Analyzing Security Protocols with Secrecy Types and Logic Programs

Martín Abadi*
Computer Science Department
University of California, Santa Cruz
abadi@cs.ucsc.edu

Bruno Blanchet†
Département d'Informatique
Ecole Normale Supérieure, Paris
Bruno.Blanchet@ens.fr

ABSTRACT

We study and further develop two language-based techniques for analyzing security protocols. One is based on a typed process calculus; the other, on untyped logic programs. Both focus on secrecy properties. We contribute to these two techniques, in particular by extending the former with a flexible, generic treatment of many cryptographic operations. We also establish an equivalence between the two techniques.

1. INTRODUCTION

Concepts and methods from programming languages have long been useful in security (e.g., [30]). In recent years, they have played a significant role in understanding security protocols. They have given rise to programming calculi for these protocols (e.g., [5, 6, 7, 12, 13, 16, 17, 19, 26, 28, 34]). They have also suggested several approaches for reasoning about protocols, leading to theories as well as tools for formal protocol analysis. We describe some of these approaches below. Although several of them are incomplete, they are applicable to many protocols, including infinite-state protocols, often with little effort. Thus, they provide an attractive alternative to finite-state model checking (e.g., [27]) and to human-guided theorem proving (e.g., [33]).

In this work we pursue these language-based approaches to protocol analysis and aim to clarify their interconnections. We examine and further develop two techniques that represent two substantial but largely disjoint lines of research. One technique relies on a typed process calculus, the other on untyped logic programs. We contribute to these two techniques, in particular by extending the former with a flexible, generic treatment of many cryptographic operations. We also establish an equivalence between the two techniques. We believe that this equivalence is surprising and illuminating.

The typed process calculus belongs in a line of research that exploits standard static-analysis ideas and adapts them with security twists. There are by now several type systems for processes in which types not only track the expected structure of values and processes but also give security information [1, 4, 11, 20, 22, 23]. A related approach relies on control-flow analysis [10]; it has an algorithmic emphasis, but it is roughly equivalent to typing at least in important special cases [9]. Such static analyses have applications in a broader security context (e.g., [3, 21, 31, 35]); security protocols constitute a particularly challenging class of examples. To date, however, such static analyses have dealt case by case with operations on data, and in particular with cryptographic operations. In this paper, we develop a general treatment of these operations.

In another line of research, security protocols are represented as logic programs, in classical or linear logic, and they are analyzed symbolically with general provers [15, 36] or with special-purpose algorithms and tools [8, 12, 14] (see also [24] for some of the roots of this approach). Superficially, these algorithms and tools are quite different from typing and control-flow analysis. However, in this paper we show that one of these tools can be viewed as an implementation of a type system.

More specifically, we develop a generic type system for a process calculus that extends the pi calculus [29] with constructor operations and corresponding destructor operations. These operations may be, for instance, tupling and projection, symmetric (shared-key) encryption and decryption, asymmetric (public-key) encryption and decryption, digital signatures and signature checking, and one-way hashing (with no corresponding destructor). As in the applied pi calculus [5], these operations are not hardwired. The applied pi calculus is even more general in that it does not require the classification of operations into constructors and destructors; we suspect that it might be treated along similar lines but with more difficulty (see section 2). Our type system for the process calculus gives secrecy information. The basic soundness theorem for the type system, which we prove only once (rather than once per choice of operations),

*This work was partly done while the author was at Bell Labs Research, Lucent Technologies, and at InterTrust's Strategic Technologies and Architectural Research Laboratory.

†This work was partly done while the author was at INRIA Rocquencourt.

POPL '02, Jan. 16-18, 2002 Portland, OR USA

states that well-typed processes do not reveal their secrets.

We compare this generic type system with an automatic protocol checker. The checker takes as input a process and translates it into an abstract representation by logic-programming rules. This representation and its manipulation, but not the translation of processes, come from previous work [8], which develops an efficient tool for establishing secrecy properties of protocols. We show that establishing a secrecy property of a protocol with this checker corresponds to typing the protocol in a particular instance of the generic type system. This result implies a soundness property for the checker. Conversely, as a completeness property, we establish that the checker corresponds to the "best" instance of our generic type system: if a secrecy property can be established using any instance of the type system, then it can also be established by the checker.

The next section presents our process calculus, without types. Section 3 gives a (fairly standard) definition of secrecy. Section 4 presents our type system, and section 5 gives the main soundness theorems for the type system and related results. As an example, section 6 explains how the type system applies to shared-key and public-key encryption operations. Section 7 formalizes and studies the logic-programming protocol checker. Section 8 concludes.

2. THE PROCESS CALCULUS (UNTYPED)

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

2.1 Syntax and Informal Semantics

The syntax of our calculus is summarized in Figure 1. It distinguishes a category of terms (data) and one of processes (programs). 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. It also assumes a set of symbols for constructors and destructors; we often use f for a constructor and g for a destructor.

Constructors are used to build terms. Therefore, the terms are variables, names, and constructor applications of the form $f(M_1, \ldots, M_n)$. On the other hand, destructors do not appear in terms, but only manipulate terms in processes. They are partial functions on terms that processes can apply. The process $let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q$ tries to evaluate $g(M_1, \ldots, M_n)$; if this succeeds, then x is bound to the result and P is executed, else Q is executed. More precisely, the semantics of a destructor g of arity n is given by a partial function from n-tuples of terms to terms, such that $g(\sigma M_1, \ldots, \sigma M_n) = \sigma g(M_1, \ldots, M_n)$ if $g(M_1, \ldots, M_n)$ is defined and σ is a substitution that maps names and variables to terms. We may isolate a minimal set $\mathrm{def}(g)$ of equations $g(M'_1, \ldots, M'_n) = M'$ that define g, where $M'_1, \ldots, M'_n, M'$ are terms without free names. Then $g(M_1, \ldots, M_n)$ is defined if and only if there exists a substitution σ and an equation $g(M'_1, \ldots, M'_n) = M'$ in $\mathrm{def}(g)$ such that $M_i = \sigma M'_i$ for all $i \in \{1, \ldots, n\}$, and $g(M_1, \ldots, M_n) = \sigma M'$. This set of equations may be infinite, but it is usually finite and small in concrete examples.

Using these constructors and destructors, we can represent data structures, such as tuples, and cryptographic operations, for instance as follows:

- $\mathit{ntuple}(M_1, \ldots, M_n)$ is the tuple of the terms $M_1, \ldots, M_n$, where ntuple is a constructor. (We sometimes abbreviate $\mathit{ntuple}(M_1, \ldots, M_n)$ to $(M_1, \ldots, M_n)$.) The n projections are destructors ith_n for $i \in \{1, \ldots, n\}$, defined by

$$ith_n(\mathit{ntuple}(M_1, \ldots, M_n)) = M_i$$

- $\mathit{sencrypt}(M, N)$ is the symmetric (shared-key) encryption of the message M under the key N, where $\mathit{sencrypt}$ is a constructor. The corresponding destructor $\mathit{sdecrypt}$ is defined by

$$\mathit{sdecrypt}(\mathit{sencrypt}(M, N), N) = M$$

Thus, $\mathit{sdecrypt}(M', N)$ returns the decryption of M' if M' is a message encrypted under N.

