Computational Secrecy by Typing for the Pi Calculus

Martín Abadi^{1,2}, Ricardo Corin^{1,3}, and Cédric Fournet^{1}

^{1} Microsoft Research
^{2} University of California, Santa Cruz
^{3} University of Twente

Abstract. We define and study a distributed cryptographic implementation for an asynchronous pi calculus. At the source level, we adapt simple type systems designed for establishing formal secrecy properties. We show that those secrecy properties have counterparts in the implementation, not formally but at the level of bitstrings, and with respect to probabilistic polynomial-time active adversaries. We rely on compilation to a typed intermediate language with a fixed scheduling strategy. While we exploit interesting, previous theorems for that intermediate language, our result appears to be the first computational soundness theorem for a standard process calculus with mobile channels.

1 Introduction

In security, both attacks and defenses can operate at various levels of abstraction. For a distributed program, reasoning about security can be in terms of programming-language constructs and concepts, or in terms of their implementations. When those implementations use cryptography, the cryptographic primitives may be represented as black boxes, as specific functions on bitstrings, or even as computing processes with timing and power-consumption characteristics that an attacker may attempt to exploit. While programming abstractions for security can be helpful, they should ideally be mapped to concrete implementations that resist realistic low-level attacks.

In the last decade, a substantial research effort has started to address this problem (e.g., [1,5,7,9,11–13,17,19]). In this paper, we contribute to this line of work by investigating an implementation of a concurrent language with message passing and channel mobility. We treat cryptography both formally (in terms of symbolic expressions) and computationally (at the level of bitstrings, with resource-bounded adversaries).

Specifically, we define and study a distributed cryptographic implementation for an asynchronous pi calculus. At the source level, we adapt simple type systems designed for establishing formal secrecy properties. In particular, we rely on secrecy types for asymmetric communication, in the style of the local pi calculus [3, 18], and on the name-confinement guarantees implied by putting names into scoped groups [14]. We show that those secrecy properties have strong computational counterparts in the implementation, with respect to probabilistic polynomial-time active adversaries that operate on concrete bitstrings.

The implementation leverages Laud's recent results [17] on secrecy by typing in the context of a simulatable cryptographic library [9, 11, 12]. Laud has defined a restricted variant of the spi calculus [6] with a fixed scheduling strategy and without channel mobility (so with fixed, global communication ports). We use Laud's calculus as an intermediate language: we translate the pi calculus to his calculus, then rely on his use of the simulatable cryptographic library. Laud employs a type system for secrecy and proves its soundness with respect to the cryptographic library. We show that our translation is type-preserving. Then, via Laud's results, we obtain computational secrecy guarantees, as a soundness theorem for our pi calculus typings.

Related Work The comparison of formal and computational cryptography is an active research field (e.g., [7, 11, 17, 19]); it has produced computational justifications for formal models of cryptographic operations and for classes of protocols that use formal cryptography. At a higher level, we have implementations of process calculi in terms of black-box, formal cryptography (e.g., [1, 4, 5]). It might be tempting to try to compose the results from those two efforts. For instance, one might imagine a translation from the pi calculus to Turing machines via the spi calculus. Unfortunately, this strategy is not viable at present, and may never be. First, compiling the pi calculus to the spi calculus while preserving security guarantees is difficult at best [1]. In addition, we lack a full computational interpretation for the pi or the spi calculus; in particular, the pi calculus features non-determinism and non-termination, which seem at odds with probabilistic polynomial-time computation. Type systems do help, as does a certain realism in setting goals—for instance, aiming to preserve only secrecy properties, and not necessarily all testing equivalences. Alternatively, one may alter the pi calculus to reflect implementation constraints; Adão and Fournet [8] thus designed a calculus with mobile names (but not mobile channels) and ad hoc communications primitives, and established the computational soundness of its implementation for observational equivalence. Other works also develop implementations of abstract security functions. In particular, Canetti and Krawczyk have considered the problem of implementing secure channels [13], without however a language framework.

Our main result appears to be the first computational soundness theorem for a standard process calculus with mobile channels. In fact, the literature does not seem to contain even a computational soundness theorem for CCS. Going beyond CCS, the main difficulties that we address pertain to channel scopes and mobility, which are central to the pi calculus. Secrecy by typing can be regarded as a discipline for that mobility.

Contents Section 2 defines our source language. Section 3 presents a local type system. Section 4 explains the intermediate language. Section 5 describes a distributed implementation of the asynchronous pi calculus. Section 6 presents the computational secrecy result. Section 7 considers the addition of name groups. Section 8 concludes.

2 The Source Language

This section introduces our source process calculus, by giving its syntax and semantics. It also discusses secrecy, informally.

The syntax of the calculus appears in Figure 1. It assumes an infinite set of names and an infinite set of variables; a, b, c, k, s, and similar identifiers range over names, and x, y, and z range over variables. The syntax distinguishes a category of terms (data)

$M, N ::=$	terms	$P, Q ::=$	processes
x, y, z	variable	$\overline{M}\langle M_1, \ldots, M_n \rangle$	output
a, b, c, k, s	name	$M(x_1, \ldots, x_n).P$	input
		$!M(x_1, \ldots, x_n).P$	replicated input
		0	nil
		$P \mid Q$	parallel composition
		$(\nu a)P$	restriction
		$\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$	conditional

Fig. 1. Syntax of the process calculus

and processes (programs). The terms are variables and names. The processes include constructs for communication, concurrency, and dynamic name creation, roughly those of the pi calculus, and a conditional. The calculus is polyadic, in the sense that messages are tuples of terms, and asynchronous, in the sense that the output construct does not have a built-in acknowledgment. Inputs may be replicated by prefixing a "!". We write $!^{=}M(x_1, \ldots, x_n)$ when the replication is optional. As usual, we may omit an "*else*" clause when it consists of the nil process 0. The name a is bound in $(\nu a)P$. The variables $x_1, \ldots, x_n$ are bound in P in the process $M(x_1, \ldots, x_n).P$. We write $\mathit{fn}(P)$ for the set of names free in P. A process is closed if it has no free variables; it may have free names. We identify processes up to renaming of bound names and variables.

The semantics of our calculus is defined as usual for the asynchronous pi calculus. We write $P \rightarrow Q$ when P reduces to Q in a single reduction step. We write $P \equiv Q$ when P and Q are structurally equivalent. We also let $\approx$ represent weak observational congruence. These relations are defined only on closed processes; their definitions appear in the full version of this paper.

Concepts of Secrecy In this formal setting, there are two different definitions of secrecy. (See [2] for some discussion and references.) According to the first definition, a process P preserves the secrecy of a piece of data M if P never publishes M, or anything that would permit the computation of M, even in interaction with an attacker. This kind of secrecy guarantee is common in the analysis of security protocols. It is particularly adequate and effective for dealing with the secrecy of fresh values that can be viewed as atomic, such as keys and nonces. Cardelli, Ghelli, and Gordon, and also Abadi and Blanchet, use versions of this definition in their work on secrecy by typing [3, 14]. Even though both Laud's type system and ours draw on those works, our computational results correspond to a stronger definition of secrecy. According to this second definition, a process $P(x)$ preserves the secrecy of the value of a variable x if an adversary cannot distinguish $P(M)$ from $P(N)$ for every M and N. This definition has the advantage of excluding partial or implicit flows of information.

3 A Local Type System for the Source Language

In this section we give a first type system for the source language. This type system enforces asymmetric communication in the sense of the local pi calculus [18].

