Analyzing Security Protocols with Secrecy Types and Logic Programs

MARTÍN ABADI
University of California, Santa Cruz
and
BRUNO BLANCHET
CNRS, École Normale Supérieure, Paris

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.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*Security and protection (e.g., firewalls)*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Process models*

General Terms: Languages, Security, Theory, Verification

Additional Key Words and Phrases: Cryptographic protocols, logic programming, process calculi, secrecy properties, typing

1. INTRODUCTION

Concepts and methods from programming languages have long been useful in security (e.g., [Morris 1973]). 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., [Abadi and Fournet 2001; Abadi and Gordon 1999; Amadio and Lugiez 2000; Cervesato et al. 1999; Dam 1998; Durante et al. 1999; Durgin et al. 2001; Focardi and Gorrieri 1997; Lincoln et al. 1998; Meadows 1997; Sumii and Pierce 2001]). 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 (in the sense that they sometimes fail to establish security properties), 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

Authors' addresses: `abadi@cs.ucsc.edu` (Martín Abadi), `blanchet@di.ens.fr` (Bruno Blanchet).
This work was presented at the 29th Annual ACM Symposium on Principles of Programming Languages (2002). A preliminary version of this paper appears in the proceedings of that symposium.

(e.g., [Lowe 1996]) and to human-guided theorem proving (e.g., [Paulson 1998]).

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 popular, 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 [Abadi 1999; Abadi and Blanchet 2003b; Cardelli et al. 2000; Gordon and Jeffrey 2001; 2002; Hennessy and Riely 2000; Honda et al. 2000]. A related approach relies on control-flow analysis [Bodei et al. 1998]; it has an algorithmic emphasis, but it is roughly equivalent to typing at least in important special cases [Bodei 2000]. Such static analyses have applications in a broader security context (e.g., [Abadi et al. 1999; Heintze and Riecke 1998; Myers 1999; Volpano et al. 1996]); 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, and they are analyzed symbolically with general provers [Denker et al. 1998; Weidenbach 1999] or with special-purpose algorithms and tools [Debbabi et al. 1997; Cervesato et al. 1999; Compton and Dexter 1999; Blanchet 2001; Selinger 2001; Goubault-Larrecq 2002; Blanchet 2002; Blanchet and Podelski 2003; Comon-Lundh and Cortier 2003; Abadi and Blanchet 2003a; Goubault-Larrecq 2004; Goubault-Larrecq et al. 2004; Blanchet 2004]. (See also [Kemmerer et al. 1994] for some of the roots of this approach.) In some of this work [Cervesato et al. 1999; Compton and Dexter 1999], the use of linear logic enables a rather faithful model of protocol state, reducing (or eliminating) the possibility of false alarms; on the other hand, the treatment of protocols with an unbounded number of sessions can become quite difficult. Partly for this reason, and partly because of the familiarity and relative simplicity of classical logic, algorithms and tools that rely on classical logic programs are prevalent. 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 [Milner 1999] 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 [Abadi and Fournet 2001], 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 expect that it can be treated along similar lines but with more difficulty (see Sections 2 and 8). 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), 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 [Blanchet 2001], 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.

Throughout this paper, we use the following concept of secrecy (e.g., [Abadi 2000]): 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. For instance, M may be a cryptographic key; its secrecy means that no adversary can obtain the key by attacking P. Although this property allows the possibility that P reveals partial information about M, the property is attractive and often satisfactory.

For example, consider the following protocol (presented informally here, and studied more rigorously in the body of this paper):

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

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 a public-key encryption function *pencrypt*, on a shared-key encryption function *sencrypt*, and on public keys pK_A for A and pK_B for B. For *pencrypt* and *sencrypt*, the second argument is the encryption key, the first the plaintext being encrypted. 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}. Can an attacker obtain s? The answer to this question may partly depend on delicate points that the informal description of the protocol does not clarify, such as whether a public key can be mistaken for a shared key. Once we address those points through a formal description of the protocol, we can apply our analyses for establishing the secrecy of s or for identifying vulnerabilities.

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 application, Section 6 explains how the type system can be instantiated to

$M, N ::=$	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

Fig. 1. Syntax of the process calculus

handle shared-key and public-key encryption operations. Section 7 formalizes and studies the logic-programming protocol checker. Section 8 discusses an extension (to general equational theories). Section 9 concludes. An appendix contains some proofs.

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. Names represent atomic data items, such as nonces and keys, while variables are formal parameters that can be replaced by any term (atomic or complex). The syntax also assumes a set of symbols for constructors and destructors, each with an arity; 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 $\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$ 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, and all variables of M' occur in $M'_1, \ldots, M'_n$. 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 $i \in \{1, \dots, n\}$, and $g(M_1, \dots, M_n) = \sigma M'$. 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:

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

$$\mathit{ith}_n(\mathit{ntuple}(M_1, \dots, M_n)) = M_i$$

—$\mathit{sencrypt}(M, N)$ 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 $\mathit{pencrypt}$: $\mathit{pk}(M)$ 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

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

—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:

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

—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 cannot be directly represented by a constructor or by a destructor. We explain how we can treat such primitives in Section 8.

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 constructor application, 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). As usual, we may omit P when it is 0.

—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 $\mathit{let}\ x = M\ \mathit{in}\ P$ executes P with x bound to the term M.

—The conditional $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ 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 $M(x).P$, $\mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$, and $\mathit{let}\ x = M\ \mathit{in}\ P$. 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 variables; it may have free names. We identify processes up to renaming of bound names and variables. We write $\{M_1/x_1, \dots, M_n/x_n\}$ for the substitution that replaces $x_1, \dots, x_n$ with $M_1, \dots, M_n$, respectively. When σ is such a substitution and D is some expression, we may write σD or $D\sigma$ for the result of applying σ to D; the distinction is one of emphasis at most. Except when stated otherwise, substitutions always map variables (not names) to expressions.

As mentioned in the introduction, our calculus resembles the applied pi calculus [Abadi and Fournet 2001]. 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. (Section 8 contains further material on equational theories.) 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.

2.2 An Example

As an example, we return to the exchange presented in the introduction, namely:

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

Next we show how to express this protocol in the process calculus. We return again to this example in later sections, and there we discuss its formal analysis.

Informal protocol descriptions, such as the one for this protocol, are often ambiguous [Abadi 2000], so several different process-calculus expressions may be reasonable counterparts to an informal description. We start with a relatively simple representation of the protocol, given in the following process P:

$$\begin{array}{rl}P \triangleq & (\nu sK_A)(\nu sK_B)\mathit{let}\ pK_A = pk(sK_A)\ \mathit{in}\\ & \quad \mathit{let}\ pK_B = pk(sK_B)\ \mathit{in}\ \overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle.(A \mid B)\end{array}$$

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

Here we write $let\ (x, y) = M\ in\ Q$ instead of $let\ z = M\ in\ let\ x = 1th_2(z)\ in\ let\ y = 2th_2(z)\ in\ Q$, 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. This code corresponds to a basic, one-shot version of the protocol, in which A talks only to B and in which honest hosts that play the roles of A and B use different keys.