- In order to represent asymmetric (public-key) encryption, we may use two constructors pk and $\mathit{pencrypt}$: $pk(M)$ builds a public key from a secret M and $\mathit{pencrypt}(M, N)$ encrypts M under N. The corresponding destructor $\mathit{pdecrypt}$ is defined by

$$\mathit{pdecrypt}(\mathit{pencrypt}(M, pk(N)), N) = M$$

- As for digital signatures, we may use a constructor sign, and write $\mathit{sign}(M, N)$ for M signed with the signature key N, and the two destructors $\mathit{checksignature}$ and $\mathit{getmessage}$ with the equations:

$$\mathit{checksignature}(\mathit{sign}(M, N), pk(N)) = M$$
$$\mathit{getmessage}(\mathit{sign}(M, N)) = M$$

- We may represent a one-way hash function by the constructor H. There is no corresponding destructor; so we model that the term M cannot be retrieved from its hash $H(M)$.

Thus, the process calculus supports many of the operations common in security protocols. It has limitations, though: for example, XOR is neither a constructor nor a destructor.

The other constructs in the syntax of Figure 1 are standard; most of them come from the pi calculus. The input process $M(x).P$ inputs a message on channel M, and executes P with x bound to the input message. The output process $\overline{M}\langle N\rangle.P$ outputs the message N on the channel M and then executes P. Here, we use an arbitrary term M to represent a channel: M can be a name, a variable, or a constructed term, but the process blocks if M does not reduce to a name at runtime. Our calculus is monadic (in that the messages are terms rather than tuples of terms), but a polyadic calculus can be simulated since tuples are terms. It is also synchronous (in that a process P is executed after the output of a message). The nil process 0 does nothing. The process $P \mid Q$ is the parallel composition of P and Q. The replication $!P$ represents an unbounded number of copies of P in parallel. The restriction $(\nu a)P$ creates a new name a, and then executes P. The local definition $let\ x = M\ in\ P$ executes P with x bound to the term M. The conditional $if\ M = N\ then\ P\ else\ Q$ executes P if M and N reduce to the same term at runtime; otherwise, it executes Q. As usual, we may omit an $else$ clause when it consists of 0.

The name a is bound in the process $(\nu a)P$. The variable x is bound in P in the processes $M(x).P$, $let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q$, and $let\ x = M\ in\ P$. We write $\mathit{fn}(P)$ and $\mathit{fv}(P)$ for the sets of names and variables free in P, respectively. A process is closed if it has no free

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
$\overline{M}\langle N \rangle.P$	output
$M(x).P$	input
0	nil
$P \mid Q$	parallel composition
$!P$	replication
$(\nu a)P$	restriction
$\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$	destructor application
$\mathit{let}\ x = M\ \mathit{in}\ P$	local definition
$\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$	conditional

Figure 1: Syntax of the process calculus

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, ..., x_n with M_1, ..., M_n, respectively.

As an example, we may express the following protocol:

Message 1. $A \rightarrow B : \mathit{pencrypt}((k, pK_A), pK_B)$
Message 2. $B \rightarrow A : \mathit{pencrypt}((k, K_{AB}), pK_A)$
Message 3. $A \rightarrow B : \mathit{sencrypt}(s, K_{AB})$

This protocol establishes a session key K_{AB} between two parties A and B, then uses the key to transmit a secret s from A to B. It relies on public keys pK_A for A and pK_B for B. First, A creates a challenge k (a nonce), sends it to B paired with A's public key, encrypted under B's public key. Then B replies with the same nonce and the session key K_{AB}, encrypted under A's public key. When A receives this message, it recognizes k; it is then confident that the key K_{AB} has been created by B. Finally, A sends the secret s under K_{AB}. The following processes represent the protocol:

$$
\begin{aligned}
P &\triangleq (\nu sK_A)(\nu sK_B)\mathit{let}\ pK_A = pk(sK_A)\ \mathit{in} \\
&\qquad \mathit{let}\ pK_B = pk(sK_B)\ \mathit{in}\ \overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle.(A \mid B) \\
A &\triangleq (\nu k)\overline{e}\langle \mathit{pencrypt}((k, pK_A), pK_B)\rangle. \\
&\qquad e(z).\mathit{let}\ (x, y) = \mathit{pdecrypt}(z, sK_A)\ \mathit{in} \\
&\qquad \mathit{if}\ x = k\ \mathit{then}\ \overline{e}\langle \mathit{sencrypt}(s, y)\rangle \\
B &\triangleq e(z).\mathit{let}\ (x, y) = \mathit{pdecrypt}(z, sK_B)\ \mathit{in} \\
&\qquad (\nu K_{AB})\overline{e}\langle \mathit{pencrypt}((x, K_{AB}), y)\rangle. \\
&\qquad e(z').\mathit{let}\ s' = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0
\end{aligned}
$$

(This code corresponds to a simple, one-shot version of the protocol; it is easy to extend it to represent more elaborate versions.) Here we write $\mathit{let}\ (x, y) = M\ \mathit{in}\ Q$ instead of $\mathit{let}\ z = M\ \mathit{in}\ \mathit{let}\ x = 1th_2(z)\ \mathit{in}\ \mathit{let}\ y = 2th_2(z)\ \mathit{in}\ Q$, using pattern-matching on tuples. The keys sK_A and sK_B are the decryption keys that match pK_A and pK_B, respectively, and e is a public channel. The messages $\overline{e}\langle pK_A\rangle$ and $\overline{e}\langle pK_B\rangle$, which publish pK_A and pK_B on e, model the fact that these keys are public. We return to this example in section 6.

2.2 Formal Semantics

The rules of Figure 2 axiomatize the reduction relation $\rightarrow$ for processes, thus defining 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.

We write $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$. As in [4], we say that the process P outputs M immediately on c if and only if $P \equiv \overline{c}\langle M\rangle.Q \mid R$ for some processes Q and R. We say that 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.

2.3 Discussion

As mentioned in the introduction, our calculus resembles the applied pi calculus [5]. Both calculi are extensions of the pi calculus with (fairly arbitrary) functions on terms. However, there are also important differences between these calculi. The first one is that we use destructors instead of the equational theories of the applied pi calculus. The second difference is that our calculus has a built-in error-handling construct (the *else* clause of the destructor application), whereas in the applied pi calculus the error-handling must be done "by hand". This error-handling construct makes typing easier.

3. A DEFINITION OF SECRECY

Throughout this paper, we use the following informal definition of secrecy: a protocol P preserves the secrecy of data M if P never publishes M, or anything that would permit the computation of M, even in interaction with an adversary Q. Equivalently, a protocol P preserves the secrecy of data M if P in parallel with an adversary Q will never output M on a public channel. Definitions along these lines are quite common in protocol analysis. (There are however alternatives, in particular some based on the concept of noninterference; see [2] for discussion.)