Our type system is based upon that of Abadi and Blanchet [3], as is Laud's (so this section is partly a review, borrowing from previous papers). More precisely, we adapt a fragment of the original type system which excludes cryptography. In order to match Laud's intermediate type system, we also modify the subtyping relation, and restrict the typing rule for conditionals. Our types are defined by the grammar:

$$T ::= \mathrm{D}^{\mathrm{Secret}} \mid \mathrm{C}^{\mathrm{Secret}}[T_1,\dots,T_n] \mid \mathrm{C}^{\mathrm{Public}}[T_1,\dots,T_n] \mid \mathrm{Public}$$

Type $\mathrm{D}^{\mathrm{Secret}}$ is used for data intended to be kept secret, like message payloads of a protocol; $\mathrm{C}^{\mathrm{Secret}}[T_1,\dots,T_n]$ is the type of a channel on which the adversary cannot communicate, and which carries n-tuples with components of types $T_1,\dots,T_n$. On the other hand, $\mathrm{C}^{\mathrm{Public}}[T_1,\dots,T_n]$ is the type of a channel on which the adversary may send (but not receive) messages; the channel may be intended to carry n-tuples with components of types $T_1,\dots,T_n$, but the adversary may send any data it has on the channel. Finally, Public is the type of all public data. The subtyping relation is the least reflexive relation such that $\mathrm{C}^{\mathrm{Public}}[T_1,\dots,T_n] \leq \mathrm{Public}$.

The rules of the type system concern four judgments:

- $E \vdash \diamond$ means that E is a well-formed environment.
- $E \vdash M : T$ means that M is a term of type T in environment E.
- $E \vdash_\diamond M : S$ means that S is the set of possible "true" types of M in environment E.
- $E \vdash P$ says that the process P is well-typed in environment E.

The rules are as follows. The metavariable u ranges over both names and variables.

Well-formed environment: $\dfrac{}{\emptyset \vdash \diamond}$ $\qquad \dfrac{E \vdash \diamond \qquad u \notin dom(E)}{E, u : T \vdash \diamond}$

Terms: $\dfrac{E \vdash \diamond \qquad (u : T) \in E}{E \vdash u : T}$ $\qquad \dfrac{E \vdash M : T \qquad T \leq T'}{E \vdash M : T'}$

Sets of types of terms: $\dfrac{E \vdash \diamond \qquad (x : T) \in E}{E \vdash_\diamond x : \{T' \mid T' \leq T\}}$ $\qquad \dfrac{E \vdash \diamond \qquad (a : T) \in E}{E \vdash_\diamond a : \{T\}}$

Processes:

$$\frac{E \vdash M : \mathrm{Public} \qquad \forall i \in \{1,\dots,n\}, E \vdash M_i : \mathrm{Public}}{E \vdash \overline{M}\langle M_1,\dots,M_n\rangle} \text{(Output Public)}$$

$$\frac{E \vdash M : \mathrm{C}^L[T_1,\dots,T_n] \qquad \forall i \in \{1,\dots,n\}, E \vdash M_i : T_i}{E \vdash \overline{M}\langle M_1,\dots,M_n\rangle} \text{(Output } \mathrm{C}^L\text{)}$$

$$\frac{(a : \mathrm{Public}) \in E \qquad E, x_1 : \mathrm{Public},\dots,x_n : \mathrm{Public} \vdash P}{E \vdash !^{=}a(x_1,\dots,x_n).P} \text{(Input Public)}$$

$$\frac{\begin{array}{c}(a : \mathrm{C}^{\mathrm{Public}}[T_1,\dots,T_m]) \in E \qquad E, x_1 : \mathrm{Public},\dots,x_n : \mathrm{Public} \vdash P \\ E, x_1 : T_1,\dots,x_m : T_m \vdash P \text{ if } m = n\end{array}}{E \vdash !^{=}a(x_1,\dots,x_n).P} \text{(Input } \mathrm{C}^{\mathrm{Public}}\text{)}$$

$$\frac{(a : \mathrm{C}^{\mathrm{Secret}}[T_1,\dots,T_n]) \in E \qquad E, x_1 : T_1,\dots,x_n : T_n \vdash P}{E \vdash !^{=}a(x_1,\dots,x_n).P} \text{(Input } \mathrm{C}^{\mathrm{Secret}}\text{)}$$

$$\frac{E \vdash \diamond}{E \vdash 0}\text{(Nil)} \qquad \frac{E \vdash P \qquad E \vdash Q}{E \vdash P \mid Q}\text{(Parallel)} \qquad \frac{E, a : T \vdash P \qquad T \neq \mathrm{D}^{\mathrm{Secret}}}{E \vdash (\nu a)P}\text{(Restriction)}$$

$$\frac{E \vdash_\diamond M : S_1 \qquad E \vdash_\diamond N : S_2 \qquad \mathrm{D}^{\mathrm{Secret}} \notin S_1 \cup S_2 \qquad \text{if } S_1 \cap S_2 \neq \emptyset \text{ then } E \vdash P \qquad E \vdash Q}{E \vdash \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q}$$

(Cond)

The typing rules for output say that any public data can be sent on a public channel, and tuples with the expected types $T_1, \dots, T_n$ can be sent on a channel of type $\mathrm{C}^L[T_1, \dots, T_n]$, for $L \in \{\mathrm{Public}, \mathrm{Secret}\}$. Therefore, by subtyping, any public data can be sent on a channel of type $\mathrm{C}^{\mathrm{Public}}[T_1, \dots, T_n]$. On the other hand, the attacker cannot have channels of type $\mathrm{C}^{\mathrm{Secret}}[T_1, \dots, T_n]$. Therefore, we can guarantee that only tuples with types $T_1, \dots, T_n$ can be sent on such channels. In the rules for input, the channel in question is required to be represented by a name a (not a variable), as in the local pi calculus. We distinguish three cases, considering the type of a.

- If a is of type Public, then the corresponding output must have been typed using (Output Public), so the input values are public. Rule (Input Public) treats this case.
- When a is of type $\mathrm{C}^{\mathrm{Public}}[T_1, \dots, T_m]$, two cases arise. In the first case, the corresponding output has been typed using (Output Public) and subtyping. Then the input values are of type Public. In the second case, the corresponding output has been typed using (Output C^L). In this case, the input values have the expected types $T_1, \dots, T_m$. Rule (Input $\mathrm{C}^{\mathrm{Public}}$) takes into account both cases, by checking that the process P executed after the input is well-typed in both.
- When a is of type $\mathrm{C}^{\mathrm{Secret}}[T_1, \dots, T_n]$, it cannot be known by the attacker, and the corresponding output must have been typed using (Output C^L). The input values are therefore of the expected types $T_1, \dots, T_n$.

Rule (Cond) exploits the idea that if two terms M and N cannot have the same type, then they are certainly different. In this case, the process $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ may be well-typed without P being well-typed. To determine whether M and N may have the same type, we determine the set of possible types of M and N. If M is a variable x, and $(x : T) \in E$, then x may of course have type T. Because of subtyping, when $T = \mathrm{Public}$, x may also be replaced at run-time with a name whose type is a subtype of T. Hence the possible types of x are $\{T' \mid T' \leq T\}$. When M is a name a, its only possible type is the type assigned to it in the environment. Rule (Cond) also has a condition that excludes any comparison of $\mathrm{D}^{\mathrm{Secret}}$ terms. This condition simply rules out any flow of information from $\mathrm{D}^{\mathrm{Secret}}$ values to the control flow of the process, which may be observable by the adversary. Finally, rule (Restriction) excludes the creation of names with type $\mathrm{D}^{\mathrm{Secret}}$ (although not of names with secret-channel types). These two last conditions on rules (Cond) and (Restriction) are not present in the work of Abadi and Blanchet, but they are imposed to meet the requirements of payload secrecy (see Section 4).

