Secrecy Types for Asymmetric Communication

Martín Abadi[1] and Bruno Blanchet[2]

[1] Bell Labs Research, Lucent Technologies,
abadi@research.bell-labs.com
[2] INRIA Rocquencourt[⋆⋆⋆],
Bruno.Blanchet@inria.fr

Abstract We develop a typed process calculus for security protocols in which types convey secrecy properties. We focus on asymmetric communication primitives, especially on public-key encryption. These present special difficulties, partly because they rely on related capabilities (e.g., "public" and "private" keys) with different levels of secrecy and scopes.

1 Introduction

A secret is something you tell to one person at a time, according to a popular definition. Research on security has led to several other concepts of secrecy (e.g., [2, 12, 16]). This paper studies secrecy in the context of security protocols with the methods of process calculi and type systems. Here, a secret is a datum (such as a cryptographic key) that never appears on a channel on which an adversary can listen, even if the adversary actively tries to obtain the datum. We develop a process calculus with a type system in which types convey secrecy properties and such that well-typed programs keep their secrets.

Many security protocols use asymmetric communication primitives, namely communication channels with only one fixed end-point (the receiver) and particularly public-key encryption. These primitives present special difficulties, partly because they rely on pairs of related capabilities (e.g., "public" and "private" keys) with different levels of secrecy and scopes. Therefore, we concentrate on these primitives, and their treatment constitutes the main novelty of this work. (An extended version of this paper gives corresponding type rules for symmetric primitives, such as shared-key encryption; they are more straightforward.)

Basically, our type system is concerned with the question of when it is permissible to tell a secret, and particularly to one person at a time. Telling it to one person at a time means that any communication channel used to transmit the secret has a unique receiver, or, if the secret is sent encrypted, that only one person has the corresponding decryption key. Of course, the person in question should be allowed to know the secret—it may still be a secret with respect to the adversary. Moreover, the person should realize that this is a secret, and treat it accordingly. The type system helps enforce these conditions.

[⋆⋆⋆] This work was partly done while the author was at Bell Labs Research.

The rest of this introduction describes the process calculus that serves as setting for this work, the type system, and some of their properties. It also describes difficulties connected with public-key encryption and other asymmetric communication primitives. Along the way, it mentions relevant related work.

The process calculus. The process calculus is a relative of the pi calculus [21] and the spi calculus [3]. It includes channels with fixed receivers, as in the local pi calculus [19], the join calculus [11], and many object-oriented languages (e.g, [6]). Such a channel can be used for transmitting secrets if the adversary cannot listen on the channel. On the other hand, the capability for sending on the channel may be published. The channel may therefore convey not only secrets but also public data from the adversary. The receiver needs static and dynamic checks for distinguishing these two cases; our type system accounts for some of these mechanisms.

In addition, the process calculus includes cryptographic operations, specifically public-key encryption. In a public-key encryption scheme, the capabilities of encryption and decryption are separate (e.g., [18]), and can be handled differently. Hence public-key encryption is often called asymmetric encryption. Typically, the capability for decryption (the "private" key) remains with one principal, while the capability for encryption (the "public" key) may be published. Therefore, both secrets and public data from the adversary may be encrypted under the public key. Thus, pleasingly, public-key encryption resembles communication on channels with fixed receivers. Our process calculus and type system treat them analogously.

The phrase "public key" has at least two distinct meanings in this context. It may refer to one of the two keys in a public-key cryptosystem, used for encrypting data or for verifying digital signatures (which we do not treat here). Although such a key may be widely known, and it often is in examples, it can also be kept secret. Alternatively, a "public key" may be a key which happens to be public, that is, not secret. We try to use "public key" only with the latter meaning, and prefer "encryption key" for the former.

The type system. The type system is based on old ideas on secrecy levels [10] and on the newer trend of representing these levels in types (e.g., [1, 9, 13–15, 22, 23, 26, 28]). For example, Public and Secret are the types of public data and secret data, respectively. In addition, the type system gives information on the intended usage and structure of data, like standard type systems. For example, $\mathrm{K}^{\mathrm{Secret}}[T_1, T_2]$ is the type of a secret encryption key that is used to encrypt pairs with components of respective types T_1 and T_2. Similarly, $\mathrm{K}^{\mathrm{Public}}[T_1, T_2]$ is the type of a public encryption key. Analogous types apply to channels.

Substantial difficulties arise because, in the study of security protocols, we cannot assume that the adversary politely obeys our typing discipline [1, 9, 27]. Although honest protocol participants may use public channels and public keys according to declared types, the adversary may be an untyped process and disregard those types. Nevertheless, the declared types remain useful when combined with the static and dynamic checks mentioned above.

In this respect, asymmetric communication primitives (channels with fixed receivers, public-key encryption) are more delicate and interesting than their symmetric counterparts (shared channels as in the pi calculus, shared-key encryption). A shared channel or shared key that the adversary knows should never be employed for transmitting secrets, only public data. Therefore, types like $\mathrm{K}^{\mathrm{Public}}[\mathrm{Secret}]$ are useless in this setting; a public shared key may simply be given the type Public. On the other hand, a channel with a fixed receiver may be employed for transmitting secrets, even if the adversary knows it (as long as the adversary is not the receiver). Similarly, an encryption key in a public-key cryptosystem may be employed for encrypting secrets, even if the adversary knows it (as long as the adversary does not know the corresponding decryption key). Therefore, types like $\mathrm{K}^{\mathrm{Public}}[\mathrm{Secret}]$ give useful information. For instance, although the adversary may encrypt public data under a key of type $\mathrm{K}^{\mathrm{Public}}[\mathrm{Secret}]$, this type can tell others that encrypting secrets under the key is acceptable too, that these secrets will not escape. The typing of asymmetric communication primitives in the context of an untyped adversary is the main novelty of this work.

