x}\rangle \\
RS_n & \triangleq & \mathbf{!}\,\mathsf{filter}\ (y, t, z)\ \mathsf{with}\ n_r^-\ \mathsf{in\ if}\ z \notin Set_{n^*}\ \mathsf{then\ filter}\ \underline{x}@t\ \mathsf{from}\ n^\circ\ \mathsf{in}\ \mathbf{!}\, net\langle\{\underline{x}\}_y\rangle \\
\mathsf{Chan}_n & \triangleq & (\mathsf{new}\ n^*, n^\circ)\ n^*\langle\emptyset\rangle \mid RS_n \mid WS_n
\end{array}
$$

Clients – The remaining clauses are defined homeomorphically: $[\![\,\Gamma \triangleright P \mid Q\,]\!]_\mu = [\![\,\Gamma \triangleright P\,]\!]_\mu \mid [\![\,\Gamma \triangleright Q\,]\!]_\mu$, $[\![\,\Gamma \triangleright \mathbf{!}\,P\,]\!]_\mu = \mathbf{!}\,[\![\,\Gamma \triangleright P\,]\!]_\mu$.

$$
\begin{array}{rcl}
[\![\,\Gamma \triangleright u\langle v@T\rangle\,]\!]_\mu & \triangleq & \mathsf{let}\ \hat{\underline{u}} = [\![\,u\,]\!]_\mu,\ \hat{\underline{v}} = [\![\,v\,]\!]_\mu\ \mathsf{in\ emit}(\{\hat{\underline{v}}{\Uparrow}\underline{T}\}_{\underline{u}_w^+}) \\
[\![\,\Gamma \triangleright u(x@T).P\,]\!]_\mu & \triangleq & (\mathsf{new}\ k)\ (\mathsf{let}\ \hat{\underline{u}} = [\![\,u\,]\!]_\mu\ \mathsf{in\ emit}(\{sk(k), \underline{T}\}_{\underline{u}_r^+})) \mid \mathsf{filter}\ \underline{x}\ \mathsf{with}\ sk(k)\ \mathsf{in}\ [\![\,\Gamma, x{:}T \triangleright P\,]\!]_\mu \\
[\![\,\Gamma \triangleright [u = v]\,P; Q\,]\!]_\mu & \triangleq & \mathsf{if}\ \underline{u}_{ID} = \underline{v}_{ID}\ \mathsf{then}\ [\![\,\Gamma{\sqcap}v{:}\Gamma(u){\sqcap}u{:}\Gamma(v) \triangleright P\,]\!]_{\mu[u \leftrightarrow v]}\ \mathsf{else}\ [\![\,\Gamma \triangleright Q\,]\!]_\mu \\
[\![\,\Gamma \triangleright (\mathsf{new}\ n : A)P\,]\!]_\mu & \triangleq & (\mathsf{new}\ n)(\mathsf{Chan}_n \mid [\![\,\Gamma, n : A \triangleright P\,]\!]_\mu) \\
[\![\,\Gamma \triangleright \mathbf{0}\,]\!]_\mu & \triangleq & \mathbf{0}
\end{array}
$$

Top−level translation $[\![\,\Gamma \triangleright P\,]\!] \triangleq [\![\,\Gamma \triangleright P\,]\!]_\emptyset$

provides the compiled code with enough capabilities to simulate all of the synchronizations of the well-typed source processes. That can be proved by showing that the translation is closed by substitution, in the sense made precise below.

Given a type environment I, define the term environment corresponding to I, noted $\{\!|\,\mathrm{I}\,|\!\}$, as follows:

$$\{\!|\,\emptyset\,|\!\} = \{net/x_o\}, \quad \{\!|\,\mathrm{I}, a : A\,|\!\} = \{\!|\,\mathrm{I}\,|\!\}, \underline{a}{\Uparrow}\underline{A}/\underline{x}$$

where x_o is a distinguished (arbitrary) variable and $\underline{x} \notin \mathrm{dom}(\{\!|\,\mathrm{I}\,|\!\})$. Now we can state the desired closure properties for the translation.