An Example We revisit and adapt an example from Abadi and Blanchet that concerns the following protocol in which A sends to B a secret s and B acknowledges it:

Message 1. $A \rightarrow B : k, a$ on b
Message 2. $B \rightarrow A : k, k'$ on a
Message 3. $A \rightarrow B : s$ on k'
Message 4. $B \rightarrow A : \mathit{ack}$ on k

Here, a and b are channels with A and B as only receivers, respectively. Initially, A creates a secret channel k, and sends it along with the return channel a on b. In response, B sends k, as proof of origin, along with a new secret channel k'. Finally, A sends s on k', and B sends ack on k. The goal of this protocol is to guarantee the secrecy of s.

In our calculus, we may represent the principals of this protocol by the processes:

$$A = (\nu k)(\overline{b}\langle k, a\rangle \mid a(x, y).\mathit{if}\ x = k\ \mathit{then}\ (\overline{y}\langle s\rangle) \mid k(z))$$
$$B = b(x, y).(\nu k')(\overline{y}\langle x, k'\rangle \mid k'(z).\overline{x}\langle ack\rangle)$$

As detailed below, we can assign types such that $A \mid B$ typechecks with type $\mathrm{D}^{\mathrm{Secret}}$ for s. According to our main result (Theorem 1), this typing implies the computational secrecy of any value substituted for s. We let

$$\begin{aligned} E = a &: \mathrm{C}^{\mathrm{Public}}[\mathrm{C}^{\mathrm{Secret}}[\mathrm{Public}], \mathrm{C}^{\mathrm{Secret}}[\mathrm{D}^{\mathrm{Secret}}]],\\ b &: \mathrm{C}^{\mathrm{Public}}[\mathrm{C}^{\mathrm{Secret}}[\mathrm{Public}], \mathrm{C}^{\mathrm{Public}}[\mathrm{C}^{\mathrm{Secret}}[\mathrm{Public}], \mathrm{C}^{\mathrm{Secret}}[\mathrm{D}^{\mathrm{Secret}}]]],\\ s &: \mathrm{D}^{\mathrm{Secret}},\ ack : \mathrm{Public} \end{aligned}$$

and obtain $E \vdash A \mid B$ as follows. In the typing of A, we choose $k : \mathrm{C}^{\mathrm{Secret}}[\mathrm{Public}]$. The output $\overline{b}\langle k, a\rangle$ is then typed by rule (Output C^{L}). The input $a(x, y)$ is typed by rule (Input $\mathrm{C}^{\mathrm{Public}}$), and two cases arise:

– $x : \mathrm{Public}, y : \mathrm{Public}$. This case is vacuous by rule (Cond): in the test $x = k$, the two terms do not have common types.
– $x : \mathrm{C}^{\mathrm{Secret}}[\mathrm{Public}], y : \mathrm{C}^{\mathrm{Secret}}[\mathrm{D}^{\mathrm{Secret}}]$. In this case, the output $\overline{y}\langle s\rangle$ is typed by (Output C^{L}). (The condition of (Cond) is fulfilled: $\mathrm{D}^{\mathrm{Secret}} \notin \{\mathrm{C}^{\mathrm{Secret}}[\mathrm{Public}]\}$.) The remaining input $k(z)$ is easily typed by rule (Input $\mathrm{C}^{\mathrm{Secret}}$).

In process B, the input $b(x, y)$ is typed by (Input $\mathrm{C}^{\mathrm{Public}}$), and two similar cases arise.

4 The Intermediate Language

The models of Backes et al. and Laud are concerned with configurations of probabilistic polynomial-time Turing machines. The machines are connected at ports; two ports can be connected by a wire. Some of these machines represent honest parties; others are controlled by the adversary. At any given time, at most one machine is active.

The Idealized Cryptographic Library [9–12] The cryptographic library provides an abstract view of cryptography, in the following sense. Each principal is associated with a deterministic machine P_i; this machine is connected to a concrete instance of the library M_i that runs all cryptographic algorithms on behalf of P_i and maintains a database that maps abstract handles to cryptographic representations. Instead of n concrete library machines M_i, one can connect a single idealized library $\mathsf{TH}_{\mathbf{n}}$, with the same ports, that maps abstract handles to shared, symbolic ("Dolev-Yao") representations. The main results of Backes et al. relate the security of two systems that use, respectively, the concrete and idealized versions of the library, under standard computational cryptographic assumptions. Hence, in order to prove the security of a system that uses the concrete version, it suffices to reason on a system that uses the idealized version.

Laud's Intermediate Language [17] Laud's language can be used for programming each of the machines P_i, using processes that can send and receive messages and abstractly operate on message contents using library calls. Although the language is inspired by the spi calculus, its semantics is significantly different, as it reflects low-level implementation constraints of the cryptographic library. In particular:

- Communications occur on global, static, bidirectional channels, associated with the ports of the underlying machines. Some of these channels are intrinsically secure, but are used solely to code initialization and security specifications.
- The adversary controls the scheduling between machines, and all channels that represent an untrusted network. Hence, it can intercept all network traffic, and even disable the execution of a local process. (In contrast, a pi calculus context can read a replicated output message on a public channel, but cannot prevent other processes from reading it as well; see [8].)
- In other respects, the language is deterministic; in particular, parallel execution within a machine is supported by an interpreter that maintains a run-queue of input processes.
- The control flow of the machines is carefully restricted. When a machine is activated, it reads a single message from one of its input wires, it processes the message and runs for a bounded amount of time, it puts at most one message in one of its output wires, and yields.
- The usage of the library imposes some programming discipline, for instance to exclude encryption cycles [9] or the leakage of private keys.

We use the following grammar for Laud's language, with minor syntactic changes:

$v ::=$	values
x	variable
n	integer constant
$\bot$	failed computation
$e ::=$	expressions
v	value
$\mathsf{gen\_nonce}()$	nonce generation
$\mathsf{gen\_symenc\_key}(i)$	symmetric-key generation
$\mathsf{privenc}(e_k, e_t)$	symmetric-key encryption
$\mathsf{privdec}(e_k, e_t)$	symmetric-key decryption
$\mathsf{keypair}()$	asymmetric-key generation
$\mathsf{pubkey}(e)$	asymmetric encryption key
$\mathsf{pubenc}(e_k, e_t)$	asymmetric-key encryption
$\mathsf{pubdec}(e_k, e_t)$	asymmetric-key decryption
$\mathsf{store}(e)$	value storage
$\mathsf{retrieve}(e)$	value retrieval (by handle)
$\mathsf{list}(e_1, \ldots, e_n)$	list
$\mathsf{list\_proj}(e_i, e)$	projection

$I ::=$	input process
$c(x).Q$	input
$!c(x).Q$	replicated input
$I^* ::=$	sequence of inputs
$I; I^*$	
0	
$Q ::=$	processes
I^*	input
$\overline{c}\langle e\rangle.I^*$	output
$\bot$	run-time failure
$\mathit{let}\ x = e\ \mathit{in}\ Q_1\ \mathit{else}\ Q_2$	let binding
$\mathit{if}\ e = e'\ \mathit{then}\ Q_1\ \mathit{else}\ Q_2$	conditional

Expressions represent calls to the cryptographic library. These calls, when successful, return handles to new entries; otherwise they return $\bot$. Expression $\mathsf{gen\_nonce}()$ creates a fresh nonce. Expression $\mathsf{gen\_symenc\_key}(i)$ generates a symmetric key (where i is a key rank used to prevent cycles; see Section 7). Expression $\mathsf{keypair}()$ generates an asymmetric key pair and returns the private decryption key; $\mathsf{pubkey}(e)$ returns the associated encryption key. Expressions $\mathsf{privenc}(e_k, e_t)$, $\mathsf{privdec}(e_k, e_t)$, $\mathsf{pubenc}(e_k, e_t)$, and $\mathsf{pubdec}(e_k, e_t)$ provide encryptions and decryptions; decryption visibly fails if e_t is not a message encrypted under the key associated with e_k. Expressions $\mathsf{store}(e)$ and $\mathsf{retrieve}(e)$ store and retrieve data, to and from the library, respectively. Expression $\mathsf{list}(e_1, \ldots, e_n)$ constructs a list from n values; $\mathsf{list\_proj}(e_i, e)$ retrieves its ith value.