Secrecy results. We prove a subject-reduction property, namely, that typing is preserved by computation. Relying on this property, we also prove a secrecy theorem that shows that well-typed processes do not reveal their secrets. As an example, let us consider a well-typed process P with just two free names, a of type Public and k of type $\mathrm{K}^{\mathrm{Secret}}[T_1, T_2]$. As an adversary, we allow any process Q with the free name a (and possibly other free names, except for k). The secrecy theorem implies that the parallel composition of P with an adversary Q never outputs k on a.

This kind of secrecy guarantee is common and useful 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. In contrast, secrecy guarantees based on the concept of noninterference are better at excluding flows of partial information about compound values and implicit information flows. (See [2, section 6] for some further discussion and references.) Most type systems for secrecy concern noninterference; a recent exception is that of Cardelli, Ghelli, and Gordon [9].

Application to protocols. Our type system can be applied to some small but subtle security protocols. For example, in the Needham-Schroeder public-key protocol [24] (a standard test case), one might expect a certain nonce to be secret; however, the protocol fails to typecheck under the assumption that this nonce is secret. This failure is not a shortcoming of the type system: it manifests Lowe's attack on the protocol [17]. On the other hand, a corrected version of the protocol does typecheck under the assumption. Our secrecy theorem yields the expected secrecy property in this case.

A variety of other techniques have been applied to this sort of protocol analysis. They include theorem proving, model checking, and control-flow analysis

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

Figure 1. Syntax of the process calculus

methods (e.g., [7, 8, 17, 20, 25]). Type-based analyses, such as ours, are in part appealing because of their simplicity and because of their connections to classical information-flow methods. They do not require exhaustive state-space exploration; they take advantage of the structure of protocols. On the other hand, they are not always applicable: some reasonable examples fail to typecheck for trivial reasons (as in most typed programming languages and logics). Typing is a discipline, and it works best for processes that use channels and keys in disciplined ways, but disciplined design can lead to more robust protocols [1, 4, 5].

Outline. The next section defines the process calculus. Section 3 defines a concept of secrecy. Section 4 presents the type system. Section 5 establishes the secrecy theorem and other results. Section 6 develops an example. Section 7 concludes. Because of space constraints, we leave for an extended version of this paper: some auxiliary rules, the study of the Needham-Schroeder public-key protocol, (negative) observations on "best" typings, the treatment of symmetric communication primitives, and details of proofs.

2 The Process Calculus (Untyped)

This section introduces our process calculus, by giving its syntax and informal semantics and then formalizing its operational semantics.

The syntax of our calculus is summarized 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. For simplicity, we do not formally separate the names for channels and those for keys. The syntax distinguishes a category of terms (data) and processes (programs). The terms are variables, names, and terms of the form $\{M_1, \ldots M_n\}_M$, which represent encryptions. The processes include constructs for communication, concurrency, and dynamic name creation, roughly those of the pi calculus, a construct for decryption (*case* M *of* $\{x_1, \ldots, x_n\}_k : P$ *else* Q), and a conditional (*if* $M = N$ *then* P *else* Q). As usual, we may omit an "*else*" clause when it consists of the nil process 0.

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. It is also local, as explained below. Therefore, the calculus could be called the local, asynchronous, polyadic spi calculus (but we refrain from such a jargon overload).

The calculus is based on asymmetric communication: channels with only one fixed end-point (the receiver) and public-key encryption. We adopt an elegant, economical approach to asymmetric communication from the local pi calculus [19]. In the local pi calculus, input is possible only on channels that are syntactically represented by names (and not variables). Output is possible on channels represented by names or variables. Thus, the input capability for a channel a remains within the scope of the restriction $(\nu a)P$ where a is created, while the output capability can be transmitted outside. Further, we extend this approach to public-key encryption, as follows. Decryption is possible only with keys that are syntactically represented by names (and not variables). Encryption is possible with keys represented by names or variables. Thus, we model that the encryption capability may be public while the decryption capability remains private, in the scope where it is generated. (We do not explicitly model the distribution of decryption keys across scopes; it is relatively unimportant in public-key cryptosystems.)

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

Specifically, the constructs for asymmetric communication are output, input, encryption, and decryption:

- The process $\overline{M}\langle M_1, \ldots, M_n\rangle$ outputs the tuple $M_1, \ldots, M_n$ on M. Here an arbitrary term M is used to refer to a channel: M can be a variable, a name, or even an encryption. This last case is however unimportant for the present purposes; when M is an encryption, no process can receive the message $\overline{M}\langle M_1, \ldots, M_n\rangle$.
- The process $a(x_1, \ldots, x_n).P$ inputs a message with n components on channel a, then executes P with the variables $x_1, \ldots, x_n$ bound to the contents of the message. Note that a must be a name.
- The term $\{M_1, \ldots, M_n\}_M$ represents an encryption of the tuple $M_1, \ldots, M_n$ under M. Here an arbitrary term M is used to refer to an encryption key: M can be a variable, a name, or even an encryption. This last case is however unimportant, again; when M is an encryption, no process can decrypt $\{M_1, \ldots, M_n\}_M$.

– In the process $case\ M\ of\ \{x_1, \ldots, x_n\}_k : P\ else\ Q$, the term M is decrypted with the key k. If M is indeed a ciphertext encrypted under k, and the underlying plaintext has n components, then the process P is executed with the variables $x_1, \ldots, x_n$ bound to those components. Otherwise, the process Q is executed. Note that k must be a name.