Lemma 1 (Substitution Closure). *Let Γ be a closed type environment such that $\Gamma \vdash v : A$. Then:*

$$[\![\,\Gamma, x{:}A \triangleright P\,]\!]\{(\underline{v}{\Uparrow}\underline{A})/\underline{x}\} \cong^{A\pi}_{\{\!|\,\Gamma\,|\!\}} [\![\,\Gamma \triangleright P\{v/x\}\,]\!]$$

Proof. (Outline) We prove a much stronger closure property, showing that the treatment of matching ensures the expected correspondence between static and dynamic type capabilities:

$$[\![\,\Gamma, x{:}A \triangleright P\,]\!]_\mu\{(\underline{v}{\Uparrow}\underline{A})/\underline{x}\} \equiv_{\{\!|\,\Gamma\,|\!\}} [\![\,\Gamma \triangleright P\{v/x\}\,]\!]_{\mu\{v/x\}} \qquad (*)$$

Here $\equiv_{\{\!|\,\Gamma\,|\!\}}$ denotes a congruence relation that relates translated pi-processes that are structurally congruent up to type consistent substitutions of the form: $\{\underline{v_i}{\Uparrow}T_i/\underline{x_i} \mid \Gamma(v_i) <: T_i <: \Gamma(x_i)\}$. The congruence $(*)$ may be proved by induction on the structure of the compiled process. Then the lemma follows by observing that $\equiv_{\{\!|\,\Gamma\,|\!\}}$ is finer than $\cong^{A\pi}$, i.e. $[\![\,\Gamma \triangleright P\,]\!] \equiv_{\{\!|\,\Gamma\,|\!\}} [\![\,\Gamma \triangleright Q\,]\!]$ implies $[\![\,\Gamma \triangleright P\,]\!] \cong^{A\pi}_{\{\!|\,\Gamma\,|\!\}} [\![\,\Gamma \triangleright Q\,]\!]$. □

To show the main result of operational correspondence, as in [2], we must rely on the presence of noise to prevent traffic analysis. Given the simple network interface we have assumed, injecting noise into the network is simply accomplished by the process

$$W \triangleq \mathbf{!}\,(\mathsf{new}\ n)\mathbf{!}\, net\langle\{n\}_n\rangle$$

which generates infinitely many copies of infinitely many secret packets. Now, we complete the definition of the computing environment for the compiler-generated principals by including a general interface to the network, the noise-generating process and channel support for the free names shared between the processes and the environment.

$$\mathsf{E}_\mathrm{I}[-] \;=\; -\mid W \mid \prod_{n \in \mathrm{dom(I)}} \mathsf{Chan}_n$$

We have finally all we need to establish the result of operational correspondence.

Theorem 2 (Operational Correspondence). *Let Γ and* I *be two closed type environments such that $\Gamma <:$ I. Then:*

- *If $P \Longrightarrow P'$ then $\mathsf{E}_\mathrm{I}[[\![\,\Gamma \triangleright P\,]\!]] \Longrightarrow \cong^{A\pi}_{\{\!|\,\mathrm{I}\,|\!\}} \mathsf{E}_\mathrm{I}[[\![\,\Gamma \triangleright P'\,]\!]]$.*
- *Conversely, if $\mathsf{E}_\mathrm{I}[[\![\,\Gamma \triangleright P\,]\!]] \Longrightarrow K$ then there exists P' s.t. $P \Longrightarrow P'$ and $K \cong^{A\pi}_{\{\!|\,\mathrm{I}\,|\!\}} \mathsf{E}_\mathrm{I}[[\![\,\Gamma \triangleright P'\,]\!]]$.*

Proof. (Outline). The proof of this result, especially the "reflection" direction in the second item, is subtle, because the translation is not "prompt" [16]. In fact, one easily sees that it takes several steps for $\mathsf{E}_\mathrm{I}[[\![\,\Gamma \triangleright P\,]\!]]$ to be ready for the commit synchronization on the channel queue that corresponds to the high-level synchronization on the channel. As it turns out, however, these steps are not observable and can be factored out in the proof by resorting to a suitable notion of (term-indexed) *administrative* equivalence, noted $\approx^A$ and included in $\cong^{A\pi}$. The definition of $\approx^A$ draws on a classification of the reductions of the compiled processes into *commitment* steps, corresponding to synchronizations on the channel queues, and *administrative reductions*, corresponding to the steps that precede and follow the commitment steps. Then two processes are equated by $\approx^A$ only if they are behaviorally equivalent and, in addition, they can simulate each other's commitment transitions in a 'strong' way.

The relation $\approx^A$ can be used to prove the following strong variant of operational correspondence:

- If $P \xrightarrow{\tau} P'$ then $\mathsf{E}_\mathrm{I}[[\![\,\Gamma \triangleright P\,]\!]] \Longrightarrow \approx^A_{\{\!|\,\mathrm{I}\,|\!\}} \mathsf{E}_\mathrm{I}[[\![\,\Gamma \triangleright P'\,]\!]]$.
- Conversely, assume $H \approx^A_{\{\!|\,\mathrm{I}\,|\!\}} \mathsf{E}_\mathrm{I}[[\![\,\Gamma \triangleright P\,]\!]]$ and $H \xrightarrow{\tau} K$. Then either $H \approx^A_{\{\!|\,\mathrm{I}\,|\!\}} K$, or there exists P' s.t. $P \xrightarrow{\tau} P'$ and $K \approx^A_{\{\!|\,\mathrm{I}\,|\!\}} \mathsf{E}_\mathrm{I}[[\![\,\Gamma \triangleright P'\,]\!]]$.