It is easy to extend the code to represent more elaborate, general versions of the protocol. For instance, the following process P' represents a version in which A and B run an unbounded number of sessions, A can talk to any host (whose public key A receives in x_{pK_B}), and the hosts that play the roles of A and B may have the same key:

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

Here B'' is much like B' but uses the same key as A'. (A separate definition of B'' is needed because, in the applied pi calculus, the syntactically different names sK_A and sK_B never mean the same. Of course, the code duplication can easily be avoided by using a variable parameter for the keys.)

This and other variants can be written rather directly as scripts in the input syntax of the automatic protocol checker, which is quite close to that of the process calculus. The following script illustrates this point:

```
(* First some declarations, with equations *)

(* Shared-key encryption *)
```

```
fun sencrypt/2.
reduc sdecrypt(sencrypt(x, y), y) = x.

(* Public-key encryption *)

fun pencrypt/2.
fun pk/1.
reduc pdecrypt(pencrypt(x, pk(y)), y) = x.

(* Declarations of free names *)

private free sA, sB.
free e.

(* A secrecy query, for protocol analysis *)

query attacker : sA;
      attacker : sB.

(* The processes *)

let processA' =
        in(e, xpkB);
        new k;
        out(e, pencrypt((k, pkA), xpkB));
        in(e, z);
        let (x, y) = pdecrypt(z, skA) in
        if x = k then
        (
          if xpkB = pkA then
          out(e, sencrypt(sA, y))
        )
        |
        (
          if xpkB = pkB then
          out(e, sencrypt(sB, y))
        ).

let processB' =
        in(e, z);
        let (x, y) = pdecrypt(z, skB) in
        new Kab;
        out(e, pencrypt((x, Kab), y));
        in(e, z2);
        let s2 = sdecrypt(z2, Kab) in
        0.

let processB'' =
        in(e, $z$);
        let $(x, y)$ = pdecrypt($z$, $skA$) in
        new $Kab$;
        out(e, pencrypt(($x$, $Kab$), $y$));
        in(e, $z2$);
        let $s2$ = sdecrypt($z2$, $Kab$) in
        0.

process new $skA$;
        new $skB$;
        let $pkA$ = pk($skA$) in
        let $pkB$ = pk($skB$) in
        out(e, $pkA$);
        out(e, $pkB$);
        ((!$processA'$) | (!$processB'$) | (!$processB''$))
```

As can be seen from this example, writing a model of a protocol in the process calculus is much like programming it in a little language with concurrency, message passing on named channels, and high-level, "black-box" operations on data (including cryptographic functions). In this respect, the calculus resembles many of the other programming calculi for protocols mentioned in the introduction.

The literature contains additional examples that provide evidence of the effectiveness of this process calculus and related ones for the analysis of a range of protocols. In particular, we have recently used this process calculus in the study of a protocol for certified email [Abadi et al. 2002; Abadi and Blanchet 2003a] and of the JFK protocol (a proposed replacement for IKE in IPsec) [Aiello et al. 2002; Abadi et al. 2004].

2.3 Formal Semantics

The rules of Figure 2 axiomatize the reduction relation $\rightarrow$ for processes, thus defining the operational semantics of our calculus. As is often done in process calculi (e.g., [Milner 1999]), 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 [Abadi and Blanchet 2003b], 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.

3. A DEFINITION OF SECRECY

As indicated in the introduction, 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. The interaction between P and Q takes place by communication on shared channels. These primarily include

$$\frac{}{P \mid 0 \equiv P} \qquad \frac{}{P \mid Q \equiv Q \mid P} \qquad \frac{}{(P \mid Q) \mid R \equiv P \mid (Q \mid R)}$$

$$\frac{}{!P \equiv P \mid !P}$$

$$\frac{}{(\nu a_1)(\nu a_2)P \equiv (\nu a_2)(\nu a_1)P} \qquad \frac{a \notin fn(P)}{(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q}$$

$$\frac{P \equiv Q}{P \mid R \equiv Q \mid R} \qquad \frac{P \equiv Q}{!P \equiv !Q} \qquad \frac{P \equiv Q}{(\nu a)P \equiv (\nu a)Q}$$

$$\frac{}{P \equiv P} \qquad \frac{Q \equiv P}{P \equiv Q} \qquad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R}$$

$$\frac{}{\overline{a}\langle M\rangle.Q \mid a(x).P \;\rightarrow\; Q \mid P\{M/x\}} \qquad \text{(Red I/O)}$$

$$\frac{g(M_1, \dots, M_n) = M'}{\mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \rightarrow P\{M'/x\}} \qquad \text{(Red Destr 1)}$$

$$\frac{g(M_1, \dots, M_n) \text{ is not defined}}{\mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \rightarrow Q} \qquad \text{(Red Destr 2)}$$

$$\frac{}{\mathit{let}\ x = M\ \mathit{in}\ P \rightarrow P\{M/x\}} \qquad \text{(Red Let)}$$

$$\frac{}{\mathit{if}\ M = M\ \mathit{then}\ P\ \mathit{else}\ Q \;\rightarrow\; P} \qquad \text{(Red Cond 1)}$$

$$\frac{M \neq N}{\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q \;\rightarrow\; Q} \qquad \text{(Red Cond 2)}$$

$$\frac{P \;\rightarrow\; Q}{P \mid R \;\rightarrow\; Q \mid R} \qquad \text{(Red Par)}$$

$$\frac{P \;\rightarrow\; Q}{(\nu a)P \;\rightarrow\; (\nu a)Q} \qquad \text{(Red Res)}$$

$$\frac{P' \equiv P,\ P \;\rightarrow\; Q,\ Q \equiv Q'}{P' \;\rightarrow\; Q'} \qquad \text{(Red} \equiv)$$

Fig. 2. Structural congruence and reduction

public channels (such as those of the Internet), on which Q may eavesdrop, modify, and inject messages; they may also include other channels known to Q. In addition to these shared channels, P may use private channels for its internal computations.

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 2.3.

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; in particular, these names may represent the cryptographic keys, communication channels, and nonces that Q knows initially. In the course of computation, Q may acquire some additional capabilities (for instance, additional cryptographic keys) not represented in S, 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 3.1. Let S be a finite set of names. The closed process Q is a S-adversary if and only if $fn(Q) \subseteq S$. 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 $c \in S$.

For instance, the process $(\nu k)\overline{a}\langle \mathit{sencrypt}(s,k)\rangle$ preserves the secrecy of s from $\{a\}$. This process publishes an encryption of s on the channel a, but not the decryption key; hence s does not escape. Similarly, the process $(\nu a)\overline{a}\langle \mathit{sencrypt}(s,k)\rangle$ preserves the secrecy of s from $\{k\}$; here the key is published but the channel remains private. On the other hand, the process $\overline{a}\langle \mathit{sencrypt}(s,k)\rangle$ does not preserve the secrecy of s from $\{a,k\}$: the adversary

$$a(x).\overline{a}\langle \mathit{sdecrypt}(x,k)\rangle$$

can receive $\mathit{sencrypt}(s,k)$ on a, decrypt s, and resend it on a.