Input processes I represent passive threads, held in the interpreter run-queue. Processes Q represent threads activated by an input; they perform at most one output, and append input processes I^* to the run-queue. Processes for input, output, and conditional are similar to those of the source calculus. Process $let\ x = e\ in\ Q_1\ else\ Q_2$ evaluates the expression e; if evaluation succeeds, then Q_1 runs with the result value substituted for x; otherwise, Q_2 runs. Process $\perp$ represents run-time failure, written $\mathcal{II}$ for "invalid input" in [17]. Intuitively, $\perp$ causes the current thread to abort, for instance after failing an evaluation or a test: the input process that triggered the thread is put back into the run-queue, and the rejected message is passed to the next input in the run-queue.

Next, we give the syntax for types for the intermediate language, as it is used in this paper. See [17] for further details, including the subtyping relation and the typing rules.

$T ::=$	intermediate types
Public	public data
$\mathrm{SecData}$	secret data
SNonce	secret nonce
$\mathrm{EK}[T]$	asymmetric encryption key
$\mathrm{DK}[T]$	asymmetric decryption key
$\mathsf{list}(T_1, \ldots, T_n)$	list
$\mathrm{SK}^i[T]$	symmetric key
$T_1 + T_2$	sum

Type Public is the type for public data. Its counterpart for secret data is $\mathrm{SecData}$. Type SNonce is the type for secret nonces. Type $\mathsf{list}(T_1, \ldots, T_n)$ is for lists. Types $\mathrm{EK}[T]$ and $\mathrm{DK}[T]$ are the types of public/private asymmetric keys for encrypting values of type T, while $\mathrm{SK}^i[T]$ is the type of symmetric keys of order i for encrypting values of type T. The index i is used for avoiding encryption cycles. Finally, type $T_1 + T_2$ is the sum type of T_1 and T_2. Sum types play a role similar to the double typing of P in rule (Input $\mathrm{C}^{\mathrm{Public}}$) of Section 3.

Secrecy by Typing A concrete configuration $C_n = \langle \mathsf{S}, \mathsf{H}, \mathsf{A} \rangle$ consists of a concrete system S of $(\mathsf{P}_i)_{i=1..n}$ machines connected to their library machines $(\mathsf{M}_i)_{i=1..n}$, along with a user machine H connected to free ports of S, plus an adversary machine A that connects all the remaining unconnected ports. Let $(I_i)_{i=1..n}$ be intermediate-level input processes (hence, consisting of passive threads) used to program the machines P_i of S. Laud's results [17, Theorem 1, Corollary 2] say that if each I_i typechecks in some environment Γ, then C_n preserves secrecy of all data communicated by the user machine H to S. More precisely, Laud shows that in the configuration C_n, the system S preserves *payload secrecy* of all user data, in the sense defined by Backes and Pfitzmann within the simulatable cryptographic library [10]. Basically, a system S preserves payload secrecy if no adversary A, even if colluding with a user machine H, can distinguish an instance of S running with the user inputs provided by H from an instance of S where the inputs are converted to random values (and then replaced back), by a "scrambling" machine F that runs between S and H. Hence, the notion of payload secrecy can be regarded as a computational version of the second formal definition of secrecy described in Section 2.

5 A Distributed Implementation of the Source Language

In this section, we translate assemblies of pi calculus processes into intermediate-language input processes. A pi calculus process represents a concurrent system, but does not indicate the distribution of its subprocesses across machines.

For the source process $P = \prod_{i=1..n} P_i$, our implementation distributes the subprocesses P_i across the machines P_i, for each $i = 1..n$.

We first rearrange the source processes P_i into threads. We then give a compositional translation for the threads that run within each machine. Finally, we describe the top-level implementation and its initialization process.

Normal Forms for Source Processes Source threads are processes that perform a series of name creations and tests, then yield a parallel composition of inputs and outputs. We use the following grammar:

$A ::=$	atomic processes	$T ::=$		threads
$M(x_1, \ldots, x_n).T$	input	$(\nu n)T$		restriction
$!M(x_1, \ldots, x_n).T$	replicated input	$\mathit{if}\ M = N\ \mathit{then}\ T\ \mathit{else}\ T'$		conditional
$\overline{M}\langle M_1, \ldots, M_n \rangle$	output	$\prod_{i=1}^{n} A_i$	$(n \geq 0)$	atomic processes

For every source process P, we show that there exists a thread $T \approx P$, obtained from P by repeatedly applying the two rewriting steps below in all process contexts:

$$P \,|\, (\nu n)Q \rightsquigarrow (\nu n)(P \,|\, Q) \qquad \text{after renaming } n \text{ so that } n \notin \mathit{fn}(P) \tag{1}$$

$$P \,|\, \mathit{if}\ M = N\ \mathit{then}\ Q\ \mathit{else}\ Q' \rightsquigarrow \mathit{if}\ M = N\ \mathit{then}\ P \,|\, Q\ \mathit{else}\ P \,|\, Q' \tag{2}$$

Step (1) is a structural equivalence. Step (2) is an observational equivalence in all contexts. Both steps preserve source typing, and the rewriting always terminates. We let $\mathcal{T}(P)$ represent one such thread for P.

Machine Translation The core of our translation maps channel-based communications to runs of a particular cryptographic protocol.

Informally, the machine run-queue contains one input process for every running atomic process of the source process. When a machine is proposed a message, the message is matched against the pending inputs in the run-queue. If the message is accepted by the translation of an input, then the message triggers the translation of a thread, which runs to completion, then returns one acknowledgment message and appends new input processes to the run-queue. If the message is accepted by the translation of an output, then the message simply triggers this pending output.

We translate a term M to a list of two elements: an encryption key and a nonce. We let $M^+ = \mathsf{list\_proj}(M, 1)$ and $M^c = \mathsf{list\_proj}(M, 2)$. We write $\mathit{let}\ x_1, \ldots, x_n = e\ \mathit{in}\ P$ to abbreviate $\mathit{let}\ l = e\ \mathit{in}\ \mathit{let}\ x_1 = \mathsf{list\_proj}(l, 1)\ \mathit{in}\ \ldots\ \mathit{in}\ \mathit{let}\ x_n = \mathsf{list\_proj}(l, n)$ $\mathit{in}\ P\ \mathit{else}\ \bot \ldots\ \mathit{else}\ \bot$ where l does not occur in P.