Here, the proof of the first item (i.e. the "preservation" direction of the result) follows from Lemma 1 [3]. For the second item, the

[3] More precisely, from a variant of Lemma 1 stated in terms of $\approx^A$ rather than $\cong^{A\pi}$.

first case occurs when the move from H is an administrative step, while the second corresponds to the case when H is finally prompt to commit on a synchronization reduction that reflects a high-level synchronization.

The proof of the theorem derives from this intermediate result and the fact that $\approx^A$ is finer than $\cong^{A\pi}$. $\Box$.

Theorem 3 (Soundness). *Let* Γ, Δ *and* I *be closed type environments s.t.* $\Gamma, \Delta <: \mathrm{I}$. *If* $\{\!|\, \mathrm{I} \,|\!\} \models \mathsf{E}_\mathrm{I}[\![\, \Gamma \triangleright P \,]\!] \cong^{A\pi} \mathsf{E}_\mathrm{I}[\![\, \Delta \triangleright Q \,]\!]$, *then* $\mathrm{I} \models P \cong^{\pi} Q$.

Proof. This follows by a standard argument from Theorem 2 (cf. [9]).

The converse direction of Theorem 3 does not hold. In fact, as we noted, the communication protocols presuppose a certain structure associated with names. Indeed, for the names that are statically shared with the context, this structure is easily enforced by allocating the corresponding channels as part of the initial computing environment. However, the context may dynamically generate new names that do not satisfy the expected invariants. Notice, for instance, that the client of a reader protocol presupposes a legitimate channel on the other end of the protocol and is not protected against malformed messages received by illegitimate channels: given that, it is easy to find a counter-example to full abstraction. For instance, in the source calculus we have:

$$a : \mathsf{w} \models a(y@\mathsf{rw}).y(x@\mathsf{rw}).y\langle x@\mathsf{rw}\rangle \cong^{\pi} a(y@\mathsf{rw})$$

This is an instance of the well-known asynchronous pi calculus law $a(x).a\langle x\rangle \cong \mathbf{0}$, and holds in our pi calculus for similar reasons. On the other hand, one easily sees that for $\mathrm{I} :> \Gamma$ we have

$$a_w^+/z \not\models \mathsf{E}_\mathrm{I}[\![\, \Gamma \triangleright a(y@\mathsf{rw}).y(x@\mathsf{rw}).y\langle x@\mathsf{rw}\rangle \,]\!] \cong^{A\pi} \mathsf{E}_\mathrm{I}[\![\, \Gamma \triangleright a(y@\mathsf{rw}) \,]\!]$$

In fact, a context may create the legitimate representation of a full-fledged name $\underline{b}$ and exchange it over a; the subsequent request $\mathsf{emit}(\{sk(k), \mathsf{rw}\}_{b_r^+})$ made by the left process can now be decrypted by the context, which possesses the decryption key b_r^-, and thus the left reduct $\mathsf{E}_\mathrm{I}[\![\, \Gamma, b : \mathsf{rw} \triangleright b(x@\mathsf{rw}).b\langle x@\mathsf{rw}\rangle \,]\!]$ can be distinguished from the null process $\mathsf{E}_\mathrm{I}[\mathbf{0}]$ (i.e., the process environment $\Gamma, b : \mathsf{rw}$ is not compatible with the context environment I).

6. Enhancing the design – full abstraction

To recover full abstraction, one must shield the client processes from such undesired interactions. That may be achieved by setting up the synchronization protocols so as to ensure that all the exchanges occur over system-generated, hence trusted, channels whose decryption keys remain secret. The new implementation introduces a separation between *client names*, used syntactically by context processes and by translated processes to communicate, and corresponding *server names* generated within the system and associated with system generated channels to be employed in the actual protocols for communication.

A proxy server maintains an association map between client and server names so as to preserve the expected interactions among clients. The map is implemented as a list of entries of the form $(M, \underline{m})$, whose intended invariant is that m is the server counterpart of the client term M (if the client is a compiled process M will be a name identity such as $pub(n)$, but this may not hold for context generated terms). We call M the index of the entry, and $\underline{m}$ the target. The proxy map is set up to ensure that each index has exactly one target. We represent the public keys used by the proxy service by letting $k_P^+ \triangleq ek(k)$ and $k_p^- \triangleq dk(k)$, with k a seed not known to the environment.