As an additional example, we may apply this definition of secrecy to the process P of the example of Section 2.2. We may ask whether P preserves the secrecy of s from $\{e\}$. This property would mean that an attacker with access to e cannot learn s. Section 6.1 shows that this property indeed holds.

Definitions along these lines are quite common in protocol analysis. They are particularly popular and useful for treating the secrecy of keys and other atomic data items. There are however alternatives, in particular some definitions based on the concept of noninterference. According to those, a protocol parameter (such as the identity of a participant) is secret if an adversary cannot tell an instance of the protocol with one value of the parameter from an instance with a different value. The adversary may actually have these values, but ignore which is in use. In contrast, Definition 3.1 implies that, when a process P keeps the secrecy of a term M, the adversary does not have M. See [Abadi 2000] for further details and discussion.

4. THE TYPE SYSTEM

This section presents a general type system for our process calculus: Section 4.1 describes parameters and assumptions of the type system, and Section 4.2 describes its judgments and the type rules, which Figure 3 gives. The following sections include instances of this general type system.

4.1 Parameters and Requirements

The type system is parameterized by a set of types *Types* and a non-empty subset $T_{\text{Public}} \subseteq \mathit{Types}$. These parameters will be fixed in each instance of the type system. Always, T_{Public} is intended as the set of types of data that can be known by the attacker. The set T_{Public} is crucial in formulating our secrecy results, in which we assume that the attacker has names with types in T_{Public} and prove that it does not have names with types not in T_{Public}.

The type system relies on a function *conveys* : *Types* $\rightarrow \mathcal{P}(\mathit{Types})$ that satisfies property (P0):

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

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. Basically, these operators O_f and O_g give the types of constructor and destructor applications (much like type declarations for f and g), so they determine the type rules for constructors and destructors. As the type rules say, if $M_1, \dots, M_n$ have respective types $T_1, \dots, T_n$, f is a constructor of arity n, and $O_f(T_1, \dots, T_n)$ is defined, then $f(M_1, \dots, M_n)$ has type $O_f(T_1, \dots, T_n)$. Similarly, if $M_1, \dots, M_n$ have respective types $T_1, \dots, T_n$, g is a destructor of arity n, and $g(M_1, \dots, M_n)$ is defined, then $g(M_1, \dots, M_n)$ has a type in $O_g(T_1, \dots, T_n)$.

These constructor and destructor 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 $T_i \in T_{\text{Public}}$ for all $i \in \{1, \dots, n\}$, then $O_f(T_1, \dots, T_n)$ is defined and $O_f(T_1, \dots, T_n) \in T_{\text{Public}}$.

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

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

These properties are both reasonable and necessary for the soundness of the type system. 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. These properties are important in the proof of the Typability Lemma (Lemma 5.1.4 in Section 5). The third property essentially says that the definition of O_g on types is compatible with the definition of g on terms. This property is useful for type preservation when a destructor is applied, in the proof of the Subject Reduction Lemma (Lemma 5.1.3 in Section 5).

Thus, in summary, the type system is parameterized by:

—the set of types *Types*,

—the subset $T_{\text{Public}} \subseteq \mathit{Types}$,

—the function *conveys* : *Types* $\rightarrow \mathcal{P}(\mathit{Types})$,

—a partial function $O_f : \mathit{Types}^n \rightarrow \mathit{Types}$ for each constructor f of arity n, and

—a function $O_g : \mathit{Types}^n \rightarrow \mathcal{P}(\mathit{Types})$ for each destructor g of arity n,

and these parameters are subject to conditions (P0, P1, P2, P3).

As it stands, the type system is not parameterized by a subtyping relation. On the other hand, we sometimes find it convenient to use specific subtyping relations in instances of the type system, without however a "subsumption" rule [Cardelli 1997]. This rule is present in many programming languages with subtyping (but not all: see for example Objective Caml). It would enables us to view every element of a type T as having a type T' whenever T is a subtype of T', and therefore to apply any function f that expects an argument of type T' to any element of type T. It would be fairly easy to add this rule, should one wish to do so; we have developed some of the corresponding theory. We have not needed this rule because, as explained above, our constructors and destructors can accept arguments of different types, and return results whose types depend on the types of the arguments. Therefore, a function f that expects an argument of type T' can be defined to handle arguments of any other type T as well.

The soundness of the type system depends on the proper definition of constructors and destructors. In particular, the result of a constructor must be a new term, not equal to any other term. For instance, the identity function cannot be a constructor (but it may be a destructor). Otherwise, taking the identity function as a constructor, we could type it with $O_{id}(T) = T' \in T_{\text{Public}}$ for all T, so it could convert a secret type into a public one, and this would lead to wrong secrecy proofs. Once the constructors and destructors are defined correctly and the required properties (P0, P1, P2, P3) hold, the type system provides secrecy guarantees, as we show in the next section.

4.2 Judgments and Rules

Figure 3 gives the rules of the type system. In the rules, the metavariable u ranges over names and variables (that is, over atomic terms), 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 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. We could easily have a variant with explicit types on restrictions. The resulting type system would be less powerful, but our soundness results would still hold.

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

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:

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

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

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

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

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

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

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

(Destructor application)

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

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

Fig. 3. Type rules

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

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

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

—Let *equals* be a binary destructor with $\mathit{equals}(M, M) = M$ (and $\mathit{equals}(M, N)$ 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 (Conditional) for $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ is more natural than might seem at first sight.

In the rules (Input) and (Destructor application), the type system uses universal quantifications over a possibly infinite set of types, and the rule (Restriction) involves picking a type from a possibly infinite set. These features are important for the richness of the type system. For example, had we attached a single type to the variable in (Destructor application), we could not have dealt with situations in which a process receives an encrypted message with a cleartext of a statically unknown type.

On the other hand, these features are challenging from an algorithmic perspective: typechecking and type inference are not easy to implement in general. Unless explicit types are given, typechecking requires guessing the types of restricted names. Even worse, typechecking requires considering bound variables with a potentially infinite set of types, and verifying hypotheses for each of those types. That is why the instance presented in Section 6.1 is intended primarily for manual proofs.

Despite these difficulties, typechecking is certainly not out of reach, as we show. First, we demonstrate that the set of types is finite in significant cases, by providing an example in Section 6.2. Moreover, the logic-programming protocol checker of Section 7 yields a practical implementation in cases in which the sets are infinite.

Finally, having a very general (infinitary) type system strengthens our relative completeness result of Section 7.3. This result shows that the checker can prove all secrecy properties that can be proved with any instance of the type system, even infinitary instances.

5. PROPERTIES OF THE TYPE SYSTEM

Next we study the properties of the type system. We first establish a subject-reduction result and other basic lemmas, then use these results for proving a theorem about secrecy. Technically, we follow the same pattern as in the special case (protocols with asymmetric communication) treated in our previous work [Abadi and Blanchet 2003b], but some of the proofs require new arguments.