Next we give a formal counterpart for this informal definition, in the context of our process calculus and relying on the operational semantics of section 2. We represent the adversary Q as a process of the calculus, with some hypotheses that characterize Q's initial capabilities. We formulate these hypotheses simply by using a set of names S. Intuitively, Q knows the names in S initially; and Q may acquire some additional capabilities not represented in S in the course of

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

$$
\begin{array}{ll}
\overline{a}\langle M\rangle.Q \mid a(x).P \;\rightarrow\; Q \mid P\{M/x\} & \text{(Red I/O)} \\
\mathit{let}\ x = g(M_1,\ldots,M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \rightarrow P\{M'/x\} & \\
\quad \text{if } g(M_1,\ldots,M_n) = M' & \text{(Red Destr 1)} \\
\mathit{let}\ x = g(M_1,\ldots,M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \rightarrow Q & \\
\quad \text{if } g(M_1,\ldots,M_n) \text{ is not defined} & \text{(Red Destr 2)} \\
\mathit{let}\ x = M\ \mathit{in}\ P \rightarrow P\{M/x\} & \text{(Red Let)} \\
\mathit{if}\ M = M\ \mathit{then}\ P\ \mathit{else}\ Q \;\rightarrow\; P & \text{(Red Cond 1)} \\
\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q \;\rightarrow\; Q & \\
\quad \text{if } M \neq N & \text{(Red Cond 2)} \\
P \;\rightarrow\; Q \;\Rightarrow\; P \mid R \;\rightarrow\; Q \mid R & \text{(Red Par)} \\
P \;\rightarrow\; Q \;\Rightarrow\; (\nu a)P \;\rightarrow\; (\nu a)Q & \text{(Red Res)} \\
P' \equiv P,\, P \;\rightarrow\; Q,\, Q \equiv Q' \;\Rightarrow\; P' \;\rightarrow\; Q' & \text{(Red} \equiv\text{)}
\end{array}
$$

Figure 2: Structural congruence and reduction

computation, by creating fresh names and receiving terms in messages. In order to represent that Q may initially know complex terms rather than just names, we may let P begin with the output of these terms on a public channel $c \in S$, so the restriction that S is a set of names entails no loss of generality.

Definition 1. Let S be a finite set of names. The closed process Q is a S-adversary if and only if $\mathit{fn}(Q) \subseteq S$. The closed process P preserves the secrecy of M from S if and only if $P \mid Q$ does not output M on c for any S-adversary Q and any $c \in S$.

If P preserves the secrecy of M from S, then it clearly cannot output M on some $c \in S$, that is, on one of the channels known to the adversary. 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 $c \in S$.

4. THE TYPE SYSTEM

This section presents a general type system for our process calculus; the following sections include instances of this general type system. Figure 3 gives the rules of the type system. In the rules, the metavariable u ranges over names and variables, and T over types. The rules concern three judgments:

- $E \vdash \diamond$ means that E is a well-formed typing environment.
- $E \vdash M : T$ means that M is a term of type T in the environment E.
- $E \vdash P$ says that the process P is well-typed in the environment E.

The type system is parameterized by a set of types *Types* and a non-empty subset $T_{\mathrm{Public}} \subseteq \mathit{Types}$. These parameters will be fixed in each instance of the type system. Always, T_{Public} is the set of types of data that can be known by the attacker.

The type system relies on a function $\mathit{conveys} : \mathit{Types} \rightarrow \mathcal{P}(\mathit{Types})$ such that:

(P0) If $T \in T_{\mathrm{Public}}$, then $\mathit{conveys}(T) = T_{\mathrm{Public}}$.

Intuitively, $\mathit{conveys}(T)$ is the set of types of data that are conveyed by a channel of type T. (It is empty when elements of T cannot be used as channels.) Data conveyed by a public channel is public, since the adversary can obtain it by listening on the channel. Conversely, public data can appear on a public channel, since the adversary can send it.

The type system also relies on a partial function from types to types $O_f : \mathit{Types}^n \rightarrow \mathit{Types}$ for each constructor f of arity n, and a function from types to sets of types $O_g : \mathit{Types}^n \rightarrow \mathcal{P}(\mathit{Types})$ for each destructor g of arity n. These operators O_f and O_g give the types of constructor and destructor applications. These applications need not have unique or most general types (but terms do have unique types in a given environment). Constructors and destructors can accept arguments of different types, and return results whose types depend on the types of the arguments. In this sense, we may say that they are overloaded functions; this overloading subsumes some forms of subtyping and parametric polymorphism. We require the following properties:

(P1) If for all $i \in \{1,\ldots,n\}$, $T_i \in T_{\mathrm{Public}}$, then $O_f(T_1, \ldots, T_n)$ is defined and $O_f(T_1,\ldots,T_n) \in T_{\mathrm{Public}}$.

(P2) If for all $i \in \{1,\ldots,n\}$, $T_i \in T_{\mathrm{Public}}$ and $T \in O_g(T_1,\ldots,T_n)$, then $T \in T_{\mathrm{Public}}$.

(P3) For each equation $g(M_1,\ldots,M_n) = M$ in $\mathrm{def}(g)$, if for all $i \in \{1,\ldots,n\}, E \vdash M_i : T_i$, then there exists $T \in O_g(T_1,\ldots,T_n)$ such that $E \vdash M : T$.

Well-formed environments:

$$\frac{}{\emptyset \vdash \diamond} \qquad \text{(Env } \emptyset\text{)}$$

$$\frac{E \vdash \diamond \qquad u \notin \mathit{dom}(E)}{E, u : T \vdash \diamond} \qquad \text{(Env atom)}$$

Terms:

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

$$\frac{E \vdash \diamond \qquad \forall i \in \{1, \dots, n\}, E \vdash M_i : T_i \qquad O_f(T_1, \dots, T_n) \text{ is defined}}{E \vdash f(M_1, \dots, M_n) : O_f(T_1, \dots, T_n)} \qquad \text{(Constructor application)}$$

Processes:

$$\frac{E \vdash M : T \qquad E \vdash N : T' \qquad T' \in \mathit{conveys}(T) \qquad E \vdash P}{E \vdash \overline{M}\langle N \rangle.P} \qquad \text{(Output)}$$

$$\frac{E \vdash M : T \qquad \forall T' \in \mathit{conveys}(T), E, x : T' \vdash P}{E \vdash M(x).P} \qquad \text{(Input)}$$

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

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

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

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

$$\frac{\forall i \in \{1, \dots, n\}, E \vdash M_i : T_i \qquad \forall T \in O_g(T_1, \dots, T_n), E, x : T \vdash P \qquad E \vdash Q}{E \vdash \mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q} \qquad \text{(Destructor application)}$$

$$\frac{E \vdash M : T \qquad E, x : T \vdash P}{E \vdash \mathit{let}\ x = M\ \mathit{in}\ P} \qquad \text{(Local definition)}$$

$$\frac{E \vdash M : T \qquad E \vdash N : T' \qquad \text{if } T = T' \text{ then } E \vdash P \qquad E \vdash Q}{E \vdash \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q} \qquad \text{(Cond)}$$

Figure 3: Type rules

The first two properties reflect that the result of applying a function to public terms should also be public, since the adversary can compute it. The third property essentially says that the definition of O_g on types is compatible with the definition of g on terms.