The read/write protocols follow the same rationale as in the previous implementation, with the difference that now the clients must first obtain access to the system channel by contacting the proxy server. The interaction between clients and proxy is as follows: the client presents a term to the proxy and the proxy replies with the corresponding server name cast at the type of the term received. In case that term is new, the proxy returns a fresh server name for which it also allocates a system channel. On the client side, the protocol is implemented as shown below:

$$\mathsf{link}\ (M, \underline{y})\ \mathsf{in}\ P \triangleq (\mathsf{new}\ h)\mathsf{emit}(\{sk(h), M\}_{k_p^+}) \,|\, \mathsf{filter}\ \underline{y}\ \mathsf{with}\ sk(h)\ \mathsf{in}\ P$$

For the proxy side, the definition is found in Table 4. There, we write $\mathsf{let}\ \tilde{y} = ?(x, z)\ \mathsf{in}\ P\ \mathsf{else}\ Q$ for the process that extracts the target $\tilde{t}$ associated with x in z and continues as $P\{\tilde{t}/\tilde{y}\}$; if x has no target in z, the process continues as Q.

The only subtlety in the definition of the proxy is that upon receiving a term that does not occur in the association map, the proxy allocates two indexes for the same target: one index is the term received from the client, the other is the public key of the target itself. This second association is needed to make linking idempotent, so that linking a server name always returns the same name. One may wonder how a client could possibly end up requesting a link for a server name, given that server names (i) originate from the proxy, and (ii) are never passed on any exchange by the clients. Notice however, that this invariant is only true of the clients that arise from the translation, not for arbitrary low-level processes of the context.

Having given the intuitions, the definitions in Table 4 should be easily understood. Notice that the clause for restriction is defined homeomorphically in the new implementation, as the creation of the channel is delegated entirely to the proxy server. For the case of matching, the translation would look more uniform had we tested equality on linked names rather than on client names. While this choice has no consequences in the present translation, it does create a problem in the distributed implementation we discuss later on in the paper, as in that case names are known at different proxies, under different linked names (see Section 7).

We may now strengthen the result of Theorem 3 as desired, provided that we plug our processes in the appropriate computing environment. We first define

$$\mathsf{CE}[-] \;=\; - \,|\, W \,|\, Proxy$$

and let the low-level term environments corresponding to the high-level type environment be extended with a new binding expressing the knowledge of the public proxy encryption key k_P^+ needed to interact with the proxy. Then we have:

Theorem 4. *Let* Γ, Δ *and* I *be closed type environment such that* $\Gamma, \Delta <: \mathrm{I}$. *Then:*

$$\mathrm{I} \models P \cong^{\pi} Q \text{ iff } \{\!|\, \mathrm{I} \,|\!\}, k_P^+/y \models \mathsf{CE}[\langle\!\langle\, \Gamma \triangleright P \,\rangle\!\rangle] \cong^{A\pi} \mathsf{CE}[\langle\!\langle\, \Delta \triangleright Q \,\rangle\!\rangle]$$

Proof. (Outline). The "if" direction of the theorem is proved as Theorem 3. The "only if" direction is more elaborate. We first define a (term-indexed) relation $\mathscr{R}$ containing pairs of the form $(C[\langle\!\langle\, \Gamma \triangleright P \,\rangle\!\rangle], C[\langle\!\langle\, \Delta \triangleright Q \,\rangle\!\rangle])$ where C is a context that results from the interaction of any applied-pi process built around the substitution $\{\!|\, \mathrm{I} \,|\!\}, k_P^+/y$, and the computing environment CE. Then we show that $\mathscr{R}$ is term-indexed behavioral equivalence by a case analysis of the possible interactions between the context C and the compiled code $\langle\!\langle\, \Gamma \triangleright P \,\rangle\!\rangle$ and $\langle\!\langle\, \Delta \triangleright Q \,\rangle\!\rangle$. $\Box$

7. A distributed implementation

While the use of the proxy server to protect against misbehaved channels is effective in achieving full abstraction, it is clear that a