5.1 Subject Reduction and Typability

LEMMA 5.1.1 (SUBSTITUTION). *If* $E, E' \vdash M : T$ *and* $E, x : T, E' \vdash M' : T'$ *then* $E, E' \vdash M'\{M/x\} : T'$. *If* $E, E' \vdash M : T$ *and* $E, x : T, E' \vdash P$ *then* $E, E' \vdash P\{M/x\}$.

PROOF. The proof is by induction on the derivations of $E, x : T, E' \vdash M' : T'$ and of $E, x : T, E \vdash P$. The treatment of all rules is straightforward. □

LEMMA 5.1.2 (SUBJECT CONGRUENCE). *If $E \vdash P$ and $P \equiv Q$ then $E \vdash Q$.*

PROOF. This proof is similar to the corresponding proof for the type system of Cardelli, Ghelli, and Gordon [Cardelli et al. 2000]; it is an easy induction on the derivation of $P \equiv Q$. In the case of scope extrusion, we use a weakening lemma, which is easy to prove by induction on type derivations. □

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

LEMMA 5.1.3 (SUBJECT REDUCTION). *If $E \vdash P$ and $P \rightarrow Q$ then $E \vdash Q$.*

PROOF. The proof is by induction on the derivation of $P \rightarrow Q$.

—In the case of (Red I/O), we have

$$\overline{a}\langle M\rangle.Q \mid a(x).P \rightarrow Q \mid P\{M/x\}$$

We assume $E \vdash \overline{a}\langle M\rangle.Q \mid a(x).P$. This judgment must have been derived using the rule (Parallel composition), so $E \vdash \overline{a}\langle M\rangle.Q$ and $E \vdash a(x).P$. The judgment $E \vdash a(x).P$ must have been derived by (Input) from $E \vdash a : T$ and $\forall T' \in \mathit{conveys}(T), E, x : T' \vdash P$ for some T. The judgment $E \vdash \overline{a}\langle M\rangle.Q$ must have been derived by (Output) from $E \vdash a : T$ (for the same T as in the (Input) derivation, since each term has at most one type), $E \vdash M : T'$, $T' \in \mathit{conveys}(T)$, and $E \vdash Q$. By the substitution lemma (Lemma 5.1.1), we obtain $E \vdash P\{M/x\}$. By (Parallel composition), $E \vdash Q \mid P\{M/x\}$.

—In the case of (Red Destr 1), we have $g(M_1, \dots, M_n) = M'$ and

$$\mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \rightarrow P\{M'/x\}$$

We assume $E \vdash \mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$. This judgment must have been derived by (Destructor application) from $\forall i \in \{1, \dots, n\}, E \vdash M_i : T_i$, and $\forall T \in O_g(T_1, \dots, T_n), E, x : T \vdash P$ for some $T_1, \dots, T_n$. There exists an equation $g(N_1, \dots, N_n) = N'$ in $\mathrm{def}(g)$ and a substitution σ such that $\forall i \in \{1, \dots, n\}, M_i = \sigma N_i$ and $M' = \sigma N'$. For each variable x_j that occurs in $N_1, \dots, N_n$, we have a subterm σx_j of $M_1, \dots, M_n$, and a type T_{x_j} must have been given to this subterm when typing $M_1, \dots, M_n$, so we have $E \vdash \sigma x_j : T_{x_j}$ for each variable x_j that occurs in $N_1, \dots, N_n$. (All occurrences of each variable x_j have the same type, since each term has at most one type.) Since $E \vdash \sigma N_i : T_i$, we have $E' \vdash N_i : T_i$ where E' is the environment that associates each variable x_j with the type T_{x_j}. Since $g(N_1, \dots, N_n) = N'$ is in $\mathrm{def}(g)$, by (P3), there exists $T \in O_g(T_1, \dots, T_n)$, such that $E' \vdash N' : T$. By the substitution lemma (Lemma 5.1.1), $E \vdash M' : T$. Since $E, x : T \vdash P$, the substitution lemma yields $E \vdash P\{M'/x\}$.

—The cases (Red Let) and (Red Cond 1) follow by (Red Destr 1). (Recall that local definitions and conditionals can be encoded as destructor applications.)

The remaining cases are easy. □

In the study of programming languages, it is common to complement subject-reduction properties with progress properties. A typical progress property says that well-typed programs do not get stuck as a result of dynamic type errors—for example, attempting to multiply a boolean and an integer. Dynamic type errors are meaningful whenever the language's execution model includes dynamic type information, such as different tags on booleans and integers. Without such tags, on the other hand, the representations of booleans and integers may well be multiplied, though the result of such an operation will typically be implementation-dependent.

In our context, by analogy, one might consider stating a progress property that would guarantee that no "dynamic secrecy-type error" causes a process to get stuck. A "dynamic secrecy-type error" might be using public data as non-public data, or vice versa. Like ordinary dynamic type errors, "dynamic secrecy-type errors" are meaningful if the execution model includes dynamic type information, in this case tags that indicate secrecy types. The operational semantics of our process calculus does not however include such tags. Indeed, it is deliberately independent of any secrecy information. Furthermore, our typings do not imply any progress property: as the following typability lemma says, every process is well-typed (at least in a fairly trivial way that makes its free names and free variables public).

LEMMA 5.1.4 (TYPABILITY). *Let P be an untyped process. If $fn(P) \subseteq \{a_1, \dots, a_n\}$, $fv(P) \subseteq \{x_1, \dots, x_m\}$, $T'_i \in T_{Public}$ for all $i \in \{1, \dots, n\}$, and $T_i \in T_{Public}$ for all $i \in \{1, \dots, m\}$, then*

$$a_1 : T'_1, \dots, a_n : T'_n, x_1 : T_1, \dots, x_m : T_m \vdash P$$

PROOF. We first prove by induction that all terms are of a type in T_{Public}; that is:

$$a_1 : T'_1, \dots, a_n : T'_n, x_1 : T_1, \dots, x_m : T_m \vdash M : T$$

with $T \in T_{\text{Public}}$ if $fn(M) \subseteq \{a_1, \dots, a_n\}$, $fv(M) \subseteq \{x_1, \dots, x_m\}$, $T'_i \in T_{\text{Public}}$ for all $i \in \{1, \dots, n\}$, and $T_i \in T_{\text{Public}}$ for all $i \in \{1, \dots, m\}$.

—For names and variables, this follows by Rule (Atom).

—For composite terms $f(M_1, \dots, M_k)$, this follows by Rule (Constructor application) and induction hypothesis, since if $T''_i \in T_{\text{Public}}$ for all $i \in \{1, \dots, k\}$, then $O_f(T''_1, \dots, T''_k) \in T_{\text{Public}}$ by (P1).

Now we prove the claim, by induction on the structure of P.

—For output, notice that if $T \in T_{\text{Public}}$, then $T_{\text{Public}} \subseteq conveys(T)$ by (P0).