We translate processes as follows:

$$\begin{aligned}
\llbracket \overline{M}\langle M_1, \ldots, M_n\rangle \rrbracket &= \mathit{cont}(\_).\overline{\mathit{net}}\langle \mathsf{pubenc}(M^+, \mathsf{list}(M^c, M_1, \ldots, M_n))\rangle \\
\llbracket !^{=}a(x_1, \ldots, x_n).P \rrbracket &= !^{=}\mathit{net}(z).\mathit{let}\ a', x_1, \ldots, x_n = \mathsf{pubdec}(a^-, z)\ \mathit{in} \\
&\ \ \mathit{if}\ a' = a^c\ \mathit{then}\ (\overline{\mathit{ack}}\langle \_\rangle.\llbracket P \rrbracket)\ \mathit{else}\ \bot \\
\llbracket 0 \rrbracket &= 0 \\
\llbracket P \mid Q \rrbracket &= \llbracket P \rrbracket; \llbracket Q \rrbracket \\
\llbracket (\nu a)P \rrbracket &= \mathit{let}\ a^- = \mathsf{keypair}()\ \mathit{in} \\
&\ \ (\mathit{let}\ a = \mathsf{list}(\mathsf{pubkey}(a^-), \mathsf{gen\_nonce}())\ \mathit{in}\ \llbracket P \rrbracket\ \mathit{else}\ 0)\ \mathit{else}\ 0 \\
\llbracket \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q \rrbracket &= \mathit{if}\ M = N\ \mathit{then}\ \llbracket P \rrbracket\ \mathit{else}\ \llbracket Q \rrbracket
\end{aligned}$$

We represent every output by an encryption followed by an output on a public channel net, and every input by the corresponding input and decryption. Specifically, we translate a local channel a to an asymmetric key pair (with public key a^+ and private key a^-) and a nonce a^c. The capability to receive on channel a is represented by having a^-, while the capability to send on channel a is represented by having both a^+ and a^c. The nonce a^c is necessary as well as the key a^+ because, under standard cryptographic assumptions, a^+ may be recovered from any message encrypted under a^+.

Every output is guarded by an input on channel $cont$. This guard ensures that our implementation sends one output at a time. Conversely, every successful input is acknowledged by an immediate output on channel ack, so that the environment knows that the message has been delivered and need not be proposed again—as required for functional correctness. (The symbol _ represents a fresh variable or a dummy value.) The translation of inputs is defined only for local channel names—not for variables, as in $x(y).P$; this condition ensures that every input translation is within the static scope of the corresponding decryption key.

Crucially, our implementation does not depend on typing information. In contrast to ordinary types, secrecy types need not be known to the implementor. They express relative secrecy properties that can be used for studying the behaviour of a system in the presence of an adversary, possibly with different typings for different adversaries.

Initialization of the Distributed Computation Initialization deals with the free names of the source processes P_i for $i = 1..n$. We first group these names, as follows. Let $\widetilde{a}_i$ be the free names used for input in P_i. Let $\widetilde{a} = \cup\widetilde{a}_i$, $\widetilde{b}_i = fn(P_i) \setminus \widetilde{a}_i$, and $\widetilde{b} = \cup\widetilde{b}_i \setminus \widetilde{a}$. Informally, the names $\widetilde{b}$ represent data supplied by the attacker or the user.

We require that $\widetilde{a}_i \cap \widetilde{a}_j = \emptyset$ when $i \neq j$, thereby reflecting a requirement of the underlying cryptographic library: asymmetric decryption keys cannot be communicated. It is similar to the locality requirement of the local pi calculus. Otherwise, our typed translation would accommodate the distribution of private encryption keys as well.

Turning our attention to the knowledge of the adversary, we let $\widetilde{a}_{RW}$ represent names controlled by the adversary, such that $\widetilde{a}_{RW} \cap \widetilde{a} = \emptyset$, and let $\widetilde{a}_W \subseteq \widetilde{a}$ represent names made available to the adversary for output. We finally let $\widetilde{s}$ be $\widetilde{b} \setminus \widetilde{a}_{RW}$. These names represent user secrets.

We are basically interested in source processes that behave like $(\nu\tilde{a})(\overline{export}\langle\widetilde{a}_W\rangle \mid import(\widetilde{a}_{RW}).(P_1 \mid \ldots \mid P_n)\{\widetilde{M}/\widetilde{s}\})$, where $\widetilde{M}$ are the secrets substituted for $\widetilde{s}$. In order to obtain a distributed program in the intermediate language, we use an additional machine P_0 for initialization. In particular, P_0 distributes the cryptographic materials associated with top-level restricted channels, using low-level secure communications.

We introduce syntactic sugar for polyadic communication in the intermediate language: we let $c(\widetilde{x}).P$ abbreviate $c(z).\mathit{let}\ \widetilde{x} = z\ \mathit{in}\ P$ and $\overline{c}\langle\widetilde{e}\rangle.P$ abbreviate $\overline{c}\langle\mathsf{list}(\widetilde{e})\rangle.P$. We arrive at the following definition for the intermediate-level input processes I_0, I_1, …, I_n initially hosted by the machines P_0, P_1, …, P_n:

$$\begin{aligned} I_0 &= (\mathit{export}_i(\widetilde{a}_i))_{i=1..n}.\overline{\mathit{export}}\langle\widetilde{a}_W\rangle.\mathit{import}(\widetilde{a}_{RW}).\mathit{user}(\widetilde{s}).(\mathit{cont}(\_).\overline{\mathit{import}_i}\langle\widetilde{b}_i\rangle)_{i=1..n} \\ I_i &= \mathit{cont}(\_).[\![(\nu\widetilde{a}_i[\_])]\!]\left[\overline{\mathit{export}_i}\langle\widetilde{a}_i\rangle.\mathit{import}_i(\widetilde{b}_i).[\![\mathcal{T}(P_i)]\!]\right] \qquad \text{for } i = 1..n \end{aligned}$$

where export_i and import_i are low-level secure channels between P_0 and P_i, user is a low-level secure channel from the user H to P_0, export and import are low-level channels between P_0 and the adversary A, the context $[\![(\nu\widetilde{a}_i)[\_]]\!][\_]$ is the translation of the source context that binds the names $\widetilde{a}_i$, and $(\_)_{i=1..n}$ abbreviates a sequence of actions for $i = 1, \ldots, n$.

(Considering that initialization is part of the specification, rather than the implementation itself, we rely on low-level secure channels. We could perform most of the initialization on net, but we would still rely on some initial key distribution.)

In summary, our concrete distributed configuration $C_n = \langle\mathsf{S}, \mathsf{H}, \mathsf{A}\rangle$ consists of a system S of $n+1$ machines P_i that each runs the intermediate-language processes I_i defined above plus $n+1$ library machines M_i that realize the cryptographic primitives, along with a user machine H and an adversary machine A.

Discussion Our definition of the processes I_i for $i = 1..n$ does not depend on the origin of the imported values $\widetilde{b}_i$. In other words, the implementation does not know a priori which values are controlled by the adversary. This origin is determined instead in the definition of I_0, by the multiplexing between values that come either from peer machines or from the adversary.

For simplicity, our implementation assumes that all communications are distributed—even if I_i includes matching inputs and outputs. We could also support (and typecheck) a sort of channels for machine-local communications, with an optimized implementation that does not rely on cryptography.