Table 4 Fully Abstract Implementation

Proxy server

$$\begin{array}{rcl} P_t & \triangleq & \mathbf{!}\,\mathsf{filter}\ (k,\underline{x},y)\ \mathsf{with}\ k_p^-\ \mathsf{in}\ \mathsf{if}\ y \notin Set_{t^*}\ \mathsf{then}\ \mathsf{let}\ s = Typeof(\underline{x})\ \mathsf{in} \\ & & \quad t(z).\mathsf{let}\ \tilde{y} = ?(\underline{x}_{ID}, z)\ \mathsf{in}\ t\langle z\rangle \mid \mathsf{let}\ \tilde{z} = \mathsf{Cast}(\tilde{y}, s)\ \mathsf{in}\ !net\langle\{\tilde{z}\}_k\rangle \\ & & \quad\quad \mathsf{else}\ (\mathsf{new}\ n)\mathsf{Chan}_n \mid\ t\langle z.(\underline{x}_{ID};\underline{n}).(n_{ID};\underline{n})\rangle\ \mid \mathsf{let}\ \tilde{z} = \mathsf{Cast}(\underline{n}, s)\ \mathsf{in}\ !net\langle\{\tilde{z}\}_k\rangle \\ Proxy & \triangleq & (\mathsf{new}\ t, t^*)\ P_t \mid t\langle\emptyset\rangle \mid t^*\langle\emptyset\rangle \end{array}$$

Channels – same as in Table 3

Clients – The clauses for composition and replication are defined homeomorphically

$$\begin{array}{rcl} \langle\!\langle \Gamma \triangleright u\langle v@T\rangle \rangle\!\rangle_\mu & \triangleq & \mathsf{let}\ \hat{\underline{u}} = [\![\, u \,]\!]_\mu,\ \hat{\underline{v}} = [\![\, v \,]\!]_\mu\ \mathsf{in}\ \mathsf{link}\ (\hat{\underline{u}}{\Uparrow}\mathsf{w}, \underline{x})\ \mathsf{in}\ \mathsf{emit}(\{\hat{\underline{v}}{\Uparrow}\underline{T}\}_{\underline{x}_\mathsf{w}^+}) \\ \langle\!\langle \Gamma \triangleright u(x@T).P \rangle\!\rangle_\mu & \triangleq & (\mathsf{let}\ \hat{\underline{u}} = [\![\, u \,]\!]_\mu\ \mathsf{in}\ \mathsf{link}\ (\hat{\underline{u}}{\Uparrow}\mathsf{r}, \underline{y})\ \mathsf{in}\ (\mathsf{new}\ k)\ \mathsf{emit}(\{sk(k), \underline{T}\}_{\underline{y}_\mathsf{r}^+})) \mid \mathsf{filter}\ \underline{x}\ \mathsf{with}\ sk(k)\ \mathsf{in}\ \langle\!\langle \Gamma, x : T \triangleright P \rangle\!\rangle_\mu \\ \langle\!\langle \Gamma \triangleright [u = v]\, P; Q \rangle\!\rangle_\mu & \triangleq & \text{same as in Table 3 with } [\![\, \cdot \,]\!] \text{ replaced by } \langle\!\langle \cdot \rangle\!\rangle. \\ \langle\!\langle \Gamma \triangleright (\mathsf{new}\ n : A)P \rangle\!\rangle_\mu & \triangleq & (\mathsf{new}\ n)\langle\!\langle \Gamma, n : A \triangleright P \rangle\!\rangle_\mu \end{array}$$

Top−level translation $\langle\!\langle \Gamma \triangleright P \rangle\!\rangle \triangleq \langle\!\langle \Gamma \triangleright P \rangle\!\rangle_\emptyset$

centralized implementation as the one we just described is hardly realistic. In this section, we discuss a new implementation that distributes the proxy services among different servers. The new solution is based on the idea of partitioning the network in domains each of which administrated by a proxy server.

To model this partitioning of the network, we extend our high level calculus with the syntactic category of nets, representing compositions of processes labelled by domains labels, and defined by the following productions:

$$S, T \ ::= \ \delta\{P\} \ \mid \ S \,|\, T \ \mid \ (\mathsf{new}\ n)S \ \mid \ \mathbf{stop}$$

Domain labels, ranged over by δ are drawn from a denumerable set disjoint from the set of names and the set of variables. We let $fd(S)$ be the set of domain labels in S. We emphasize that domain labels are not names, and are never exchanged over channels, nor are they created dynamically by a restriction. The typing and dynamics of nets arise in the simples possible way from the corresponding notions defined for processes. The two core rules are as follows:

$$\frac{\Gamma \vdash P}{\Gamma \vdash \delta\{P\}}\ (\textsc{Typing}) \qquad \frac{P \xrightarrow{\alpha} Q}{\delta\{P\} \xrightarrow{\alpha} \delta\{Q\}}\ (\textsc{Dynamics})$$

As a result, domains have no impact on the dynamics and/or the typing of the high-level calculus: indeed, they serve a different purpose, namely to help devise the association of processes to proxies in the implementation. Notice, in particular, that the same (channel) name may be known at different domains: in the implementation, this will correspond to the name being represented by different channels, located at the different domains at which the name is known. In the syntax of nets we tacitly assume that each domain label occurs at most once. This involves no loss of generality, as the dynamics of the net $\delta\{P_1\} \,|\, \delta\{P_2\}$ is just the same as that of the net $\delta\{P_1 \mid P_2\}$. Similarly, $(\mathsf{new}\ n)\delta\{P\}$ is the same as $\delta\{(\mathsf{new}\ n)P\}$.