The remaining constructs are standard. The nil process 0 does nothing. The process $P \mid Q$ is the parallel composition of P and Q. The replication $!P$ represents any number of copies of the process P in parallel. The restriction $(\nu a)P$ creates a new name a, and then executes P. The conditional $if\ M = N\ then\ P\ else\ Q$ executes P if M and N evaluate to the same closed term; otherwise, it executes Q. This construct is not always present in relatives of the pi calculus, but it is helpful in modeling security protocols.

For example, the following expression is a process:

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

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

The name a is bound in $(\nu a)P$. The variables $x_1, \ldots, x_n$ are bound in P in the processes $a(x_1, \ldots, x_n).P$ and $case\ M\ of\ \{x_1, \ldots, x_n\}_k : P\ else\ Q$. We write $fn(P)$ and $fv(P)$ for the sets of names and variables free in P, respectively. 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. We write $\{M_1/x_1, \ldots, M_n/x_n\}$ for the substitution that replaces $x_1, \ldots, x_n$ with $M_1, \ldots, M_n$, respectively.

The rules of Figure 2 axiomatize the reduction relation $\rightarrow$ for processes; they are a formal definition for the operational semantics of our calculus. Auxiliary rules axiomatize the structural congruence relation $\equiv$; this relation is useful for transforming processes so that the reduction rules can be applied. Both $\equiv$ and $\rightarrow$ are defined only on closed processes. (In particular, we do not include rules for structural congruence under an input, a decryption, or a conditional; such rules are not necessary to apply reduction rules.) All rules are fairly standard.

Using this operational semantics, we can give a precise definition of a simple concept of output, which we use below in a definition of secrecy. Here, $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$.

Definition 1. *The process P outputs M immediately on c if and only if $P \equiv \overline{c}\langle M\rangle \mid R$ for some process R. The process P outputs M on c if and only if $P \rightarrow^* Q$ and Q outputs M immediately on c for some process Q.*

3 A Definition of Secrecy

We think of an attacker as any process Q of the calculus, under some restrictions that characterize the attacker's initial capabilities. We formulate the restrictions

Structural congruence:

$$
\begin{array}{ll}
P \mid 0 \equiv P & P \equiv Q \;\Rightarrow\; P \mid R \equiv Q \mid R \\
P \mid Q \equiv Q \mid P & P \equiv Q \;\Rightarrow\; !P \equiv !Q \\
(P \mid Q) \mid R \equiv P \mid (Q \mid R) & P \equiv Q \;\Rightarrow\; (\nu a)P \equiv (\nu a)Q \\
!P \equiv P \mid !P & \\
 & P \equiv P \\
(\nu a_1)(\nu a_2)P \equiv (\nu a_2)(\nu a_1)P \;\text{ if } a_1 \neq a_2 & Q \equiv P \;\Rightarrow\; P \equiv Q \\
(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q \;\text{ if } a \notin fn(P) & P \equiv Q,\, Q \equiv R \;\Rightarrow\; P \equiv R
\end{array}
$$

Reduction:

$$
\begin{array}{ll}
\overline{a}\langle M_1, \ldots, M_n\rangle \mid a(x_1, \ldots, x_n).P \;\rightarrow\; P\{M_1/x_1, \ldots, M_n/x_n\} & \text{(Red I/O)} \\
\mathit{case}\ \{M_1, \ldots, M_n\}_k\ \mathit{of}\ \{x_1, \ldots, x_n\}_k : P\ \mathit{else}\ Q & \\
\quad \rightarrow\; P\{M_1/x_1, \ldots, M_n/x_n\} & \text{(Red Decrypt 1)} \\
\mathit{case}\ M\ \mathit{of}\ \{x_1, \ldots, x_n\}_k : P\ \mathit{else}\ Q \;\rightarrow\; Q & \\
\quad \text{if } M \text{ is not of the form } \{M_1, \ldots, M_n\}_k & \text{(Red Decrypt 2)} \\
\mathit{if}\ M = M\ \mathit{then}\ P\ \mathit{else}\ Q \;\rightarrow\; P & \text{(Red Cond 1)} \\
\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q \;\rightarrow\; Q & \\
\quad \text{if } M \neq N & \text{(Red Cond 2)} \\
P \rightarrow Q \;\Rightarrow\; P \mid R \rightarrow Q \mid R & \text{(Red Par)} \\
P \rightarrow Q \;\Rightarrow\; (\nu a)P \rightarrow (\nu a)Q & \text{(Red Res)} \\
P' \equiv P,\, P \rightarrow Q,\, Q \equiv Q' \;\Rightarrow\; P' \rightarrow Q' & \text{(Red} \equiv\text{)}
\end{array}
$$

Figure 2. Operational semantics

by using a set of names RW ("read-write") and a set of closed terms W ("write"). Initially, the attacker is able to output, input, encrypt, and decrypt using the names of RW. He can output and encrypt using names in W. He has the terms in RW and W, and can compute on them and include them in messages. In the course of computation, the attacker may acquire capabilities not represented in RW and W, by creating fresh names and receiving terms in messages.

In order to express the limited use of the terms in W, we also introduce a set of variables $\{x_1, \ldots, x_l\}$ of the same cardinality l as W. When $W = \{M_1, \ldots, M_l\}$, the attacker is a process Q of the form $Q'\{M_1/x_1, \ldots, M_l/x_l\}$. Since Q' should be a well-formed process before the application of the substitution $\{M_1/x_1, \ldots, M_l/x_l\}$, it cannot input or decrypt using the variables $\{x_1, \ldots, x_l\}$. Further, in order to express that the attacker cannot initially use any other names or terms, we impose that $fn(Q') \subseteq RW$ and $fv(Q') \subseteq \{x_1, \ldots, x_l\}$.