Our implementation is not meant to resist all attacks. Indeed, the adversary can affect the control flow of the program, for instance by replaying messages. Consider for example the source process $P = (\nu a)(\overline{a}\langle\rangle \mid a().a().\overline{adv}\langle s\rangle)$. According to the pi calculus semantics, P preserves the secrecy of s from a context that knows adv—in fact P behaves just like the inert process 0. With our implementation, the secrecy of s is broken if the adversary has the decryption key for adv: the adversary observes an opaque message on net (produced by evaluating $\mathsf{pubenc}(a^+, \mathsf{list}(a^c))$) and it can forward that message twice to the machine that hosts the inputs on a, causing that machine to send back $\mathsf{pubenc}(adv^+, \mathsf{list}(adv^c, s))$, and eventually the adversary can extract s. Note, however, that the rules of Section 3 safely exclude any typing $E \vdash P$ that contains both $s : \mathrm{D}^{\mathrm{Secret}}$ and $adv : \mathrm{Public}$.

Functional Correctness Although we are mainly interested in secrecy, it is also important to check that our implementation actually works. We therefore establish that our implementation is functional for one particular definition of the adversary that implements a reliable network.

To this end, we briefly recall the main notations used by Laud in the deterministic operational semantics of the intermediate language. Let $\mathsf{P}_i[Q]$ represent the passive state of a local machine that implements the series of input processes Q, along with the state of the idealized cryptographic library. We write $(\mathsf{P}_i[Q], \alpha) \longrightarrow\!\!\!\!\rightarrow (\mathsf{P}'_i[Q'], \beta_\perp)$ for a series of computation steps from state $\mathsf{P}_i[Q]$ to state $\mathsf{P}'_i[Q']$. The message α represents an encoded input from the adversary. The outcome $\beta_\perp$ represents either an encoded output or $\perp$, which indicates either that the input was not accepted or that the input was accepted with no response. We omit the definition of encoded inputs and outputs, and simply write $(\!(A)\!)$ for an encoded message produced by P_i to send the source output A.

We state operational correspondences for inputs and outputs as follows. We let T, T', T'' range over parallel compositions of source inputs and outputs, and let A range over source outputs.

- If $T \mid A \to T'$ then $T \mid A \to P$ and $(\mathsf{P}_i[[\![T]\!]], net(\!(A)\!)) \longrightarrow\!\!\!\!\rightarrow (\mathsf{P}'_i[[\![T'']\!]]), ack)$ for some P and $(\nu\widetilde{a}')T'' \equiv \mathcal{T}(P)$. Otherwise, $(\mathsf{P}_i[[\![T]\!]], net(\!(A)\!)) \longrightarrow\!\!\!\!\rightarrow (\mathsf{P}'_i[[\![T]\!]], \perp)$.
- If T has an output, then $T \equiv A \mid T''$ and $(\mathsf{P}_i[[\![T]\!]], cont) \longrightarrow\!\!\!\!\rightarrow (\mathsf{P}'_i[[\![T'']\!]], net(\!(A)\!))$ for some A and T''. Otherwise, $(\mathsf{P}_i[[\![T]\!]], cont) \longrightarrow\!\!\!\!\rightarrow (\mathsf{P}'_i[[\![T]\!]], \perp)$.

These correspondences reflect an unknown, deterministic scheduling; they guarantee only that, if some threads in T may input A, then one of their implementations will input A, and similarly for outputs. In the first correspondence, $(\nu\widetilde{a}')$ represents the new restrictions in evaluation context; their translations create new keys recorded in the library, so the source restrictions are discarded in T''.

The proposition below relies on the cooperation of an adversary N that performs initialization, then repeatedly retrieves all pending outputs, stores them in a queue, and repeatedly attempts to deliver the pending outputs to each of the machines in turn. The proposition states that the implementation then follows one of the expected (finite or infinite) source traces.

Proposition 1 (Functional correctness). *Let the machines $(\mathsf{P}_i)_{i=0..n}$ implement the source processes $(P_i)_{i=1..n}$ with initialization parameters $\widetilde{a}, \widetilde{a}_{RW}, \widetilde{a}_W, \widetilde{s}$. Let S be the idealized system $((\mathsf{P}_i)_{i=0..n}, \mathsf{TH}_\mathbf{n})$. Let $P = (\nu\widetilde{a})\prod_{i=1..n} P_i$.*

There exist an adversary N, a user H, and source reductions $P \to^* P' \not\to$ (or $P \to^\ell P'$ for any $\ell \geq 0$) such that $P' \equiv (\nu\widetilde{a}')\prod_{i=1..n} P'_i$ and the configuration $(\mathsf{S}, \mathsf{H}, \mathsf{N})$ reaches a state such that the run-queue of every machine P_i of S contains the input processes $[\![\mathcal{T}(P'_i)]\!]$ for $i = 1..n$.

6 Computational Secrecy by Local Typing

We establish payload secrecy for the distributed implementation of arbitrary source processes. We translate types and type environments, then verify that source type derivations always yield valid type derivations in the intermediate language. The translation of types is as follows:

$$\begin{aligned}
[\![\mathrm{D}^{\mathrm{Secret}}]\!]^t &= \mathrm{SecData} \\
[\![\mathrm{Public}]\!]^t &= \mathrm{Public} \\
[\![\mathrm{C}^{\mathrm{Secret}}[T_1, \ldots, T_n]]\!]^t &= \mathsf{list}(\mathrm{EK}[\mathsf{list}(\mathrm{SNonce}, [\![T_1]\!]^t, \ldots, [\![T_n]\!]^t)], \mathrm{SNonce}) \\
[\![\mathrm{C}^{\mathrm{Public}}[T_1, \ldots, T_n]]\!]^t &= \mathsf{list}(\mathrm{EK}[\mathsf{list}(\mathrm{Public}, [\![T_1]\!]^t, \ldots, [\![T_n]\!]^t)], \mathrm{Public})
\end{aligned}$$

Hence, the translation of channel types follows our choice of communication protocol.

We lift our translation from types to environments. When translating the name binding for a, we bind two variables: a to the translated type, and a^- to the type of the corresponding private decryption key. We translate bindings as follows:

$$\begin{aligned}
[\![x{:}T]\!]^t &= x{:}[\![T]\!]^t \\
[\![a{:}\mathrm{D}^{\mathrm{Secret}}]\!]^t &= a{:}[\![\mathrm{D}^{\mathrm{Secret}}]\!]^t \\
[\![a{:}\mathrm{Public}]\!]^t &= a{:}[\![\mathrm{Public}]\!]^t, a^-{:}[\![\mathrm{Public}]\!]^t \\
[\![a{:}\mathrm{C}^{\mathrm{Secret}}[T_1,\dots,T_n]]\!]^t &= a{:}[\![\mathrm{C}^{\mathrm{Secret}}[T_1,\dots,T_n]]\!]^t, a^-{:}\mathrm{DK}[\mathsf{list}(\mathrm{SNonce}, [\![T_1]\!]^t,\dots,[\![T_n]\!]^t)] \\
[\![a{:}\mathrm{C}^{\mathrm{Public}}[T_1,\dots,T_n]]\!]^t &= a{:}[\![\mathrm{C}^{\mathrm{Public}}[T_1,\dots,T_n]]\!]^t, a^-{:}\mathrm{DK}[\mathsf{list}(\mathrm{Public}, [\![T_1]\!]^t,\dots,[\![T_n]\!]^t)]
\end{aligned}$$

We let Γ_0 be the intermediate-language environment that assigns types to the implementation channels net, ack, $cont$, $export$, $import$, and $export_i$, $import_i$, $user_i$ for $i = 1..n$ in such a way that $\Gamma_0 \vdash I_0$. (The definition of Γ_0 appears in the full version of this paper.) We let $[\![E]\!]^t$ be Γ_0 plus the translations of the bindings in E.