Observational Equivalence The definition of observational equivalence for nets is inherited from that of processes. There is an important difference, however, in the notion of contextuality, in that a context may not include new domains, but only processes belonging to existing domains. This is ensured by the side condition $fd(U) \subseteq fd(S,T)$ in the definition below, and constitutes the key assumption for our distributed implementation: namely, we do not trust domains and proxy servers generated by the environment. While this is a somewhat strong assumption, on the other hand it appears to be realistic: notice, in fact, that the procedure of adding a domain to a network in real-world scenarios requires physical authentication, rather than network protocols.

Definition 8 (Contextuality for Nets). *A type-indexed relation $\mathscr{R}$ over nets is contextual whenever*

- $\mathrm{I} \vDash S\mathscr{R}T$ *and* $\mathrm{I} \vdash U$ *and* $fd(U) \subseteq fd(S,T)$ *implies* $\mathrm{I} \vDash (S \,|\, U)\mathscr{R}(T \,|\, U)$
- $\mathrm{I} \vDash S\mathscr{R}T$ *implies* $\mathrm{I}, a : A \vDash S\mathscr{R}T$
- $\mathrm{I}, a : A \vDash S\mathscr{R}T$ *implies* $\mathrm{I} \vDash ((\mathsf{new}\ a : A)S)\mathscr{R}((\mathsf{new}\ a : A)T)$

The definition of behavioral equivalence for nets, noted again $\cong^\pi$, arises now as expected, as the largest type-indexed equivalence relation which is contextual (in the sense above), barb preserving and reduction closed.

The new implementation Each domain of a high-level net corresponds to a domain manager, which manages the channels it has created and acts as a proxy for the processes of the domain. The processes of a domain, in turn, are instructed to send their requests to the proxy associated with their domain. Since different proxies may have different entries for the same client name (remember that a name is possibly known in more domains), more channels servers may correspond to the same pi-calculus name. The domain managers must therefore mask the presence of multiple queues located at the distributed channels associated to the same client name: that, in turn, is based on a further service provided by domain managers to to gain access to fellow proxies.

We represent the public keys used to support the proxy service in each domain manager by means of a one-way constructor p, and define $\delta_p^+ \triangleq ek(p(\delta))$ and $\delta_p^- \triangleq dk(p(n))$; these keys corresponds to the keys k_P^+, k_P^- utilized in the centralized translation. Further, we use an one-way constructor q to represent the public keys used by the queue service for a domain δ: $\delta_q^+ \triangleq ek(q(\delta))$ and $\delta_q^- \triangleq dk(q(\delta))$ We assume that domain managers are connected via secure links, represented by a shared key $sk(k_D)$ generated from the private name k_D. Finally, we introduce the following two bits of new notation:

Table 5 Distributed Translation

Proxy Service

$$P^{\delta}_{q,t} \triangleq \; !\,\mathsf{filter}\ (k, \underline{x}, y)\ \mathsf{with}\ \delta_p^-\ \mathsf{in\ if}\ y \notin Set_{t^*}\ \mathsf{then\ let}\ s = Typeof(\underline{x})\ \mathsf{in}$$
$$t(z).\mathsf{let}\ \tilde{y} = ?(\underline{x}_{ID}, y)\ \mathsf{in}\ t\langle z\rangle \mid \mathsf{let}\ \tilde{z} = \mathsf{Cast}(\tilde{y}, s)\ \mathsf{in}\ !net\langle\{\tilde{z}\}_k\rangle$$
$$\mathsf{else}\ (\mathsf{new}\ n)\mathsf{Chan}_n[q(X).(q\langle X\rangle \mid \Pi_{k_\delta \in X}\ !\,n^\circ(\underline{w}).\mathsf{emit}(\{\underline{x}_{ID}, \underline{w}\}_{k_\delta}))]$$
$$\mid\ t\langle z.(\underline{x}_{ID}; \underline{n}).(n_{ID}; \underline{n})\rangle\ \mid \mathsf{let}\ \tilde{z} = \mathsf{Cast}(\underline{n}, s)\ \mathsf{in}\ !net\langle\{\tilde{z}\}_k\rangle$$