The type rules for nil, parallel composition, replication, restriction, and local definition are standard. 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 a form of polymorphism: the type of a can change according to the environment.

By the rule (Output), the process $\overline{M}\langle N\rangle.P$ is well-typed only if data of the type T' of N can be conveyed on a channel of the type T of M, that is, $T' \in \mathit{conveys}(T)$. Conversely, for typechecking the process $M(x).P$ via the rule (Input), the variable x is considered with all types $T' \in \mathit{conveys}(T)$ where T is the type of M. The universal quantification on the type of x is unusual; it arises because a channel may convey data of several types. In security protocols, this flexibility is important because a channel may convey data from the adversary and from honest participants, and types can help distinguish these two cases.

The rule (Constructor application) types $f(M_1, \ldots, M_n)$ according to the corresponding operator O_f. The rule (Destructor application) is similar to (Input); in $\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$, the variable x is considered with all the possible types of $g(M_1, \ldots, M_n)$, that is, all elements of $O_g(T_1, \ldots, T_n)$.

Rule (Cond) exploits the property that if two terms M and N have different types then they are certainly different. In this case, $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ may be well-typed without P being well-typed.

The constructs $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ and $\mathit{let}\ x = M\ \mathit{in}\ P$ can be defined as special cases from $\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$, and their typing follows:

- Let *equals* be a binary destructor with $\mathit{equals}(M, M) = M$ (and with $\mathit{equals}(M, N)$ undefined otherwise), $O_{\mathit{equals}}(T, T) = \{T\}$, and $O_{\mathit{equals}}(T, T') = \emptyset$ if $T \neq T'$. Then $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ can be defined and typed as $\mathit{let}\ x = \mathit{equals}(M, N)\ \mathit{in}\ P\ \mathit{else}\ Q$, where $x \notin \mathit{fv}(P)$.
- Let *id* be a unary destructor with $\mathit{id}(M) = M$ and $O_{\mathit{id}}(T) = \{T\}$. Then $\mathit{let}\ x = M\ \mathit{in}\ P$ can be defined and typed as $\mathit{let}\ x = \mathit{id}(M)\ \mathit{in}\ P\ \mathit{else}\ 0$.

Because of these encodings, we may omit the cases of $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ and $\mathit{let}\ x = M\ \mathit{in}\ P$ in various arguments and proofs. The encodings also suggest that the typing rule (Cond) for $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ is more natural than might seem at first sight.

5. PROPERTIES OF THE TYPE SYSTEM

The subject-reduction lemma says that typing is preserved by computation.

Lemma 1 (Subject reduction) *If $E \vdash P$ and $P \rightarrow Q$ then $E \vdash Q$.*

The secrecy theorem says that if a closed process P is well-typed in an environment E, and a name s is not of a type in T_{Public} according to E, then P preserves the secrecy of s from S, where S is the set of names that are of a type in T_{Public} according to E. In other words, P preserves the secrecy of names whose type is not in T_{Public} against adversaries that can output, input, and compute on names of types in T_{Public}.

Theorem 1 (Secrecy) *Let P be a closed process. Suppose that $E \vdash P$, $E \vdash s : T'$, and $T' \notin T_{\mathit{Public}}$. Let $S = \{a \mid E \vdash a : T \text{ and } T \in T_{\mathit{Public}}\}$. Then P preserves the secrecy of s from S.*

This secrecy theorem is a consequence of the subject-reduction lemma and the typability of the adversary (i.e., that any S-adversary Q can be typed with all names and variables of a type in T_{Public}).

6. A SIMPLE INSTANCE OF THE TYPE SYSTEM

As an example, we show how the type system applies to symmetric and asymmetric encryption. The resulting instance of the type system is similar in scope and power to a previous special-purpose type system [4], but treats additional constructs and could easily treat even more. For this instance, the grammar of types is given in Figure 4. Informally, types have the following meanings:

- Public is the type of public data.
- Secret is the type of secret data.
- $T_1 \times \ldots \times T_n$ is the type of tuples, whose components are of types $T_1, \ldots, T_n$.
- $\mathrm{C}[T]$ is the type of a channel that can convey data of type T and that cannot be known by the adversary. (Channels that can be known by the adversary are of type Public.)
- $\mathrm{K}^{\mathrm{Secret}}[T]$ is the type of symmetric keys that can be used to encrypt data of type T and that cannot be known by the adversary. (Symmetric keys that can be known by the adversary are of type Public.)
- $\mathrm{EK}^{\mathrm{Secret}}[T]$ is the type of secret asymmetric encryption keys that can be used to encrypt cleartexts of type T.
- $\mathrm{DK}^{\mathrm{Secret}}[T]$ is the type of asymmetric decryption keys for cleartexts of type T and such that the corresponding encryption keys are secret. These decryption keys are also secret.
- $\mathrm{EK}^{\mathrm{Public}}[T]$ is the type of public asymmetric encryption keys that can be used to encrypt cleartexts of type T. The adversary can use these keys to encrypt its messages, so public messages can also be encrypted under these keys.
- $\mathrm{DK}^{\mathrm{Public}}[T]$ is the type of asymmetric decryption keys for cleartexts of type T and such that the corresponding encryption keys are public. These decryption keys are however secret. When decrypting a message with such a key, the result can be of type T (in normal use of the key) or of type Public (when the adversary has used the corresponding encryption key to encrypt one of its messages).

$T ::=$		types
	Public	public data
	Secret	secret data
	$T_1 \times \ldots \times T_n$	tuple
	$\mathrm{C}[T]$	secret channel
	$\mathrm{K}^{\mathrm{Secret}}[T]$	secret shared key
	$\mathrm{DK}^{\mathrm{Secret}}[T]$	decryption key whose corresponding encryption key is secret
	$\mathrm{EK}^{\mathrm{Secret}}[T]$	secret encryption key
	$\mathrm{DK}^{\mathrm{Public}}[T]$	decryption key whose corresponding encryption key is public
	$\mathrm{EK}^{\mathrm{Public}}[T]$	public encryption key

Figure 4: Grammar of types in an instance of the type system

$$\begin{aligned} T_{\mathrm{Public}} &= \{T \mid T \leq \mathrm{Public}\} \\ &= \{\mathrm{Public}, \mathrm{EK}^{\mathrm{Public}}[T]\} \cup \{T_1 \times \ldots \times T_n \mid \forall i \in \{1, \ldots, n\}, T_i \in T_{\mathrm{Public}}\} \end{aligned}$$

If $T \leq \mathrm{Public}$, then $\mathit{conveys}(T) = T_{\mathrm{Public}}$;
$\mathit{conveys}(\mathrm{C}[T]) = \{T' \mid T' \leq T\}$.