—For input, if $T \in T_{\text{Public}}$, then $T_{\text{Public}} \supseteq conveys(T)$ by (P0).

—In the case of restriction, we let the type of the new name be in T_{Public}.

—For destructor application, notice that if $T''_i \in T_{\text{Public}}$ for all $i \in \{1, \dots, k\}$, then $T \in T_{\text{Public}}$ for all $T \in O_g(T''_1, \dots, T''_k)$, by (P2).

□

This typability 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.

5.2 Secrecy

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

This secrecy theorem is a consequence of the subject-reduction lemma and the typability lemma.

Proof. Suppose that $S = \{a_1, \dots, a_l\}$, let T_i be the type of a_i, so $(a_i : T_i) \in E$ with $T_i \in T_{\text{Public}}$.

In order to derive a contradiction, we assume that P does not preserve the secrecy of s from S. Then there exists a process Q with $fv(Q) = \emptyset$ and $fn(Q) \subseteq S$, such that $P \mid Q \rightarrow^* R$ and $R \equiv \overline{c}\langle s\rangle.Q' \mid R'$, where $c \in S$. By Lemma 5.1.4, $a_1 : T_1, \dots, a_l : T_l \vdash Q$, so $E \vdash Q$. Therefore, $E \vdash P \mid Q$. By Lemma 5.1.3, $E \vdash R$, and by Lemma 5.1.2, $E \vdash \overline{c}\langle s\rangle.Q' \mid R'$. Since $c \in S$, we have $E \vdash c : T$ and $T \in T_{\text{Public}}$ for some T. The judgment $E \vdash \overline{c}\langle s\rangle.Q'$ must be derived by (Output) from $E \vdash c : T$ and $E \vdash s : T'$ with $T' \in conveys(T)$. Furthermore, $T' \in T_{\text{Public}}$ by (P0), contradicting the hypotheses of the theorem. So P preserves the secrecy of s from S. $\square$

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

Corollary 5.2.2. *Suppose that $a{:}T, s{:}T' \vdash P$ with $T \in T_{Public}$ and $T' \notin T_{Public}$. Then P preserves the secrecy of s from a. That is, for all closed processes Q such that $fn(Q) \subseteq \{a\}$, $P \mid Q$ does not output s on a.*

Suppose that the secrecy theorem implies that a process P preserves the secrecy of two names s and s', treating each of these names separately. The two applications of the secrecy theorem may in general rely on two different ways of showing that P is well-typed, with two different typing environments E and E'. We must have that $E \vdash s : T$ and $E' \vdash s' : T'$ for some types $T, T' \notin T_{\text{Public}}$. However, we may also have that $E \vdash s' : T_1$ and $E' \vdash s : T_1'$ for some types $T_1, T_1' \in T_{\text{Public}}$. Ideally, we may like to have a single environment E such that $E \vdash P$, $E \vdash s : T$, and $E \vdash s' : T'$ with $T, T' \notin T_{\text{Public}}$. Thus, E would make secret as much as possible, providing a "most secret typing" for P. In general, most secret typings are not always possible. For example, the instance of Section 6.1 does not guarantee the existence of most secret typings. (The proof is very similar to that in [Abadi and Blanchet 2003b, Section 5.3].) In contrast, in the instance of Section 7, the types generated by the verifier yield most secret typings.

6. SOME INSTANCES OF THE TYPE SYSTEM

As an important example, we show how the general type system applies to symmetric and asymmetric encryption. Specifically, we show two instances of the general type system, one infinitary and the other finitary, for processes that use symmetric and asymmetric encryption, built using the constructors *ntuple*, *sencrypt*, *pencrypt*, and *pk*, and the corresponding destructors ith_n, *sdecrypt*, and *pdecrypt*, introduced in Section 2. These instances are similar in scope and power to previous special-purpose type systems [Abadi and Blanchet 2003b; Abadi 1999], but they treat additional constructs and could easily treat even more.

In both instances, we include types for public and secret data, Public and Secret (as well as types with more structure, such as certain types for tuples). It would be straightforward to extend the instances with additional levels of secrecy, for example introducing an extra type TopSecret. Our results carry over to such extensions, and they can imply, for example, that even when data of type Secret is allowed to become public (by letting Secret $\in T_{\text{Public}}$), data of type TopSecret need not be. Such results are perhaps reminiscent of classic work on multilevel security (e.g., [Denning 1982]). However, unlike in that classic work, we need not require that the types form a lattice. We also have different concerns (for example, protecting against network attacks rather than against Trojan horses), and obtain different security guarantees (since our definition of secrecy does not preclude all information flows).

6.1 An Infinitary Instance

For the first 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.) Channel types are ordinary data types, so channel names can be encrypted and can be sent in messages.

—$\mathrm{K}^{\text{Secret}}[T]$ 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.)

—$\mathrm{EK}^{\text{Secret}}[T]$ is the type of secret asymmetric encryption keys that can be used to encrypt cleartexts of type T.

—$\mathrm{DK}^{\text{Secret}}[T]$ 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.

—$\mathrm{EK}^{\text{Public}}[T]$ 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.

—$\mathrm{DK}^{\text{Public}}[T]$ 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

Fig. 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 undefined, $O_g(T_1, \ldots, T_n) = \emptyset$.

Fig. 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

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

Importantly, the 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 of [Abadi and Blanchet 2003b].) As explained above, we do not need a "subsumption" rule.

Proposition 6.1.1. *These definitions satisfy the constraints of the general type system (P0, P1, P2, P3).*

Proof. (P0), (P1), and (P2) are obvious. We prove (P3).

—$ith_n(ntuple(M_1, \ldots, M_n)) = M_i$. Suppose that $E \vdash ntuple(M_1, \ldots, M_n) : T$. This judgment must have been derived by (Constructor application). Therefore, $T = T_1 \times \ldots \times T_n$ and $E \vdash M_i : T_i$, with $T_i \in O_{ith_n}(T)$.

—$sdecrypt(sencrypt(M, N), N) = M$. Suppose that $E \vdash sencrypt(M, N) : T_1$ and $E \vdash N : T_2$. The former judgment must have been derived by (Constructor application). Therefore, $E \vdash M : T$ and $O_{sencrypt}(T, T_2) = T_1 = \text{Public}$ for some T. By definition of $O_{sencrypt}$, we have two cases.
In case $T \leq \text{Public}$ and $T_2 \leq \text{Public}$, we obtain $E \vdash M : T$ and $T \in T_{\text{Public}} = O_{sdecrypt}(\text{Public}, T_2)$.
Otherwise, $T_2 = \mathrm{K}^{\text{Secret}}[T']$ with $T \leq T'$, so $E \vdash M : T$ and $T \in O_{sdecrypt}(\text{Public}, T_2)$.