The next lemma states that source subtyping is preserved, and that all type derivations for source terms and processes yield type derivations in the intermediate language.

Lemma 1 (Type preservation).

1. *If $T \leq T'$ then $[\![T]\!]^t \leq [\![T']\!]^t$.*
2. *If $E \vdash M : T$ then $[\![E]\!]^t \vdash M : [\![T]\!]^t$.*
3. *If $E \vdash P$, then $[\![E]\!]^t \vdash [\![P]\!]$.*

We obtain:

Theorem 1. *Let $P = \prod_{i=1..n} P_i$. Let the machines $(\mathsf{P}_i)_{i=0..n}$ implement the source processes $(P_i)_{i=1..n}$ with initialization parameters $\widetilde{a}, \widetilde{a}_{RW}, \widetilde{a}_W, \widetilde{s}$.*

Let E be the source typing environment that contains

– $(s : \mathrm{D}^{\mathrm{Secret}})$ *for each $s \in \widetilde{s}$;*
– $(a : \mathrm{C}^{\mathrm{Secret}}[\widetilde{T}])$ *for each $a \in \widetilde{a} \setminus \widetilde{a}_W$;*
– $(a : \mathrm{C}^{\mathrm{Public}}[\widetilde{T}])$ *for each $a \in \widetilde{a}_W$;*
– $(b : \mathrm{Public})$ *for each $b \in \widetilde{a}_{RW}$.*

If $E \vdash P$, then the concrete system $(\mathsf{P}_i, \mathsf{M}_i)_{i=0..n}$ preserves payload secrecy of $\widetilde{s}$.

We illustrate the use of the theorem on the example of Section 3. We have established that $E \vdash A \,|\, B$. Let S be the system that includes machines $(\mathsf{P}_i)_{i=0,1,2}$ with initialization parameters $\widetilde{a}_1 = \{a\}$, $\widetilde{a}_2 = \{b\}$, $\widetilde{a} = \{a, b\}$, $\widetilde{b}_1 = \{b, s\}$, $\widetilde{b}_2 = \{ack\}$, $\widetilde{b} = \{ack, s\}$, $\widetilde{a}_{RW} = \{ack\}$, $\widetilde{a}_W = \{a, b\}$, and $\widetilde{s} = \{s\}$, such that P_1 hosts the translation of A, P_2 hosts the translation of B, and P_0 runs the initialization process I_0:

$$\begin{aligned}
I_0 = {} & export_1(a).export_2(b).\overline{export}\langle a, b\rangle.import(ack).user(s). \\
& cont(\_).\overline{import_1}\langle b, s\rangle.cont(\_).\overline{import_2}\langle ack\rangle
\end{aligned}$$

Since E meets the conditions of Theorem 1, system S preserves payload secrecy of s.

7 Types for Channel Groups

In this section, we supplement our type system with typing rules adapted from Cardelli et al. [14]. These rules are also designed to ensure formal secrecy by typing, but they concern symmetric communication channels, confined using scoped groups of names. Relying on this confinement discipline, we can implement channels using symmetric encryption, with computational secrecy guarantees.

Group Types in the Source Language Group types embody static scoping policies in the pi calculus; they help control the dynamic extrusion of channels by partitioning them into named groups and statically controlling the scope of these groups. Groups can be dynamically created as part of the computation; they ensure that "channels of group G are forever secret outside the initial scope of (νG)" [14].

We extend the grammars for source processes and types accordingly:

$P, Q ::=$	processes	$T ::=$	types
$\ldots$	(see Section 2)	$\ldots$	(see Section 3)
$\overline{M}\langle M_1, \ldots, M_n \rangle_{\mathsf{s}}$	output	$G[T_1, \ldots, T_n]$	channel in group G
$M(x_1, \ldots, x_n)_{\mathsf{s}}.P$	input		
$!M(x_1, \ldots, x_n)_{\mathsf{s}}.P$	replicated input		
$(\nu G)P$	group restriction		
$(\nu_{\mathsf{s}} a : G[T_1, \ldots, T_n])P$	restriction		

We assume an infinite set of groups and let G, G' range over groups. The process $(\nu G)P$ binds G with scope P. We consider processes up to renaming of bound groups.

The other processes enable communication and restriction on names that belong to a group, much as the processes of Section 2, except for an additional "s" that indicates the usage of group names (so that we can select symmetric-key cryptography in the implementation). Restrictions also mention types, which are useful here for guiding the translation.

Operationally, group restrictions behave like name restrictions, with similar structural-equivalence rules and an additional context rule for reductions: $P \rightarrow P' \Rightarrow (\nu G)P \rightarrow (\nu G)P'$. Hence, group types do not play any dynamic role, and we can retrieve untyped source processes and the untyped semantics by type erasure [14, Section 3].

We supplement our type system with additional typing rules for groups:

$$\frac{E \vdash \diamond \quad G \notin dom(E)}{E, G \vdash \diamond} \qquad \frac{E \vdash \diamond \quad G \in dom(E) \quad u \notin dom(E) \quad E \vdash T_1, \ldots, E \vdash T_n}{E, u : G[T_1, \ldots, T_n] \vdash \diamond}$$

$$\frac{E \vdash M : G[T_1, \ldots, T_n] \qquad E, x_1 : T_1, \ldots, x_n : T_n \vdash P}{E \vdash !^{=} M(x_1, \ldots, x_n)_{\mathsf{s}}.P} \text{ (Input } G\text{)}$$

$$\frac{E \vdash M : G[T_1, \ldots, T_n] \qquad \forall i \in \{1, \ldots, n\}, E \vdash M_i : T_i}{E \vdash \overline{M}\langle M_1, \ldots, M_n \rangle_{\mathsf{s}}} \text{ (Output } G\text{)}$$

$$\frac{E, G \vdash P}{E \vdash (\nu G)P} \text{ (Group Restriction)} \qquad \frac{E, a : G[T_1, \ldots, T_n] \vdash P}{E \vdash (\nu_{\mathsf{s}} a : G[T_1, \ldots, T_n])P} \text{ (Restriction } G\text{)}$$

The well-formedness rules demand that all groups are recorded in E and group types are not mutually recursive. Thus, (Group Restriction) ensures that a restricted G never occurs in the type of a free variable.

The other rules are standard. In contrast with the rules for local channels, (Input G) enables inputs on any term with a group type.

An Example Consider the processes:

$$A = (\nu_{\mathsf{s}} d : G[\mathrm{D}^{\mathrm{Secret}}])(\overline{c}\langle d\rangle_{\mathsf{s}} \mid \overline{d}\langle s_1\rangle_{\mathsf{s}} \mid \overline{d}\langle s_2\rangle_{\mathsf{s}})$$
$$B = !c(z)_{\mathsf{s}}.z(x_1)_{\mathsf{s}}.z(x_2)_{\mathsf{s}}$$

Here, c represents a private, long-term channel between A and B, and d represents a private channel for a session; A creates d in group G, sends it to B on c, and uses d to send secrets s_1 and s_2 to B.

We can assign types so that $(\nu G)(\nu_{\mathsf{s}} c : G[G[\mathrm{D}^{\mathrm{Secret}}]])(A \mid B)$ typechecks, with s_1 and s_2 of type $\mathrm{D}^{\mathrm{Secret}}$.

Two Difficulties with Symmetric Encryption Scoped group types are a good match for symmetric keys, with their limitations. We discuss two such standard limitations in the context of the intermediate language and the idealized cryptographic library.