$O_{ntuple}(T_1, \ldots, T_n) = T_1 \times \ldots \times T_n$.
If $T_1 \leq \mathrm{Public}$ and $T_2 \leq \mathrm{Public}$, then $O_{sencrypt}(T_1, T_2) = \mathrm{Public}$;
if $T' \leq T$, then $O_{sencrypt}(T', \mathrm{K}^{\mathrm{Secret}}[T]) = \mathrm{Public}$.
If $T_1 \leq \mathrm{Public}$ and $T_2 \leq \mathrm{Public}$, then $O_{pencrypt}(T_1, T_2) = \mathrm{Public}$;
if $T' \leq T$, then $O_{pencrypt}(T', \mathrm{EK}^{L}[T]) = \mathrm{Public}$.
If $T_1 \leq \mathrm{Public}$, then $O_{pk}(T_1) = \mathrm{Public}$;
$O_{pk}(\mathrm{DK}^{L}[T]) = \mathrm{EK}^{L}[T]$.

$O_{ith_n}(T_1 \times \ldots \times T_n) = \{T_i\}$.
If $T \leq \mathrm{Public}$, then $O_{sdecrypt}(\mathrm{Public}, T) = T_{\mathrm{Public}}$;
$O_{sdecrypt}(\mathrm{Public}, \mathrm{K}^{\mathrm{Secret}}[T]) = \{T' \mid T' \leq T\}$.
If $T \leq \mathrm{Public}$, then $O_{pdecrypt}(\mathrm{Public}, T) = T_{\mathrm{Public}}$;
$O_{pdecrypt}(\mathrm{Public}, \mathrm{DK}^{\mathrm{Secret}}[T]) = \{T' \mid T' \leq T\}$;
$O_{pdecrypt}(\mathrm{Public}, \mathrm{DK}^{\mathrm{Public}}[T]) = \{T' \mid T' \leq T\} \cup T_{\mathrm{Public}}$.

Other cases: $\mathit{conveys}(T) = \emptyset$, $O_f(T_1, \ldots, T_n)$ is not defined, and $O_g(T_1, \ldots, T_n) = \emptyset$.

Figure 5: Definition of T_{Public} and type operators in an instance of the type system

We define T_{Public} and the type operators of the system in Figure 5. For this purpose, we let the subtyping relation $\leq$ be reflexive and transitive, with

$$\mathrm{C}[T] \leq \text{Secret}, \mathrm{K}^{\text{Secret}}[T] \leq \text{Secret}, \mathrm{DK}^{\text{Secret}}[T] \leq \text{Secret},$$
$$\mathrm{EK}^{\text{Secret}}[T] \leq \text{Secret}, \mathrm{DK}^{\text{Public}}[T] \leq \text{Secret},$$
$$\mathrm{EK}^{\text{Public}}[T] \leq \text{Public}, \text{Public} \times \ldots \times \text{Public} \leq \text{Public},$$
$$\text{if } \exists i \in \{1, \ldots, n\}, T_i = \text{Secret then } T_1 \times \ldots \times T_n \leq \text{Secret},$$
$$\text{if } T_1 \leq T'_1, \ldots, T_n \leq T'_n \text{ then } T_1 \times \ldots \times T_n \leq T'_1 \times \ldots \times T'_n.$$

(We do not adopt a subsumption rule.) Importantly, these definitions allow encryption under a public key of type $\mathrm{EK}^{\text{Public}}[T]$ to accept data both of type Public and of type T. For the corresponding decryption, we handle both cases: $O_{pdecrypt}(\text{Public}, \mathrm{DK}^{\text{Public}}[T])$ includes both subtypes of T and subtypes of Public. (A similar idea appears in the special-purpose type system mentioned above [4].)

Proposition 1 *These definitions satisfy the constraints of the general type system (P0, P1, P2, P3).*

For example, the type system can be used to prove that s remains secret in the example protocol of section 2. For this proof, we define $E \triangleq s : \text{Secret}, e : \text{Public}$, and derive $E \vdash P$. In the (Restriction) rule, we choose the types

$$T_{sK_A} \triangleq \mathrm{DK}^{\text{Public}}[\text{Secret} \times \mathrm{K}^{\text{Secret}}[\text{Secret}]]$$

for sK_A and

$$T_{sK_B} \triangleq \mathrm{DK}^{\text{Public}}[\text{Secret} \times \mathrm{EK}^{\text{Public}}[\text{Secret} \times \mathrm{K}^{\text{Secret}}[\text{Secret}]]]$$

for sK_B. Then $pk(sK_A)$ has the type

$$T_{pK_A} \triangleq O_{pk}(T_{sK_A}) = \mathrm{EK}^{\text{Public}}[\text{Secret} \times \mathrm{K}^{\text{Secret}}[\text{Secret}]]$$

and $pk(sK_B)$ has the type

$$T_{pK_B} \triangleq O_{pk}(T_{sK_B})$$
$$= \mathrm{EK}^{\text{Public}}[\text{Secret} \times \mathrm{EK}^{\text{Public}}[\text{Secret} \times \mathrm{K}^{\text{Secret}}[\text{Secret}]]]$$

The remainder of the process is typed in the environment:

$$E' \triangleq E, sK_A : \mathrm{DK}^{\text{Public}}[\text{Secret} \times \mathrm{K}^{\text{Secret}}[\text{Secret}]],$$
$$sK_B : \mathrm{DK}^{\text{Public}}[\text{Secret} \times \mathrm{EK}^{\text{Public}}[\text{Secret} \times \mathrm{K}^{\text{Secret}}[\text{Secret}]]],$$
$$pK_A : \mathrm{EK}^{\text{Public}}[\text{Secret} \times \mathrm{K}^{\text{Secret}}[\text{Secret}]],$$
$$pK_B : \mathrm{EK}^{\text{Public}}[\text{Secret} \times \mathrm{EK}^{\text{Public}}[\text{Secret} \times \mathrm{K}^{\text{Secret}}[\text{Secret}]]]$$

We check that $T_{pK_A} \in \textit{conveys}(\text{Public})$ and $T_{pK_B} \in \textit{conveys}(\text{Public})$ (since these types are subtypes of Public). Then we only have to show that $E' \vdash A$ and $E' \vdash B$. In the typing of A, we choose k of type Secret. Then

$$E', k : \text{Secret} \vdash \textit{pencrypt}((k, pK_A), pK_B) : \text{Public}$$

follows by (Constructor application), hence the output $\overline{e}\langle \textit{pencrypt}((k, pK_A), pK_B)\rangle$ is well-typed by (Output). In the input $e(z)$, by (Input), z can be of any subtype of Public, then by (Destructor application), we have to prove $E', k : \text{Secret}, x : T_x, y : T_y \vdash \textit{if } x = k \textit{ then } \overline{e}\langle \textit{sencrypt}(s, y)\rangle$, where either $T_x \leq \text{Secret}$ and $T_y \leq \mathrm{K}^{\text{Secret}}[\text{Secret}]$ or $T_x \leq \text{Public}$ and $T_y \leq \text{Public}$.