—$pdecrypt(pencrypt(M, pk(N)), N) = M$. Suppose that $E \vdash pencrypt(M, pk(N)) : T_1$ and $E \vdash N : T_2$. The former judgment must have been derived by applying (Constructor application) twice, from $E \vdash M : T$ with $O_{pencrypt}(T, O_{pk}(T_2)) = T_1 = \text{Public}$ for some T. By definition of O_{pk}, we have three cases.
In case $T_2 \leq \text{Public}$, we have $O_{pk}(T_2) = \text{Public}$. Moreover, since $O_{pencrypt}(T, \text{Public}) = \text{Public}$, we also have $T \in T_{\text{Public}}$. Thus, $E \vdash M : T$ and $T \in T_{\text{Public}} = O_{pdecrypt}(\text{Public}, T_2)$.
In case $T_2 = \mathrm{DK}^{\text{Secret}}[T']$, we have $O_{pk}(T_2) = \mathrm{EK}^{\text{Secret}}[T']$. Moreover, since $O_{pencrypt}(T, \mathrm{EK}^{\text{Secret}}[T']) = \text{Public}$, we also have $T \leq T'$. Thus, $E \vdash M : T$ and $T \in O_{pdecrypt}(\text{Public}, T_2)$.

Otherwise, $T_2 = \mathrm{DK}^{\mathrm{Public}}[T']$, and we have $O_{pk}(T_2) = \mathrm{EK}^{\mathrm{Public}}[T']$. Moreover, since $O_{pencrypt}(T, \mathrm{EK}^{\mathrm{Public}}[T']) = \mathrm{Public}$, we also have $T \leq T'$ or $T \in \mathcal{T}_{\mathrm{Public}}$. We obtain $E \vdash M : T$ and $T \in O_{pdecrypt}(\mathrm{Public}, T_2)$.

□

As an immediate corollary, Theorem 5.2.1 applies, so we can prove secrecy by typing. For example, the type system can be used to establish that s remains secret in the process P of the example protocol of Section 2.2. For this proof, we define $E \triangleq s : \mathrm{Secret}, e : \mathrm{Public}$, and derive $E \vdash P$. In the (Restriction) rule, we choose the types

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

for sK_A and

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

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

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

and $pk(sK_B)$ has the type

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

The remainder of the process is typed in the environment:

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

We have that $T_{pK_A} \in \mathit{conveys}(\mathrm{Public})$ and $T_{pK_B} \in \mathit{conveys}(\mathrm{Public})$ (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

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

follows by (Constructor application), so the output $\overline{e}\langle \mathit{pencrypt}((k, pK_A), pK_B)\rangle$ 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 $E', k{:}\mathrm{Secret}, x{:}T_x, y{:}T_y \vdash$ *if* $x = k$ *then* $\overline{e}\langle \mathit{sencrypt}(s, y)\rangle$, where either $T_x \leq \mathrm{Secret}$ and $T_y \leq \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]$ or $T_x \leq \mathrm{Public}$ and $T_y \leq \mathrm{Public}$.

—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.

For typing B, by (Input), the type of z is a subtype of Public. By (Destructor application), we have to show that

$$E', x : T_x, y : T_y \vdash (\nu K_{AB}) \begin{pmatrix} \overline{e}\langle \mathit{pencrypt}((x, K_{AB}), y)\rangle. \\ e(z').\mathit{let}\ s' = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0 \end{pmatrix}$$

where either $T_x \leq$ Secret and $T_y \leq \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]$, or $T_x \leq$ Public and $T_y \leq$ Public.

—In the first case, we choose K_{AB} of type $\mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]$. We have

$$T_x \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}] \leq \mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]$$

and $T_y = \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]$ (the only subtype of $\mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]$ is itself), so $O_{pencrypt}(T_x \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}], T_y) = \mathrm{Public}$.

—In the second case, we choose K_{AB} of type Public. We have $T_x \times \mathrm{Public} \leq \mathrm{Public}$ and $T_y \leq \mathrm{Public}$, therefore $O_{pencrypt}(T_x \times \mathrm{Public}, T_y) = \mathrm{Public}$.

In both cases, it follows that the encryption is of type Public by (Constructor application), and that the output is well-typed. In both cases, also, the input $e(z').\mathit{let}\ s' = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0$ is clearly well-typed. Thus, we obtain $E \vdash P$. Finally, by Theorem 5.2.1, we conclude that P preserves the secrecy of s from $\{e\}$.

As for the process P' of Section 2.2, we cannot show that it preserves the secrecy of s_A or s_B from $\{e\}$ using this instance of the type system. This difficulty stems from two different uses of the key sK_A, which appear because A plays both roles in the protocol. (Indeed, if s_A or s_B were of type Secret, then y would be of type $\mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]$ in A', so sK_A would have a type of the form $\mathrm{DK}^{\mathrm{Public}}[T \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]$, and y could be of type $\mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]$ in B'' in conflict with the use of y as public encryption key.) In Section 6.2, we present a variant that avoids this difficulty. We postpone the formal analysis of the process P' itself to Section 7.

6.2 A Finitary Instance

Typechecking may be difficult, or at least non-trivial, in infinitary instances such as the one of Section 6.1, in which the type rules contain universal quantifications over infinite sets of types. In this section, we present a weaker instance that deals with the same function symbols but uses only a finite number of types. For this finitary instance, automatic typechecking and type inference are easy by exhaustive exploration of all typings.

The set of types of this instance is:

$$\mathit{Types} = \{\mathrm{Public}, \mathrm{Secret}, \mathrm{EK}^{\mathrm{Public}}, \text{Public-Secret-}\mathrm{EK}^{\mathrm{Public}}\}$$

These types have the following meanings:

—Public is the type of public data and Secret is the type of secret data, as in the previous instance.

—$\mathrm{EK}^{\mathrm{Public}}$ is the type of public asymmetric encryption keys such that the corresponding decryption keys are secret.

—Public-Secret-$\mathrm{EK}^{\mathrm{Public}}$ is the type of triples whose first component is of type Public, second component is of type Secret, and third component is of type $\mathrm{EK}^{\mathrm{Public}}$.

$T_{\text{Public}} = \{\text{EK}^{\text{Public}}, \text{Public}\}$.

If $T \in T_{\text{Public}}$, then $\mathit{conveys}(T) = T_{\text{Public}}$;
$\mathit{conveys}(\text{Secret}) = \{\text{Public-Secret-EK}^{\text{Public}}\}$.

If $T \in T_{\text{Public}}$, then $O_{3tuple}(T, \text{Secret}, \text{EK}^{\text{Public}}) = \text{Public-Secret-EK}^{\text{Public}}$;
$O_{ntuple}(\text{Secret}, \dots, \text{Secret}) = \text{Secret}$;
$O_{ntuple}(\text{EK}^{\text{Public}}, \dots, \text{EK}^{\text{Public}}) = \text{EK}^{\text{Public}}$;
if $T_1, \dots, T_n \in T_{\text{Public}}$ and there exists $i \in \{1, \dots, n\}$ such that $T_i \neq \text{EK}^{\text{Public}}$,
  then $O_{ntuple}(T_1, \dots, T_n) = \text{Public}$.
$O_{sencrypt}(\text{Public-Secret-EK}^{\text{Public}}, \text{Secret}) = \text{Public}$;
if $T_1, T_2 \in T_{\text{Public}}$, then $O_{sencrypt}(T_1, T_2) = \text{Public}$.
$O_{pencrypt}(\text{Public-Secret-EK}^{\text{Public}}, \text{EK}^{\text{Public}}) = \text{Public}$;
if $T_1, T_2 \in T_{\text{Public}}$, then $O_{pencrypt}(T_1, T_2) = \text{Public}$.
$O_{pk}(\text{Secret}) = \text{EK}^{\text{Public}}$;
if $T_1 \in T_{\text{Public}}$, then $O_{pk}(T_1) = \text{Public}$.