Queue Service

$$Q^{\delta}_{q,t} \triangleq \; !\,\mathsf{filter}\ (x, \underline{s}, y)\ \mathsf{with}\ \delta_q^-\ \mathsf{in\ if}\ y \notin Set_{q^*}\ \mathsf{then}\ t(z).(\,t\langle z\rangle \mid$$
$$\mathsf{let}\ \tilde{y} = ?(x, z)\ \mathsf{in}\ \mathsf{emit}(\{\underline{s}\}_{\underline{y}^+_w})\ \mathsf{else}$$
$$q(X).(q\langle X\rangle \mid \mathsf{Choose}_{k_\delta \in X}\ \mathsf{in}\ \mathsf{emit}(\{x, \underline{s}\}_{k_\delta})\,)$$

Domain Service

$$D^{\delta}_{q,l} \triangleq \; \mathsf{emit}(\{\delta_q^+\}_{sk(k_D)}) \mid !\,\mathsf{filter}\ (x, c)\ \mathsf{with}\ sk(k_D)\ \mathsf{in\ if}\ c \notin Set_{l^*}\ \mathsf{then}\ q(y).q\langle y :: x\rangle$$

Domain Manager

$$M^{\delta} \triangleq \; (\mathsf{new}\ t, t^*, q, q^*, l^*)\ P^{\delta}_{q,t} \mid Q^{\delta}_{q,t} \mid D^{\delta}_{q,l} \mid t\langle\emptyset\rangle \mid t^*\langle\emptyset\rangle \mid q\langle\emptyset\rangle \mid q^*\langle\emptyset\rangle \mid l^*\langle\emptyset\rangle$$

Translation of nets – the clauses for parallel composition and restriction are homeomorphic

$$(\!|\,\Gamma \triangleright \delta\{P\}\,|\!)_\mu \;=\; M^{\delta} \mid (\!|\,\Gamma \triangleright P\,|\!)^{\delta}_{\mu}$$

Channels – as in Table 3

Clients – as in Table 4, with all definitions now parametrized on δ.

Top−level translation $(\!|\,\Gamma \triangleright S\,|\!) \triangleq (\!|\,\Gamma \triangleright S\,|\!)_\emptyset$

- $\mathsf{Chan}_n[P]$ indicates the process $(\mathsf{new}\ n^*, n^\circ)\ n^*\langle\emptyset\rangle \mid RS_n \mid WS_n \mid P$, that is Chan_n augmented with a further process P
- we adapt the notation for links is extended as expected, with a parameter corresponding to the assigned proxy:

$$\mathsf{link}^{\delta}\ (M, \underline{y})\ \mathsf{in}\ P \triangleq$$
$$(\mathsf{new}\ k)\mathsf{emit}(\{sk(k), M\}_{\delta^+_p}) \mid \mathsf{filter}\ \underline{y}\ \mathsf{with}\ sk(k)\ \mathsf{in}\ P$$

The new implementation is given in Table 5. For each domain δ, the domain manager consists of three threads $P^{\delta}_{q,t}$, $Q^{\delta}_{q,t}$, $D^{\delta}_{q,l}$ responsible for the proxy, queue and domain services, respectively. The three threads share a channel q collecting the public keys that grant access to fellow proxies. The domain service $D^{\delta}_{q,l}$ is responsible for updating this queue with newly acquired domain identities, and for publishing the public queue key of the proxy associated to the domain δ.

The proxy and the queue service also share the table of binding client and server names stored on the private channel t. The new definition of the proxy service extends the one given in Table 4, by including a collection of forwarders. Each of the forwarders, tries to extract a message from the channel queue n° and sends it off to one of the domain managers known at the time the channel n was created. The queue service $Q^{\delta}_{q,t}$ waits for the packets sent by the forwarders and by other domain managers. The server retrieves the name index and checks whether the entry is associated to a channel. If it is, the message is sent to the queue of the associated channel. Otherwise both the name index and the message are non-deterministically sent to some of the known domain managers. The non-deterministic choice $\mathsf{Choose}_{k\in X}$ in P may be implemented in the standard way, namely in terms of a non-deterministic synchronization on a private name: $(\mathsf{new}\ n)n\langle\rangle \mid \Pi_{k\in X} n().P$.

The translation of nets is compositional, and does not rely on any pre-existing infrastructure for communications, as now the domain managers are dynamically generated within the translation; as in previous implementations we assume the presence of noise on the communication interface. We extend the low-level term environment with the public encryption keys of the proxy managers corresponding to the high-level free domains: $\{\!|\,\delta_1, \ldots, \delta_n\,|\!\} \triangleq \{\delta_{1p}{}^+/x, \ldots, \delta_{np}{}^+/z\}$. We finally obtain:

Theorem 5. *Let* Γ, Δ *and* I *be closed type environment such that* $\Gamma, \Delta <: \mathrm{I}$. *Then:* $\mathrm{I} \models S \cong^{\pi} T$ *if and only if* $\{\!|\,\mathrm{I}\,|\!\}, \{\!|\,fd(S,T)\,|\!\} \models W \mid (\mathsf{new}\ k_D)(\!|\,\Gamma \triangleright S\,|\!) \cong^{A\pi} W \mid (\mathsf{new}\ k_D)(\!|\,\Delta \triangleright T\,|\!)$.