- In the first case, the conditional is well-typed, since the output is well-typed.
- In the second case, the conditional is well-typed, since x and k cannot have the same type.

We leave to the reader the verification that B is well-typed. Finally, by Theorem 1, P preserves the secrecy of s from $\{e\}$.

7. PROTOCOL CHECKER

In this section we give a precise definition of a protocol checker based on untyped logic programs, then study its properties, in particular proving its equivalence to the type system. This equivalence is considerably less routine and predictable than properties such as subject reduction (Lemma 1).

7.1 Definition of the Protocol Checker

Given a closed process P_0 and a set of names S, the protocol checker builds a set of rules, in the form of Horn clauses. We assume that each restriction $(\nu a)P$ in P_0 has a different name a, and that this name is different from any free name of P_0.

The rules use two predicates: attacker and message. The fact attacker(p) means that the attacker may have p, and the fact message(p, p') means that the message p' may appear on channel p.

$F ::=$	facts
attacker(p)	attacker knowledge
message(p, p')	channel messages

Here p and p' range over patterns (or "terms", but we prefer the word "patterns" in order to avoid confusion), which are generated by the following grammar:

$p ::=$	patterns
x, y, z	variable
$a[p_1, \ldots, p_n]$	name
$f(p_1, \ldots, p_n)$	constructor application

For each name a in P_0 we have a corresponding pattern construct $a[p_1, \ldots, p_n]$. We treat a as a function symbol, and write $a[p_1, \ldots, p_n]$ rather than $a(p_1, \ldots, p_n)$ only for clarity. If a is a free name, then the arity of this function is 0. If a is bound by a restriction $(\nu a)P$ in P_0, then this arity is the number of input statements above the restriction $(\nu a)P$ in the abstract syntax tree of P_0. Thus, in the checker, a new name behaves as a function of the inputs that take place (lexically) before its creation. Therefore, we distinguish names only when they are created after receiving different inputs. In contrast, a restriction in a process always generates fresh names; hence the rules will not exactly reflect the operational semantics of processes, but this approximation is useful for automation and harmless in most examples. As we show below, this approximation is also compatible with soundness and completeness theorems that prove the equivalence between the type system and the logic-programming system.

The rules comprise rules for the attacker and rules for the protocol. Next we define these two kinds.

7.1.1 *Rules for the attacker*

Initially, the attacker has all the names in a set S, hence the rules attacker$(a[])$ for each $a \in S$. Moreover, the abilities of the attacker are represented by the following rules:

For each constructor f of arity n,

$$\text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \qquad \text{(Rf)}$$
$$\Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$$

For each destructor g,

for each equation $g(M_1, \ldots, M_n) = M$ in $\text{def}(g)$, (Rg)

$$\text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M)$$

$$\text{message}(x, y) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(y) \qquad \text{(Rl)}$$

$$\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{message}(x, y) \qquad \text{(Rs)}$$

The rules (Rf) and (Rg) mean that the attacker can apply all operations to all terms it has, (Rf) for constructors, (Rg) for destructors. The set of these rules is finite if the set of constructors and each of the sets $\text{def}(g)$ is finite; handling this set is easiest in this finite case. For (Rg), notice that equations in $\text{def}(g)$ do not have free names and that terms without free names are also patterns, so the rules have the required format. Rule (Rl) means that the attacker can listen on all the channels it has, and (Rs) that it can send all the messages it has on all the channels it has.

7.1.2 Rules for the protocol

When a function ρ associates a pattern with each name and variable, and f is a constructor, we extend ρ as a substitution by $\rho(f(M_1, \ldots, M_n)) = f(\rho(M_1), \ldots, \rho(M_n))$.

The translation $[\![P]\!]\rho h$ of a process P is a set of rules, where ρ is a function which associates a pattern with each name and variable, and h is a sequence of facts of the form $\text{message}(p, p')$. The empty sequence is denoted by $\emptyset$; the concatenation of a fact F to the sequence h is denoted by $h \wedge F$.

- $[\![0]\!]\rho h = \emptyset$,
- $[\![P \mid Q]\!]\rho h = [\![P]\!]\rho h \cup [\![Q]\!]\rho h$,
- $[\![!P]\!]\rho h = [\![P]\!]\rho h$,
- $[\![(\nu a)P]\!]\rho h = [\![P]\!](\rho[a \mapsto a[p'_1, \ldots, p'_n]])h$ if $h = \text{message}(p_1, p'_1) \wedge \ldots \wedge \text{message}(p_n, p'_n)$,
- $[\![M(x).P]\!]\rho h = [\![P]\!](\rho[x \mapsto x])(h \wedge \text{message}(\rho(M), x))$,
- $[\![\overline{M}\langle N\rangle.P]\!]\rho h = [\![P]\!]\rho h \cup \{h \Rightarrow \text{message}(\rho(M), \rho(N))\}$,
- $[\![\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho h = \cup\{[\![P]\!]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma h) \mid g(p'_1, \ldots, p'_n) = p'$ is in $\text{def}(g)$ and (σ, σ') is a most general pair of substitutions such that $\sigma\rho(M_1) = \sigma' p'_1, \ldots, \sigma\rho(M_n) = \sigma' p'_n\} \cup [\![Q]\!]\rho h$.

Thus, the translation of a process is, very roughly, a set of rules that enable us to prove that it sends certain messages. The sequence h keeps track of messages received by the process, since these may trigger other messages. The translation of 0 is the empty set, because this process does nothing. The translation of a parallel composition $P \mid Q$ is the union of the translations of P and Q, because $P \mid Q$ sends the messages of P and Q plus any messages that result from the interaction of P and Q. Replication is ignored, because the target logic is classical, so all logical rules are applicable arbitrarily many times. For restriction, we replace the restricted name a in question with a pattern $a[\ldots]$ that depends on the messages received, as recorded in the sequence h. This sequence is extended in the translation of an input, with the input in question. On the other hand, the translation of an output adds a clause; this clause represents that reception of the messages in h can trigger the output in question. Finally, the translation of a destructor application takes the union of the clauses for the case where the destructor succeeds (with an appropriate substitution) and those for the case where the destructor fails; thus we avoid having to determine whether it will succeed or fail.

7.1.3 Summary and secrecy results

Let $\rho = \{a \mapsto a[] \mid a \in \mathit{fn}(P_0)\}$. We define the rule base corresponding to process P_0 as:

$$B_{P_0,S} = [\![P_0]\!]\rho\emptyset \cup \{\text{attacker}(a[]) \mid a \in S\}$$
$$\cup \{(\text{Rf}), (\text{Rg}), (\text{Rl}), (\text{Rs})\}$$

We have the following secrecy result. Let $s \in \mathit{fn}(P_0)$. If $\text{attacker}(s[])$ cannot be derived from $B_{P_0,S}$, then P_0 preserves the secrecy of s from S. This result is the basis for a method for proving secrecy properties. Of course, whether a fact can be derived from $B_{P_0,S}$ may be undecidable, but in practice there exist algorithms that terminate on numerous examples of protocols. In particular, we can use variants of resolution algorithms, such as the algorithm described in [8].