$O_{ith_n}(\text{EK}^{\text{Public}}) = \{\text{EK}^{\text{Public}}\}$;
$O_{ith_n}(\text{Public}) = T_{\text{Public}}$;
$O_{ith_n}(\text{Secret}) = \{\text{Secret}\}$;
$O_{1th_3}(\text{Public-Secret-EK}^{\text{Public}}) = T_{\text{Public}}$;
$O_{2th_3}(\text{Public-Secret-EK}^{\text{Public}}) = \{\text{Secret}\}$;
$O_{3th_3}(\text{Public-Secret-EK}^{\text{Public}}) = \{\text{EK}^{\text{Public}}\}$.
If $T \in T_{\text{Public}}$, then $O_{sdecrypt}(\text{Public}, T) = T_{\text{Public}}$;
$O_{sdecrypt}(\text{Public}, \text{Secret}) = \{\text{Public-Secret-EK}^{\text{Public}}\}$.
If $T \in T_{\text{Public}}$, then $O_{pdecrypt}(\text{Public}, T) = T_{\text{Public}}$;
$O_{pdecrypt}(\text{Public}, \text{Secret}) = \{\text{Public-Secret-EK}^{\text{Public}}, \text{EK}^{\text{Public}}, \text{Public}\}$.

Other cases: $\mathit{conveys}(T) = \emptyset$, $O_f(T_1, \dots, T_n)$ is undefined, $O_g(T_1, \dots, T_n) = \emptyset$.

Fig. 6. Definition of T_{Public} and type operators in another instance of the type system

We define T_{Public} and the type operators of the system in Figure 6. The resulting instance of the type system has similarities with the special-purpose type system of [Abadi 1999]. However, that type system does not handle public-key encryption; more importantly, it establishes a different notion of secrecy (a form of non-interference), and accordingly its typing of tests is more restrictive in order to prevent the so-called "implicit" information flows.

PROPOSITION 6.2.1. *These definitions satisfy the constraints of the general type system (P0, P1, P2, P3).*

PROOF. (P0), (P1), and (P2) are obvious. We prove (P3).

—$ith_n(ntuple(M_1, \dots, M_n)) = M_i$. Suppose that $E \vdash ntuple(M_1, \dots, M_n):T$. This judgment must have been derived by (Constructor application), so we have four cases, one for each case in the definition of O_{ntuple}.

(1) $n = 3$, $T =$ Public-Secret-EK$^{\text{Public}}$, $E \vdash M_1 : T'$ with $T' \in T_{\text{Public}}$, $E \vdash M_2 :$ Secret, $E \vdash M_3 : \text{EK}^{\text{Public}}$, and $T' \in T_{\text{Public}} = O_{1th_3}(T)$, Secret $\in O_{2th_3}(T)$, $\text{EK}^{\text{Public}} \in O_{3th_3}(T)$.

(2) $T =$ Secret, $E \vdash M_i :$ Secret, and Secret $\in O_{ith_n}$(Secret).

(3) $T = \text{EK}^{\text{Public}}$, $E \vdash M_i : \text{EK}^{\text{Public}}$, and $\text{EK}^{\text{Public}} \in O_{ith_n}(\text{EK}^{\text{Public}})$.

(4) $T =$ Public, $E \vdash M_i : T_i$ with $T_i \in T_{\text{Public}}$, and $T_i \in T_{\text{Public}} = O_{ith_n}$(Public).

—$sdecrypt(sencrypt(M, N), N) = M$. Suppose that $E \vdash sencrypt(M, N) : T_1$ and $E \vdash N : T_2$. The former judgment must have been derived by (Constructor application). Therefore, $E \vdash M : T$ and $O_{sencrypt}(T, T_2) = T_1 =$ Public for some T. By definition of $O_{sencrypt}$, we have two cases.
In case $T, T_2 \in T_{\text{Public}}$, we obtain $E \vdash M : T$ and $T \in T_{\text{Public}} = O_{sdecrypt}$(Public, T_2).
Otherwise, $T_2 =$ Secret and $T =$ Public-Secret-EK$^{\text{Public}}$, so $E \vdash M : T$ and $T \in O_{sdecrypt}(\text{Public}, T_2)$.

—$pdecrypt(pencrypt(M, pk(N)), N) = M$. Suppose that $E \vdash pencrypt(M, pk(N)) : T_1$ and $E \vdash N : T_2$. The former judgment must have been derived by applying (Constructor application) twice, from $E \vdash M : T$ with $O_{pencrypt}(T, O_{pk}(T_2)) = T_1 =$ Public for some T. By definition of O_{pk}, we have two cases.
In case $T_2 \in T_{\text{Public}}$, we have $O_{pk}(T_2) =$ Public. Moreover, since $O_{pencrypt}(T,$ Public$) =$ Public, we also have $T \in T_{\text{Public}}$. Thus, $E \vdash M : T$ and $T \in T_{\text{Public}} = O_{pdecrypt}(\text{Public}, T_2)$.
Otherwise, $T_2 =$ Secret, and we have $O_{pk}(T_2) = \text{EK}^{\text{Public}}$. Moreover, since $O_{pencrypt}(T, \text{EK}^{\text{Public}}) =$ Public, we also have $T =$ Public-Secret-EK$^{\text{Public}}$ or $T \in T_{\text{Public}}$. We obtain $E \vdash M : T$ and $T \in O_{pdecrypt}(\text{Public}, T_2)$.