Definition 2. *Let RW be a finite set of names and let $W = \{M_1, \ldots, M_l\}$ be a finite set of closed terms. The process Q is a (RW, W)-adversary if and only if it is of the form $Q'\{M_1/x_1, \ldots, M_l/x_l\}$ for some process Q' such that $fn(Q') \subseteq RW$ and $fv(Q') \subseteq \{x_1, \ldots, x_l\}$.*

We say that a process preserves the secrecy of a piece of data M if the process never publishes M, or anything that would permit the computation of M, even in interaction with an attacker. This concept of secrecy is common in the literature on security protocols. A precise definition of it appears in [2], for the spi calculus; Cardelli, Ghelli, and Gordon use that definition in their work on secrecy and groups in the pi calculus [9]. Here we introduce and use a different definition that captures the same concept. This definition takes into account asymmetric communication; it is also more syntactic, and a little easier to treat in our proofs.

Definition 3. *Let RW be a finite set of names and let W be a finite set of closed terms. The process P preserves the secrecy of M from (RW, W) if and only if $P \mid Q$ does not output M on c for any (RW, W)-adversary Q and $c \in RW$.*

Clearly, if P preserves the secrecy of M from (RW, W), it cannot output M on some $c \in RW$, that is, on one of the channels on which an (RW, W)-adversary can read. This guarantee corresponds to the informal requirement that P never publishes M on its own. Moreover, P cannot publish data that would enable an adversary to compute M, because the adversary could go on to output M on some $c \in RW$.

For instance, $(\nu k)\overline{a}\langle\{s\}_k, k\rangle$ preserves the secrecy of s from $(\{a\}, \emptyset)$. This process publishes an encryption of s and the corresponding encryption key on the channel a, but keeps the decryption key, so s does not escape. Similarly, $\overline{a}\langle\{s\}_k, k\rangle$ preserves the secrecy of s from $(\{a\}, \{k\})$. However, $\overline{a}\langle\{s\}_k, k\rangle$ does not preserve the secrecy of s from $(\{a, k\}, \emptyset)$: the adversary $a(x, y).\mathit{case}\ x\ \mathit{of}\ \{z\}_k : \overline{a}\langle z\rangle$ can receive $\{s\}_k$ on a, decrypt s, and resend it on a. As a more substantial, untyped example we consider the following process P:

$$P \triangleq (\nu k)(\overline{a}\langle\{k_1, k_2\}_k\rangle \mid !b(x).\mathit{case}\ x\ \mathit{of}\ \{y_1, y_2\}_k : \overline{c}\langle y_1\rangle)$$

This process relies on three public channels, a, b, and c. It generates a fresh key pair k; outputs the encryption $\{k_1, k_2\}_k$ on a; and receives messages on b, filtering for one encrypted under k, of which it outputs the first component of the plaintext on c. Although P does not output k_1 in clear on its own, it does not preserve the secrecy of k_1 from $(\{a, c\}, \{b\})$: the adversary $a(x).\overline{b}\langle x\rangle$ can receive $\{k_1, k_2\}_k$ on a and resend it on b, causing k_1 to appear on c. On the other hand, P does preserve the secrecy of k_2 from $(\{a, c\}, \{b\})$.

4 The Type System

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

$T ::=$ types
- Public
- Secret
- $\mathrm{C}^{\mathrm{Public}}[T_1, \ldots, T_n]$
- $\mathrm{C}^{\mathrm{Secret}}[T_1, \ldots, T_n]$
- $\mathrm{K}^{\mathrm{Secret}}[T_1, \ldots, T_n]$
- $\mathrm{K}^{\mathrm{Public}}[T_1, \ldots, T_n]$

We let L range over $\{\text{Public}, \text{Secret}\}$, and may for example write $\mathrm{C}^L[T_1, \ldots, T_n]$ and $\mathrm{K}^L[T_1, \ldots, T_n]$. The subtyping relation is the least reflexive relation such that $\mathrm{C}^L[T_1, \ldots, T_n] \leq L$ and $\mathrm{K}^L[T_1, \ldots, T_n] \leq L$. (We do not have Secret $\leq$ Public or vice versa.)

The types have the following informal meanings:

- Public is the type of public data, and is a supertype of all types $\mathrm{C}^{\text{Public}}[T_1, \ldots, T_n]$ and $\mathrm{K}^{\text{Public}}[T_1, \ldots, T_n]$.
- Similarly, Secret is the type of secret data, and is a supertype of all types $\mathrm{C}^{\text{Secret}}[T_1, \ldots, T_n]$ and $\mathrm{K}^{\text{Secret}}[T_1, \ldots, T_n]$.
- $\mathrm{C}^{\text{Secret}}[T_1, \ldots, T_n]$ is the type of a channel on which the adversary cannot send messages, and which conveys n-tuples with components of respective types $T_1, \ldots, T_n$.
- Similarly, $\mathrm{K}^{\text{Secret}}[T_1, \ldots, T_n]$ is the type of an encryption key that the adversary does not have, and which is used to encrypt n-tuples with components of respective types $T_1, \ldots, T_n$.
- $\mathrm{C}^{\text{Public}}[T_1, \ldots, T_n]$ is the type of a channel on which the adversary may send messages; the channel may be intended to convey n-tuples with components of respective types $T_1, \ldots, T_n$, but the adversary may send any data it has (that is, any public data) on the channel.
- Similarly, $\mathrm{K}^{\text{Public}}[T_1, \ldots, T_n]$ is the type of an encryption key that the adversary may have; this key may be intended for encrypting n-tuples with components of respective types $T_1, \ldots, T_n$, but the adversary may encrypt any data it has (that is, any public data) under this key.