This result can be proved directly. Instead, we establish it by showing that we can build a typing of P_0 in a suitable instance of our general type system; the result then follows from Theorem 1. We also establish a completeness theorem, as a converse: the checker yields the "best" instance of our general type system.

7.2 Correctness

We use the rule base $B_{P_0,S}$ to define an instance of our general type system, as follows.

- The grammar of types is

$T ::=$	types
$a[T_1, \ldots, T_n]$	name
$f(T_1, \ldots, T_n)$	constructor application

  The types are exactly closed patterns.

- $T_{\text{Public}} = \{T \mid \text{attacker}(T)$ is derivable from $B_{P_0,S}\}$ (that is, the protocol checker says that the attacker may have T).
- $\mathit{conveys}(T) = \{T' \mid \text{message}(T, T')$ is derivable from $B_{P_0,S}\}$ (that is, the protocol checker says that the channel T may convey T').
- $O_f(T_1, \ldots, T_n) = f(T_1, \ldots, T_n)$.
- $O_g(T_1, \ldots, T_n) = \{\sigma M \mid$ there exists an equation $g(M_1, \ldots, M_n) = M$ in $\text{def}(g)$, σ maps variables to types, and $\forall i \in \{1, \ldots, n\}, \sigma M_i = T_i\}$.

We can show that the type system as defined above satisfies the constraints (P0, P1, P2, P3) of the general type system, and that $E \vdash P_0$, where $E = \{a : a[] \mid a \in \mathit{fn}(P_0)\}$. The secrecy theorem for the protocol checker then follows from the secrecy theorem for the general type system (Theorem 1):

Theorem 2 (Secrecy) *Let P_0 be a closed process and $s \in \mathit{fn}(P_0)$. If $\text{attacker}(s[])$ cannot be derived from $B_{P_0,S}$, then P_0 preserves the secrecy of s from S.*

7.3 Completeness

The protocol checker is incomplete in the sense that it fails to prove some true properties. However, as the next theorem states, the protocol checker is relatively complete: it is as complete as the type system of section 4.

Theorem 3 (Completeness) *Let P_0 be a closed process, s a name, and S a set of names. Suppose that an instance of the general type system proves (by Theorem 1) that P_0 preserves the secrecy of s from S. Then* attacker(s[]) *cannot be derived from $B_{P_0,S}$, so the protocol checker also proves that P_0 preserves the secrecy of s from S.*

This theorem is proved by establishing a correspondence between types T of an instance of the general type system and types T_c of the checker (of section 7.2): we define a partial function ϕ that maps T_c to T. (The appendix gives the definition of ϕ.) Then we prove that all rules of $B_{P_0,S}$ are satisfied, in the following sense:

- The closed fact attacker(T_c) is satisfied if $\phi(T_\mathrm{c}) \in T_{\mathrm{Public}}$.
- The closed fact message($T_\mathrm{c}, {T'}_\mathrm{c}$) is satisfied if $\phi({T'}_\mathrm{c}) \in$ *conveys*$(\phi(T_\mathrm{c}))$.
- The rule $F_1 \wedge \ldots \wedge F_n \Rightarrow F$ is satisfied if σF is satisfied for every closed substitution σ such that $\forall i \in \{1, \ldots, n\}, \sigma F_i$ is satisfied.

Therefore, all facts derived from $B_{P_0,S}$ are satisfied. Moreover, if s is proved secret by the instance of the general type system, then attacker(s[]) is not satisfied. (If attacker(s[]) were satisfied, we would also have that $\phi(s[]) \in T_{\mathrm{Public}}$, so the instance of the general type system would not be able to prove the secrecy of s.) Hence, attacker(s[]) cannot be derived from $B_{P_0,S}$. The result follows.

This completeness result shows the power of the protocol checker. This power is not only theoretical: it has been demonstrated in practice on several examples [8], from simple protocols like variants of the Needham-Schroeder protocols [32] to Skeme [25].

The completeness result does not however mean that the protocol checker constitutes the only useful instance of the general type system. In particular, simpler instances are easier to use in manual reasoning. Presenting those instances by type rules (rather than logic programs) is often quite convenient. Moreover, the checker does not always terminate, in particular when it tries to establish properties of an infinite family of types; in other instances of the type system, we may merge those types (obtaining some finite proofs at the cost of completeness). Similarly, the (rare) case where a set def(g) is large or infinite is more problematic for the checker than for the general type system. Finally, the general type system may be combined with other type-based analyses for proving protocol properties other than secrecy (e.g., as in [20]).

8. CONCLUSION

This paper makes two main contributions:

(1) a type system for expressing and proving secrecy properties of security protocols with a generic treatment of many cryptographic operations;

(2) a tight relation between two useful but superficially quite different approaches to protocol analysis, respectively embodied in the type system and in a logic-programming tool.

The first contribution can be seen as the continuation of a line of work on static analyses for security, discussed in the introduction. So far, those static analyses have been developed successfully but often in ad hoc ways. We believe that type systems such as ours not only are useful in examples but also shed light on the constraints and the design space for static analyses.

In the last few years, there has been a vigorous proliferation of frameworks and techniques for reasoning about security protocols. Their relations are seldom explicit or obvious. Moreover, little is known about how to combine techniques. The second contribution is part of a broader effort to understand those relations. Previous work (in particular [18]) suggests connections between (untyped) process calculi and logic-programming notations for protocols; we go further by relating proof methods in those two worlds. Such connections are perhaps the start of a healthy consolidation.

9. REFERENCES

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

[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. Volume for the 20th International Summer School on Foundations of Secure Computation, held in Marktoberdorf, Germany (1999).

[3] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 147–160, Jan. 1999.

[4] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, Apr. 2001.

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

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

[7] R. M. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In C. Palamidessi, editor, *CONCUR 2000: Concurrency Theory (11th International Conference)*, volume 1877 of *Lecture Notes in Computer Science*, pages 380–394. Springer-Verlag, Aug. 2000.

[8] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer*

Security Foundations Workshop (CSFW-14), pages 82–96, June 2001.

[9] C. Bodei. *Security Issues in Process Calculi*. PhD thesis, Università di Pisa, Jan. 2000.

[10] C. Bodei, P. Degano, F. Nielson, and H. 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, Sept. 1998.

[11] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In C. Palamidessi, editor, *CONCUR 2000: Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 365–379. Springer-Verlag, Aug. 2000.

[12] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW'99)*, pages 55–69, June 1999.

[13] M. Dam. Proving trust in systems of second-order processes. In *Proceedings of the 31th Hawaii International Conference on System Sciences*, volume VII, pages 255–264, 1998.

[14] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi. A new algorithm for the automatic verification of authentication protocols: From specifications to flaws and attack scenarios. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, Rutgers University, New Jersey, Sept. 1997.