– *Encryption cycles* may occur when the same symmetric keys are used both as encryption keys and within encrypted values. Cycles are potentially unsafe, and therefore excluded by standard computational definitions of secrecy [7]. In particular, cycles must be excluded in the cryptographic library [9], as follows: every secret symmetric-key encryption has an integer rank, k, and the idealized library checks that, for every encryption, (the symbolic representation of) the value to be encrypted includes only encryptions of a strictly lower rank.
– *Key compromises* may occur during the computation, but they are hard to model computationally. The cryptographic library simplifies the issue by requiring that any symmetric key that may eventually be leaked to the adversary be leaked before any encryption under the key becomes known by the adversary [9]. Laud's type system simplifies further, and excludes any leakage of symmetric keys by typing.

To address the second limitation, we extend the intermediate language, as follows. By convention, we use rank 0 to indicate a key that is (immediately) leaked to the adversary. We refine the rule (SK) for $\mathsf{gen\_symenc\_key}(k)$ [17]:

$$\frac{k \geq 0}{\mathsf{gen\_symenc\_key}(k) : \mathrm{SK}^k[T]} \quad \text{(SK)}$$

into two rules:

$$\frac{}{\mathsf{gen\_symenc\_key}(0) : \mathrm{Public}} \quad \text{(PSK)} \qquad \frac{k > 0}{\mathsf{gen\_symenc\_key}(k) : \mathrm{SK}^k[T]} \quad \text{(SK}')$$

The special typing rule for $k = 0$ is admissible[4]; indeed, Laud's system already supports symmetric keys of type Public received from the adversary.

Moreover, we assume that the library implementation of $\mathsf{gen\_symenc\_key}(k)$ detects $k = 0$ and then leaks the key to the adversary, using some additional port. Technically, we establish payload secrecy result for systems with this modification. However, it is straightforward to show that, if a system preserves payload secrecy while leaking some symmetric keys, then the same system without the leak also preserves payload secrecy.

[4] P. Laud, private communication

(If an adversary breaks payload secrecy for the system without the leak, then the same adversary breaks payload secrecy for the system with the leak—by just ignoring the extra input.) This latter system does not dynamically rely on the rank parameters k.

Distributed Implementation We describe the distributed implementation of source processes with groups as an extension of the implementation of Section 5.

As a global, preliminary step, we partition the free groups of the source processes P_i for $i = 1, \dots, n$ into public and private groups, we rename the restricted groups so that all groups are pairwise distinct, and we give a rank to every type: $\mathsf{rank}(G[T_1, \dots, T_n]) = 1 + \mathsf{max}_{i=1..n}(\mathsf{rank}(T_i))$ when G is private or restricted; all other types have rank 0.

We extend the translations for types and processes as follows:

$$\begin{aligned}
[\![G[T_1, \dots, T_n]]\!]^t &= \begin{cases} \mathrm{Public} & \text{when } G \text{ public} \\ \mathrm{SK}^{\mathsf{rank}(G[T_1,\dots,T_n])}[\mathsf{list}([\![T_1]\!]^t, \dots, [\![T_n]\!]^t)] & \text{otherwise} \end{cases} \\
[\![\overline{M}\langle M_1, \dots, M_n\rangle_{\mathsf{s}}]\!] &= \mathit{cont}(\_).\overline{\mathit{net}}\langle \mathsf{privenc}(M, \mathsf{list}(M_1, \dots, M_n))\rangle \\
[\![!^{=}M(x_1, \dots, x_n)_{\mathsf{s}}.P]\!] &= !^{=}\mathit{net}(z).\mathit{let}\ x_1, \dots, x_n = \mathsf{privdec}(M, z)\ \mathit{in}\ \overline{\mathit{ack}}\langle\_\rangle.[\![P]\!] \\
[\![(\nu G)P]\!] &= [\![P]\!] \\
[\![(\nu_{\mathsf{s}} a : T)P]\!] &= \mathit{let}\ a = \mathsf{gen\_symenc\_key}(\mathsf{rank}(T))\ \mathit{in}\ [\![P]\!]\ \mathit{else}\ 0
\end{aligned}$$

The translation of environment is extended to group-type bindings pointwise, and discards groups. As in Section 6, we show that our translation of processes is well-typed. Initialization applies unchanged: we exchange private-group names in $\widetilde{a} \setminus \widetilde{a}_W$ (just as names of type $\mathrm{C}^{\mathrm{Secret}}[T_1, \dots, T_n]$) and public-group names in $\widetilde{a}_{RW}$.

Finally, we generalize Proposition 1, Lemma 1, and Theorem 1 to systems with both kinds of channel implementations, with the additional requirement that, in the top-level source environment, the types within public-group types be either Public or other public-group types. (We leave details for the full version of this paper.)

8 Conclusion

In summary, we obtain computational secrecy guarantees for an implementation of a standard process calculus with mobile channels. The guarantees apply to processes that conform to typing disciplines originally designed for establishing formal secrecy. It is pleasing that these typing disciplines have a strong, non-trivial computational meaning. One may also be able to extend these results to other secrecy requirements. Further, we expect that analogous results may be established for typing disciplines that enforce authenticity [16] (as already suggested by Laud) and authorization [15]. In addition, implementations such as the one considered in this paper can be hardened against many kinds of attacks, whether or not the corresponding security properties are captured in type systems. Unfortunately, however, some attractive extensions appear challenging. For instance, protection against traffic analysis may require expensive implementation strategies or changes in the source calculus [1, 8]. An interesting direction for further research is the development of high-level models and calculi that would be both convenient for programming and amenable to sound, efficient implementations.

Acknowledgments Abadi's work was partly supported by the National Science Foundation under Grants CCR-0208800 and CCF-0524078. Corin's work was partly supported by an NWO travel grant 02-1979.

References

1. M. Abadi. Protection in programming-language translations. In *25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *LNCS*, pages 868–883. Springer-Verlag, 1998.
2. M. Abadi. Security protocols and their properties. In F. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, NATO Science Series, pages 39–60. IOS Press, 2000.
3. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387–415, Apr. 2003.
4. M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *27th ACM Symposium on Principles of Programming Languages*, pages 302–315, Jan. 2000.
5. M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, Apr. 2002.
6. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
7. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.
8. P. Adão and C. Fournet. Cryptographically sound implementations for communicating processes (extended abstract). In *33rd International Colloquium on Automata, Languages and Programming*, volume 4052 of *LNCS*, pages 83–94. Springer-Verlag, July 2006.
9. M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *17th IEEE Computer Security Foundations Workshop*, pages 204–218, 2004.
10. M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. In *IEEE Symposium on Security and Privacy*, pages 171–182, 2005.
11. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *10th ACM Conference on Computer and Communications Security*, pages 220–230, 2003.
12. M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. *International Journal of Information Security*, 4(3):135–154, 2005.
13. R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Encrocrypt 2001*, volume 2045 of *LNCS*. Springer-Verlag, 2001.
14. L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.
15. C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *14th European Symposium on Programming*, volume 3444 of *LNCS*, pages 141–156. Springer, 2005.
16. A. D. Gordon and A. S. A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *J. Computer Security*, 12(3/4):435–484, 2004.
17. P. Laud. Secrecy types for a simulatable cryptographic library. In *12th ACM Conference on Computer and Communications Security*, pages 26–35, 2005. Also Research Report IT-LU-O-162-050823, Cybernetica, Aug. 2005.
18. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *LNCS*, pages 856–867. Springer-Verlag, 1998.
19. D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *1st Theory of Cryptography Conference (TCC)*, pages 133–151, 2004.