Figure 3 gives the main rules of the type system. In the rules, the metavariable u ranges over names and variables. The rules concern four judgments:

(1) $E \vdash \diamond$ means that E is a well-formed environment. The environment E is well-formed if and only if E is a list of pairs $u : T$ where each u is a name or a variable and distinct from all others in E.
(2) $E \vdash M : T$ means that M is a term of type T in the environment E. Basically, names and variables have the types declared in E, and any supertypes, while encryptions all have the type Public.
(3) $E \vdash_\diamond M : S$ means that S is the set of possible "true" types of M in the environment E (the declared type when M is a name, the declared type and any subtype when M is a variable, and Public when M is an encryption).
(4) $E \vdash P$ says that the process P is well-typed in the environment E.

Figure 3 omits the rules for proving the first and the third.

The type rules for output say that any public data can be sent on a public channel (Output Public), and tuples with the expected types $T_1, \ldots, T_n$ can be sent on a channel of type $\mathrm{C}^L[T_1, \ldots, T_n]$ (Output C^L). Therefore, by subtyping, any public data can be sent on a channel of type $\mathrm{C}^{\text{Public}}[T_1, \ldots, T_n]$. This use of the channel may not seem to conform to its declared type. However, it is unavoidable, since we expect that an attacker can use the channel; moreover, it does not cause harm from the point of view of secrecy. Similarly, the type rule

$$\frac{E \vdash \diamond \qquad (u : T) \in E}{E \vdash u : T} \qquad \text{(Atom)}$$

$$\frac{E \vdash M : \text{Public} \qquad \forall i \in \{1, \ldots, n\}, E \vdash M_i : \text{Public}}{E \vdash \{M_1, \ldots, M_n\}_M : \text{Public}} \qquad \text{(Encrypt Public)}$$

$$\frac{E \vdash M : \mathrm{K}^L[T_1, \ldots, T_n] \qquad \forall i \in \{1, \ldots, n\}, E \vdash M_i : T_i}{E \vdash \{M_1, \ldots, M_n\}_M : \text{Public}} \qquad \text{(Encrypt K}^L\text{)}$$

$$\frac{E \vdash M : T \qquad T \leq T'}{E \vdash M : T'} \qquad \text{(Subsumption)}$$

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

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

$$\frac{(a : \text{Public}) \in E \qquad E, x_1 : \text{Public}, \ldots, x_n : \text{Public} \vdash P}{E \vdash a(x_1, \ldots, x_n).P} \qquad \text{(Input Public)}$$

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

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

$$\frac{E \vdash \diamond}{E \vdash 0} \qquad \text{(Nil)}$$

$$\frac{E \vdash P \qquad E \vdash Q}{E \vdash P \mid Q} \qquad \text{(Parallel)}$$

$$\frac{E \vdash P}{E \vdash !P} \qquad \text{(Replication)}$$

$$\frac{E, a : T \vdash P}{E \vdash (\nu a)P} \qquad \text{(Restriction)}$$

$$\frac{\begin{array}{c}E \vdash M : \text{Public} \qquad (k : \text{Public}) \in E \\ E, x_1 : \text{Public}, \ldots, x_n : \text{Public} \vdash P \qquad E \vdash Q\end{array}}{E \vdash \mathit{case}\ M\ \mathit{of}\ \{x_1, \ldots, x_n\}_k : P\ \mathit{else}\ Q} \qquad \text{(Decrypt Public)}$$

$$\frac{\begin{array}{c}E \vdash M : \text{Public} \qquad (k : \mathrm{K}^{\text{Public}}[T_1, \ldots, T_m]) \in E \\ E, x_1 : \text{Public}, \ldots, x_n : \text{Public} \vdash P \qquad E \vdash Q \\ E, x_1 : T_1, \ldots, x_m : T_m \vdash P \ \text{ if } m = n\end{array}}{E \vdash \mathit{case}\ M\ \mathit{of}\ \{x_1, \ldots, x_n\}_k : P\ \mathit{else}\ Q} \qquad \text{(Decrypt K}^{\text{Public}}\text{)}$$

$$\frac{\begin{array}{c}E \vdash M : \text{Public} \qquad (k : \mathrm{K}^{\text{Secret}}[T_1, \ldots, T_n]) \in E \\ E, x_1 : T_1, \ldots, x_n : T_n \vdash P \qquad E \vdash Q\end{array}}{E \vdash \mathit{case}\ M\ \mathit{of}\ \{x_1, \ldots, x_n\}_k : P\ \mathit{else}\ Q} \qquad \text{(Decrypt K}^{\text{Secret}}\text{)}$$

$$\frac{E \vdash_\diamond M : S_1 \qquad E \vdash_\diamond N : S_2 \qquad \text{if } S_1 \cap S_2 \neq \emptyset \text{ then } E \vdash P \qquad E \vdash Q}{E \vdash \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q} \qquad \text{(Cond)}$$

Figure 3. Type rules: terms and processes

(Output Public) also permits processes that use an encryption as a channel, such as $\overline{\{M\}_k}\langle N\rangle$. A standard type system might attempt to exclude such processes. Ours does not, essentially because an attacker might run $\overline{\{M\}_k}\langle N\rangle$, but this does not cause harm from the point of view of secrecy. On the other hand, the attacker cannot have channels of type $\mathrm{C}^{\mathrm{Secret}}[T_1, \ldots, T_n]$. Therefore, we can guarantee that such channels are represented by names at run-time and that only tuples with types $T_1, \ldots, T_n$ can be sent on such channels.