[15] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols*, Indianapolis, Indiana, 25 June 1998.

[16] A. Durante, R. Focardi, and R. Gorrieri. CVS: A compiler for the analysis of cryptographic protocols. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW'99)*, pages 203–212, June 1999.

[17] N. Durgin, J. Mitchell, and D. Pavlovic. A compositional logic for protocol correctness. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 241–255, June 2001.

[18] N. A. Durgin and J. C. Mitchell. Analysis of security protocols. In M. Broy and R. Steinbruggen, editors, *Calculational System Design*, pages 369–395. IOS Press, 1999.

[19] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9), Sept. 1997.

[20] A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 145–159, June 2001.

[21] N. Heintze and J. 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.

[22] M. Hennessy and J. 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.

[23] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In G. Smolka, editor, *Programming Languages and Systems: Proceedings of the 9th European Symposium on Programming (ESOP 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 2000.

[24] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, Spring 1994.

[25] H. Krawczyk. SKEME: A versatile secure key exchange mechanism for internet. In *Proceedings of the Internet Society Symposium on Network and Distributed Systems Security*, Feb. 1996. Available at `http://bilbo.isu.edu/sndss/sndss96.html`.

[26] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, pages 112–121, 1998.

[27] G. 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.

[28] C. Meadows. Panel on languages for formal specification of security protocols. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, page 96, 1997.

[29] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[30] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, Jan. 1973.

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

[32] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.

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

[34] E. Sumii and B. C. Pierce. Logical relations and encryption (Extended abstract). In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 256–269, June 2001.

[35] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, 1996.

[36] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 314–328. Springer-Verlag, July 1999.

APPENDIX

Function ϕ for the completeness proof

This appendix describes the construction of a function ϕ on types that plays a role in the completeness proof, as explained in the main body of the paper.

In order to prove completeness, we consider a closed process P_0, a name s, and a set of names S. We consider an instance of the general type system, and assume that this instance proves (by Theorem 1) that P_0 preserves the secrecy of s from S. That is, we assume that, in this instance, there exists an environment E_0 such that $E_0 \vdash P_0$, $E_0 \vdash s : T$ with $T \notin T_{\text{Public}}$, and $S = \{a \mid E_0 \vdash a : T \text{ and } T \in T_{\text{Public}}\}$. Without loss of generality, we may assume that E_0 contains only names. We fix a proof of $E_0 \vdash P_0$ for the rest of this argument.

Intuitively, a well-chosen environment for a subprocess P of P_0 is an environment that can be used to type P in a "standard" proof that P_0 is well-typed, using the type system associated with the protocol checker in section 7.2. A "standard" proof is one in which types introduced by the rule (Restriction) for $(\nu a)Q$ are of the form $a[T_{c1}, \dots, T_{cn}]$, where $T_{c1}, \dots, T_{cn}$ are the types of the variables bound by inputs above $(\nu a)Q$ in P_0's syntax tree.

A $(T_{c1}, \dots, T_{cn})$-well-chosen environment for P is similar, except that the parameters $(T_{c1}, \dots, T_{cn})$ indicate which types should be chosen for the variables bound by inputs. Note that a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for P does not always exist, for example when the number of parameters $(T_{c1}, \dots, T_{cn})$ does not correspond to the number of variables bound by inputs above P in P_0.

Definition 2. Let $T_{c1}, \dots, T_{cn}$ be closed patterns. A $(T_{c1}, \dots, T_{cn})$-well-chosen environment for an occurrence of a subprocess of P_0 is defined as follows:

- A ()-well-chosen environment for P_0 is $\rho_0 = \{a \mapsto a[] \mid (a : T) \in E_0\}$.
- If E_c is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for $\overline{M}\langle N\rangle.P$, then E_c is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for P.
- If E_c is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for $M(x).P$, then $E_c[x \mapsto T_{cn+1}]$ is a $(T_{c1}, \dots, T_{cn}, T_{cn+1})$-well-chosen environment for P.
- If E_c is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for $P \mid Q$, then E_c is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for P and Q.
- If E_c is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for $!P$, then E_c is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for P.
- If E_c is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for $(\nu a)P$, then $E_c[a \mapsto a[T_{c1}, \dots, T_{cn}]]$ is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for P.
- If E_c is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for *let* $x = g(M_1, \dots, M_n)$ *in* P *else* Q, then E_c is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for Q, and if in addition there exists an equation $g(M'_1, \dots, M'_n) = M'$ in $\text{def}(g)$ and a substitution σ such that for all $i \in \{1, \dots, n\}$, $\sigma M'_i = E_c(M_i)$, then $E_c[x \mapsto \sigma M']$ is a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for P.

Finally, the function ϕ is defined so that if a type T_c appears in a standard proof that P_0 is well-typed using the type system associated with the protocol checker in section 7.2, then $\phi(T_c)$ appears in the corresponding place in the proof of $E_0 \vdash P_0$ in the instance of the general type system under consideration.

Definition 3. The partial function $\phi : T_c \to T$ from types of the protocol checker to types of the instance of the general type system is defined by induction on the term T_c:

- $\phi(f(T_{c1}, \dots, T_{cn})) = O_f(\phi(T_{c1}), \dots, \phi(T_{cn}))$. (Therefore, $\phi(f(T_{c1}, \dots, T_{cn}))$ is undefined if $O_f(\phi(T_{c1}), \dots, \phi(T_{cn}))$ is undefined.)
- If $E_0 \vdash a : T$, then $\phi(a[]) = T$.
- When a is bound by a restriction in P_0, we define $\phi(a[T_{c1}, \dots, T_{cn}])$ as follows. Without loss of generality, we may assume that a is bound exactly once. Let P be the process such that $(\nu a)P$ is a subprocess of P_0. Let E_c be a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for $(\nu a)P$. Let $E = \phi \circ E_c$. Then $\phi(a[T_{c1}, \dots, T_{cn}]) = T'$ where T' is such that $E, a : T' \vdash P$ is a judgment used to prove $E_0 \vdash P_0$. There is at most one such judgment, so T' is unique.

  If a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for $(\nu a)P$ does not exist, or if no suitable judgment $E, a : T' \vdash P$ appears in the proof of $E_0 \vdash P_0$, then $\phi(a[T_{c1}, \dots, T_{cn}])$ is undefined.

This definition is recursive, and we can check that it is well-founded using the following ordering. Names are ordered by $a < b$ if a is bound above b in P_0, or a is free and b is bound in P_0. The ordering on terms is then the lexicographic ordering of pairs containing as first component the multiset of names that appear in the term and as second component the size of the term. In the first case of the definition of ϕ, the first component is constant and the second one decreases. In the third case, the first component decreases: when defining $\phi(a[T_{c1}, \dots, T_{cn}])$, in the recursive calls used to compute $\phi \circ E_c$, the name a at the top of the term has disappeared, and the only names that have appeared due to the computation of the well-chosen environment are free names or names bound above a (therefore names smaller than a).