8. Conclusion

We have developed a secure implementation of a typed pi calculus, in which the access to communication channels is regulated by capability types. The implementation draws on a representation of the typed capabilities in the high-level calculus as term capabilities protected by encryption keys only known to the intended receivers. The implementation relies on a proxy service to protect against malformed messages from the environment. This is achieved by generating certified names (and associated channels) to represent the context-generated names within the system. We have also developed a distributed implementation in which the certification service is implemented by a set of distributed proxies. Being fully compositional, the distributed implementation appears to be adequate for open-ended networks. The only limitation in this respect is represented by our current assumption that all proxies that participate in the synchronization protocols be fully trusted. While a certain degree of trust appears necessary to achieve a secure implementation, it would be desirable to have some form of guarantees also in the presence of malicious proxies. Achieving that seems fea-

sible with our implementation by strengthening the protocols that govern the interactions among proxies. We leave this to our plans of future work.

Our translation has several analogies with previous attempts in the literature [2, 12]. In [2], the authors provide a fully abstract implementation of the join calculus into a dialect of the calculus equipped with cryptographic primitives. The located nature of channels in the join calculus makes it possible to rely on a very compact representation in which a communication channel a is associated with a low-level key name a^+, and to protect communications using an asymmetric cryptosystem. In [12] a fully abstract implementation of the pi-calculus without matching into the join calculus is given; like ours, their translation relies both on the presence of proxy pairs of internal and external names, and on relays among the pairs' components. By composing the encodings [12] and [2] one can securely implement the untyped pi-calculus without matching in a join calculus equipped with cryptographic primitives. However, it is not clear how to implement matching in a join calculus extended with a name-matching construct [11] following this approach. In fact, as noted in [19], the ability of test syntactic equality on names invalidates the semantic equalities on names provided by the equators [15] used in [12] to merge internal and external names.

Acknowledgments

The name representation based on self-signed certificates was suggested to us by Cedric Fournet. We would like to thank him for his insightful comments and constructive criticism on a previous draft of the paper.

This work has been partially supported by M.I.U.R. (Italian Ministry of Education, University and Research) under contract n. 2005015785.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages (POPL '01)*, pages 104–115. ACM Press, 2001.

[2] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, April 2002.

[3] Martín Abadi. Protection in programming-language translations. In *Proc. of ICALP '98*, pages 868–883, 1998.

[4] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *J. ACM*, 52(1):102–146, 2005.

[5] Michael Baldamus, Joachim Parrow, and Björn Victor. A fully abstract encoding of the π-calculus with data terms. In *Proc. of ICALP '05*, pages 1202–1213, 2005.

[6] Bruno Blanchet. From Secrecy to Authenticity in Security Protocols. In Manuel Hermenegildo and Germán Puebla, editors, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes on Computer Science*, pages 342–359, Madrid, Spain, September 2002. Springer Verlag.

[7] Bruno Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.

[8] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2001.

[9] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theor. Comput. Sci.*, 195(2):205–226, 1998.

[10] M. Bugliesi and M. Giunti. Typed processes in untyped contexts. In Rocco Nicola and Davide Sangiorgi, editors, *Proc. of TGC 2005, Symposium on Trustworthy Global Computing*, volume 3705 of *Lecture Notes on Computer Science*, pages 19–32. Springer-Verlag, 2005.

[11] Cédric Fournet and Cosimo Laneve. Bisimulations in the join-calculus. *Theor. Comput. Sci.*, 266(1-2):569–603, 2001.

[12] Cédric Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, Palaiseau., November 1998. Also published by INRIA, TU-0556.

[13] Marco Giunti. *Secure Implementation of Typed Channel Abstractions*. PhD thesis, Università di Venezia, 2007. Forthcoming.

[14] M. Hennessy and J. Rathke. Typed behavioral equivalences for processes in the presence of subtyping. *Mathematical Structures in Computer Science*, 14(5):651–684, 2003.

[15] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theor. Comput. Sci.*, 151(2):437–486, 1995.

[16] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. *Inf. Comput.*, 163(1):1–59, 2000.

[17] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.

[18] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *J. ACM*, 47(3):531–584, 2000.

[19] D. Sangiorgi and D. Walker. *The π-calculus A theory of mobile processes*. Cambridge, 2001.