As for input, we distinguish three cases, considering the type of the channel a on which an input happens:

- 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. This rule may seem superfluously liberal, but it is helpful in our proofs. It enables us to type an arbitrary adversary, which can input public values on the public channels on which it listens (see section 5).
- When a is of type $\mathrm{C}^{\mathrm{Public}}[T_1, \ldots, T_n]$, 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 $T_1, \ldots, T_n$. 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 $\mathrm{C}^{\mathrm{Secret}}[T_1, \ldots, T_n]$, 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 $T_1, \ldots, T_n$.

The type rules for encryption are similar to those for output. Any public data can be encrypted under a public encryption key (Encrypt Public), and data of types $T_1, \ldots, T_n$ can be encrypted under a key of type $\mathrm{K}^L[T_1, \ldots, T_n]$ (Encrypt K^L). Ciphertexts are always of type Public; this typing simplifies the rules and is reasonable for most protocols (particularly for most public-key protocols).

The type rules for decryption resemble those for input, in the same way as those for encryption resemble those for output.

The type rules for nil, parallel composition, replication, and restriction are standard. It is worth noting that we use a Curry-style typing for restriction, so we do not mention a type of a explicitly in the construct (νa). (That is, we do not write $(\nu a : T)$ for some T.) This style of typing gives rise to an interesting form of polymorphism: the type of a can change according to the environment. For instance, in $c(x).(\nu a)\overline{x}\langle a\rangle$, with c of type $\mathrm{C}^{\mathrm{Public}}[\mathrm{C}^{\mathrm{Secret}}[\mathrm{Secret}]]$, a can be of type Secret when x is of type $\mathrm{C}^{\mathrm{Secret}}[\mathrm{Secret}]$, and of type Public when x is of type Public, so that the output $\overline{x}\langle a\rangle$ is well-typed in both cases.

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, *if* $M = N$ *then* P *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, x may also be substituted at run-time by 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. When M is a ciphertext, its only possible type is Public, by definition of the judgment $E \vdash M : T$.

The following example illustrates the use of rule (Cond), informally. Suppose that a participant A in a security protocol invents a fresh quantity a of type Secret (as a nonce challenge), sends it on a channel of type $\mathrm{C}^{\mathrm{Secret}}[\mathrm{Secret}]$, so a should remain secret. Some time later, A gets a ciphertext encrypted under a key k of type $\mathrm{K}^{\mathrm{Public}}[\mathrm{Secret}, T]$; decryption yields a pair x_1, x_2. The type system covers two possibilities: (1) x_1 has type Secret and x_2 has type T, (2) both are public (for example, because the attacker constructed the ciphertext). That is, the type rule for decryption (Decrypt $\mathrm{K}^{\mathrm{Public}}$) has hypotheses that correspond to each of these possibilities. Suppose further that A checks, dynamically, that $x_1 = a$ (and halts if $x_1 \neq a$). This check guarantees that x_1 has type Secret, and hence is not public. At the same time, it guarantees that x_2 has type T. After the check, A can assume that x_2 has type T, and act accordingly. Dynamic checks of this sort are common and important in security protocols.

This type system reflects a binary view of secrecy, according to which the world is divided into system and attacker, and a secret is something that the attacker does not have. When we wish to express that a piece of data is a secret for a given set of principals, we define the system to include only the processes that represent those principals.

In this respect and in others, our type system is most similar to that of Abadi [1] for the spi calculus and that of Cardelli, Ghelli, and Gordon [9, section 4] for the pi calculus. Both treat only symmetric communication primitives. The latter, however, is mostly introduced as an auxiliary type system for a proof. The proof concerns another type system, which elegantly exploits a powerful construct for group creation. Group creation directly supports a rich view of secrecy that does not simply divide the world into two parts. We believe that the type system with group creation can be extended with symmetric cryptographic primitives and further extended to deal with asymmetric communication. Unfortunately (as far as we can tell) this last extension does not retain the elegance of the original.

5 The Secrecy Theorem and Other Results

This section studies the type system of section 4. First it establishes a subject-reduction theorem and a typability lemma. Then it derives the secrecy theorem sketched in the introduction.

The subject-reduction theorem says that typing is preserved by computation. Its proof is mostly a fairly routine induction on computations with a case analysis on typing proofs.

Theorem 1 (Subject congruence and subject reduction). *If $E \vdash P$ and if $P \equiv Q$ or $P \to Q$ then $E \vdash Q$.*

The following typability lemma says that every process is well-typed, at least in a fairly trivial way that makes its free names public. This lemma is important because it means that any process that represents an adversary is well-typed. It is a formal counterpart to the informal idea that the type system cannot constrain the adversary. Its proof is an easy induction on the structure of P.

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

$$a_1 : Public, \ldots, a_n : Public, x_1 : T_1, \ldots, x_m : T_m \vdash P$$

The secrecy theorem says that if a closed process P is well-typed in an environment E, and a name s has type Secret in E, then P preserves the secrecy of s from (RW, W), where RW is the set of names declared Public in E, and W is the set of names declared $\mathrm{C}^{\mathrm{Public}}[\ldots]$ or $\mathrm{K}^{\mathrm{Public}}[\ldots]$. The name s may be declared Secret in E, but it may also be declared $\mathrm{C}^{\mathrm{Secret}}[\ldots]$ or $\mathrm{K}^{\mathrm{Secret}}[\ldots]$. In other words, P preserves the secrecy of Secret names against adversaries that can output, input, encrypt, and decrypt on names declared Public, and output and encrypt on names declared $\mathrm{C}^{\mathrm{Public}}[\ldots]$ and $\mathrm{K}^{\mathrm{Public}}[\ldots]$.

Theorem 2 (Secrecy). *Let P be a closed process. Suppose that $E \vdash P$ and $E \vdash s : Secret$. Let*

$$\begin{aligned} RW &= \{a \mid (a : Public) \in E\} \\ W &= \{a' \mid (a' : C^{Public}[\ldots]) \in E \text{ or } (a' : K^{Public}[\ldots]) \in E\} \end{aligned}$$

Then P preserves the secrecy of s from (RW, W).

Proof. The secrecy theorem is a fairly easy consequence of the subject-reduction theorem and the typability lemma. (In truth, the type system and the definition of secrecy were refined with this proof of the secrecy theorem in mind.) Suppose that $RW = \{a_1, ..., a_n\}$ and $W = \{a'_1, ..., a'_l\}$, and that T'_i is the type of a'_i in E, so $(a'_i{:}T'_i) \in E$ for all $i \in \{1, \ldots, l\}$. In order to derive a contradiction, we assume that P does not preserve the secrecy of s from (RW, W). Then there exists a process $Q = Q'\{a'_1/x_1, \ldots, a'_l/x_l\}$ with $fn(Q') \subseteq RW$ and $fv(Q') \subseteq \{x_1, \ldots, x_l\}$, such that $P \mid Q \rightarrow^* R$ and $R \equiv \overline{c}\langle s\rangle \mid R'$, where $c \in RW$. By Lemma 1, $a_1 : \mathrm{Public}, \ldots, a_n : \mathrm{Public}, x_1 : T'_1, \ldots, x_l : T'_l \vdash Q'$. By a standard substitution lemma, $E \vdash Q'\{a'_1/x_1, \ldots, a'_l/x_l\}$, that is, $E \vdash Q$. Therefore, $E \vdash P \mid Q$. By Theorem 1, $E \vdash R$ and $E \vdash \overline{c}\langle s\rangle \mid R'$. Since $c \in RW$, we have $(c : \mathrm{Public}) \in E$, so $E \vdash \overline{c}\langle s\rangle$ could have been derived only by (Output Public). Such a derivation would require that $E \vdash s : \mathrm{Public}$. However, this is impossible, since we have $E \vdash s : \mathrm{Secret}$ and the two typings are incompatible. We have a contradiction, so P preserves the secrecy of s from (RW, W). □

We restate a special case of the theorem, as it may be particularly clear.

Corollary 1. *Suppose that $a : Public, s : T \vdash P$ with $T \leq Secret$. Then P preserves the secrecy of s from $(\{a\}, \emptyset)$. That is, for all closed processes Q such that $fn(Q) \subseteq \{a\}$, $P \mid Q$ does not output s on a.*

For instance, we can obtain $a : \mathrm{Public}, s : \mathrm{Secret} \vdash (\nu k)\overline{a}\langle\{s\}_k, k\rangle$ by letting $k : \mathrm{K}^{\mathrm{Public}}[\mathrm{Secret}]$. So this corollary implies that $(\nu k)\overline{a}\langle\{s\}_k, k\rangle$ preserves the secrecy of s from $(\{a\}, \emptyset)$, as claimed in section 3. In other words, if Q is a closed process and $fn(Q) \subseteq \{a\}$, then $((\nu k)\overline{a}\langle\{s\}_k, k\rangle) \mid Q$ does not output s on a. Thus, assuming that Q does not have s in advance, Q cannot guess s or compute it from the message on a.

6 Further Examples

This section applies the type system to security protocols. Although these protocols are often fairly small, informal reasoning about them is error-prone and difficult. The type system provides a simple yet rigorous approach for proving secrecy properties of these protocols.

Because of space constraints, we develop only one example. This example does not explicitly rely on cryptography but it brings up some issues common in cryptographic protocols. An extended version of this paper also covers the Needham-Schroeder public-key protocol and a variant (in 3–4 pages).

Our example concerns a protocol in which a principal A sends a secret s to a principal B. The following message sequence describes the protocol informally:

$$\begin{array}{l}\text{Message 1. } A \rightarrow B : k, a \text{ on } b\\ \text{Message 2. } B \rightarrow A : k, b' \text{ on } a\\ \text{Message 3. } A \rightarrow B : s \text{ on } b'\end{array}$$

Here, a and b are channels with A and B as only receivers, respectively; k is a secret nonce, created by A; and b' is a new channel, created by B, with B as only receiver and A as only sender. Instead of sending s directly on b, A creates the nonce k and sends it along with the return channel a on b; B's reply contains k, as proof of origin; the reply also includes the fresh channel b' on which A sends s. This channel is analogous to a session key in a cryptographic protocol.

We may represent the principals of this protocol by the processes:

$$\begin{array}{l}A \triangleq (\nu k)(\overline{b}\langle k, a\rangle \mid a(x, y).\mathit{if}\ x = k\ \mathit{then}\ \overline{y}\langle s\rangle)\\ B \triangleq b(x, y).(\nu b')(\overline{y}\langle x, b'\rangle \mid b'(z).0)\end{array}$$

We can then use our type system to prove that s remains secret, as expected. For this proof, we let:

$$\begin{array}{rl}E \triangleq & a : \mathrm{C}^{\mathrm{Public}}[\mathrm{Secret}, \mathrm{C}^{\mathrm{Secret}}[\mathrm{Secret}]],\\ & b : \mathrm{C}^{\mathrm{Public}}[\mathrm{Secret}, \mathrm{C}^{\mathrm{Public}}[\mathrm{Secret}, \mathrm{C}^{\mathrm{Secret}}[\mathrm{Secret}]]],\\ & s : \mathrm{Secret}\end{array}$$

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

(1) $x : \text{Public}, y : \text{Public}$: This case is vacuous by rule (Cond): in the test $x = k$, the two terms cannot have the same type.
(2) $x : \text{Secret}, y : \mathrm{C}^{\text{Secret}}[\text{Secret}]$: In this case, $\overline{y}\langle s\rangle$ is typed by (Output C^L).

The input $b(x, y)$ is also typed by (Input $\mathrm{C}^{\text{Public}}$), and again two cases arise:

(1) $x : \text{Public}, y : \text{Public}$: In this case, we let $b' : \text{Public}$.
(2) $x : \text{Secret}, y : \mathrm{C}^{\text{Public}}[\text{Secret}, \mathrm{C}^{\text{Secret}}[\text{Secret}]]$: In this case, instead, we let $b' : \mathrm{C}^{\text{Secret}}[\text{Secret}]$.

In both cases, the rest of process B is easy to typecheck. Having derived $E, c : \text{Public} \vdash A \mid B$, we apply Theorem 2, and conclude that $A \mid B$ preserves the secrecy of s from $(\{c\}, \{a, b\})$.

We can also treat a more general system in which A and B communicate with other principals: A may initiate sessions with others than B, and B is willing to respond to several principals at once. Still, s should remain within $A \mid B$, and not escape to third parties. The following process represents the system:

$$P \triangleq (A \mid A') \mid !B$$

Here A' is an arbitrary process, notionally grouped with A, which may receive messages on a, under the assumption that $E, E' \vdash A'$ for some E'. In particular, this assumption implies that A' respects the secrecy of s and uses a in conformance with $a : \mathrm{C}^{\text{Public}}[\text{Secret}, \mathrm{C}^{\text{Secret}}[\text{Secret}]]$. It follows that $E, E' \vdash P$, so Theorem 2 still applies. For example, A' may be:

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

that is, a variant of A that initiates a session with a third party on the channel c and sends s'. With $E' = c : \text{Public}, s' : \text{Public}$, Theorem 2 yields that P preserves the secrecy of s from $(\{c, s'\}, \{a, b\})$. Remarkably, the replication of B causes no complication whatsoever. In contrast, replication tends to be problematic for proof methods based on state-space exploration.

7 Conclusion

This paper presents a type system that can serve in establishing secrecy properties of processes. The type system is superficially straightforward: it consists of fairly elementary rules in a standard format. The proof of its soundness (the subject-reduction theorem) is also fairly standard; the subject-reduction theorem then yields the main secrecy theorem. This simplicity is not entirely accidental: we explored more complex type systems (with dependent types) and notions of secrecy before arriving at the current ones.

On the other hand, the type system is powerful enough to apply to some delicate security protocols, yielding concise proofs for subtle results. It is also fairly tricky, when examined more closely. For instance, as indicated above, the type rules allow certain forms of polymorphism.

A challenging subject for further work is to develop type systems with richer forms of polymorphism, with stronger theories, perhaps even with algorithms for inferring secrecy types. In another direction, it would be worthwhile to give type rules for more cryptographic primitives, for example for digital signatures. Finally, it would be useful to integrate proofs by typing with other proof methods.

References

[1] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

[2] Martín Abadi. Security protocols and their properties. In F.L. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, NATO Science Series, pages 39–60. IOS Press, 2000. Volume for the 20th International Summer School on Foundations of Secure Computation, held in Marktoberdorf, Germany (1999).

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.

[4] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[5] Ross Anderson and Roger Needham. Robustness principles for public key protocols. In *Proceedings of Crypto '95*, pages 236–247, 1995.

[6] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software Practice and Experience*, S4(25):87–130, December 1995.

[7] Chiara Bodei. *Security Issues in Process Calculi*. PhD thesis, Università di Pisa, January 2000.

[8] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the π-calculus. In *CONCUR'98: Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98. Springer Verlag, September 1998.

[9] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In Catuscia Palamidessi, editor, *CONCUR 2000: Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 365–379. Springer-Verlag, August 2000.

[10] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Mass., 1982.

[11] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, January 1996.

[12] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, April 1984.

[13] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[14] Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 415–427. Springer-Verlag, 2000.

[15] Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In Gert Smolka, editor, *Programming Languages and Systems: Proceedings of the 9th European Symposium on Programming (ESOP 2000), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 2000.

[16] K. Rustan M. Leino and Rajeev Joshi. A semantic approach to secure information flow. In *Mathematics of Program Construction, 4th International Conference*, volume 1422 of *Lecture Notes in Computer Science*, pages 254–271. Springer Verlag, 1998.

[17] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996.

[18] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[19] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer-Verlag, 1998.

[20] Jon Millen and Harald Ruess. Protocol-independent secrecy. In *Proceedings 2000 IEEE Symposium on Security and Privacy*, pages 110–119, May 2000.

[21] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, September 1992.

[22] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, January 1999.

[23] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 129–142, 1997.

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

[25] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[26] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, September 2000.

[27] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 93–104, January 1999.

[28] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, 1996.