□

The process P of Section 2.2 clearly does not typecheck in this type system, since the type system supports encryption of only public data and triples, and the protocol uses encryption of pairs containing secrets. This point illustrates that this instance is more restrictive than the instance of the previous section. We can however adapt the protocol to obtain a similar protocol that does typecheck. More precisely, we modify the encryptions so that their cleartexts are always of type Public-Secret-EK$^{\text{Public}}$. Thus the protocol becomes:

Message 1. $A \to B : \mathit{pencrypt}((a_{\text{Public}}, k, pK_A), pK_B)$
Message 2. $B \to A : \mathit{pencrypt}((a'_{\text{Public}}, (k, K_{AB}), a'_{\text{EK}^{\text{Public}}}), pK_A)$
Message 3. $A \to B : \mathit{sencrypt}((a''_{\text{Public}}, s, a''_{\text{EK}^{\text{Public}}}), K_{AB})$

where a_{Public} and similar fields indicate arbitrary padding of the appropriate types. This protocol can be represented by the following process:

$$\begin{aligned}
P &\triangleq (\nu sK_A)(\nu sK_B)\mathit{let}\ pK_A = pk(sK_A)\ \mathit{in}\\
&\qquad \mathit{let}\ pK_B = pk(sK_B)\ \mathit{in}\ \overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle.(A \mid B)\\
A &\triangleq (\nu k)(\nu a_{\text{Public}})\overline{e}\langle \mathit{pencrypt}((a_{\text{Public}}, k, pK_A), pK_B)\rangle.\\
&\qquad e(z).\mathit{let}\ (x'_1, (x, y), x'_3) = \mathit{pdecrypt}(z, sK_A)\ \mathit{in}\\
&\qquad \mathit{if}\ x = k\ \mathit{then}\ (\nu a''_{\text{Public}})(\nu a''_{\text{EK}^{\text{Public}}})\\
&\qquad \overline{e}\langle \mathit{sencrypt}((a''_{\text{Public}}, s, a''_{\text{EK}^{\text{Public}}}), y)\rangle\\
B &\triangleq e(z).\mathit{let}\ (x_1, x, y) = \mathit{pdecrypt}(z, sK_B)\ \mathit{in}\\
&\qquad (\nu K_{AB})(\nu a'_{\text{Public}})(\nu a'_{\text{EK}^{\text{Public}}})\\
&\qquad \overline{e}\langle \mathit{pencrypt}((a'_{\text{Public}}, (x, K_{AB}), a'_{\text{EK}^{\text{Public}}}), y)\rangle.\\
&\qquad e(z').\mathit{let}\ (x''_1, s', x''_3) = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0
\end{aligned}$$

This process is typable in this instance of the type system: letting $E \triangleq s : \text{Secret}, e : \text{Public}$, we can show that $E \vdash P$. In the (Restriction) rule, we choose the type Secret for sK_A and sK_B. Then $pk(sK_A)$ and $pk(sK_B)$ have the type $O_{pk}(\text{Secret}) = \text{EK}^{\text{Public}}$. The remainder of the process is typed in the environment:

$$E' \triangleq E, sK_A : \text{Secret}, sK_B : \text{Secret}, pK_A : \text{EK}^{\text{Public}}, pK_B : \text{EK}^{\text{Public}}$$

We check that $\text{EK}^{\text{Public}} \in \mathit{conveys}(\text{Public})$ (since this type is in T_{Public}), so the outputs $\overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle$ are well-typed. 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, a_{Public} of type Public. Then

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

follows by (Constructor application), so the output $\overline{e}\langle \mathit{pencrypt}((a_{\text{Public}}, k, pK_A), pK_B)\rangle$ is well-typed by (Output). In the input $e(z)$, by (Input), z can be of type Public or $\text{EK}^{\text{Public}}$, then by (Destructor application), $(x'_1, (x, y), x'_3)$ can be of type Public-Secret-$\text{EK}^{\text{Public}}$, Public, or $\text{EK}^{\text{Public}}$, so (x, y) can be of type Secret, Public, or $\text{EK}^{\text{Public}}$, hence we have to prove $E', k : \text{Secret}, x : T_x, y : T_y, a''_{\text{Public}} : \text{Public}, a''_{\text{EK}^{\text{Public}}} : \text{EK}^{\text{Public}} \vdash \mathit{if}\ x = k\ \mathit{then}\ \overline{e}\langle \mathit{sencrypt}((a''_{\text{Public}}, s, a''_{\text{EK}^{\text{Public}}}), y)\rangle$, where either $T_x = T_y = \text{Secret}$ or $T_x, T_y \in T_{\text{Public}}$.

—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.

For typing B, by (Input), the type of z is in T_{Public}. By (Destructor application), we have to show that

$$\begin{aligned} E', x : T_x, y : T_y \vdash{}& (\nu K_{AB})(\nu a'_{\text{Public}})(\nu a'_{\text{EK}^{\text{Public}}}) \\ & \quad \overline{e}\langle \mathit{pencrypt}((a'_{\text{Public}}, (x, K_{AB}), a'_{\text{EK}^{\text{Public}}}), y)\rangle. \\ & \quad e(z').\mathit{let}\ (x''_1, s', x''_3) = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0 \end{aligned}$$

where either $T_x = \text{Secret}$ and $T_y = \text{EK}^{\text{Public}}$, or $T_x, T_y \in T_{\text{Public}}$.

—In the first case, we choose K_{AB} of type Secret, a'_{Public} of type Public, and $a'_{\text{EK}^{\text{Public}}}$ of type $\text{EK}^{\text{Public}}$. Then $(a'_{\text{Public}}, (x, K_{AB}), a'_{\text{EK}^{\text{Public}}})$ is of type Public-Secret-$\text{EK}^{\text{Public}}$ and $O_{\mathit{pencrypt}}(\text{Public-Secret-EK}^{\text{Public}}, \text{EK}^{\text{Public}}) = \text{Public}$.

—In the second case, we choose K_{AB}, a'_{Public}, and $a'_{\text{EK}^{\text{Public}}}$ of type Public. Then $(a'_{\text{Public}}, (x, K_{AB}), a'_{\text{EK}^{\text{Public}}})$ is of type Public and $O_{\mathit{pencrypt}}(\text{Public}, T_y) = \text{Public}$.

In both cases, it follows that the encryption is of type Public by (Constructor application), and that the output is well-typed. The input $e(z').\mathit{let}\ (x''_1, s', x''_3) = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0$ is clearly well-typed in both cases. Thus, we obtain $E \vdash P$. Finally, by Theorem 5.2.1, we conclude that P preserves the secrecy of s from $\{e\}$.

We can adapt the process P' of Section 2.2 in a similar way, with the redefinitions:

$$\begin{aligned} P' &\triangleq (\nu sK_A)(\nu sK_B)\mathit{let}\ pK_A = \mathit{pk}(sK_A)\ \mathit{in} \\ &\qquad \mathit{let}\ pK_B = \mathit{pk}(sK_B)\ \mathit{in}\ \overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle.(!A' \mid !B' \mid !B'') \\ A' &\triangleq e(x_{pK_B}).(\nu k)(\nu a_{\text{Public}})\overline{e}\langle \mathit{pencrypt}((a_{\text{Public}}, k, pK_A), x_{pK_B})\rangle. \\ &\qquad e(z).\mathit{let}\ (x'_1, (x, y), x'_3) = \mathit{pdecrypt}(z, sK_A)\ \mathit{in} \\ &\qquad \mathit{if}\ x = k\ \mathit{then} \\ &\qquad (\mathit{if}\ x_{pK_B} = pK_B\ \mathit{then}\ (\nu a''_{\text{Public}})(\nu a''_{\text{EK}^{\text{Public}}}) \\ &\qquad\ \ \overline{e}\langle \mathit{sencrypt}((a''_{\text{Public}}, s_B, a''_{\text{EK}^{\text{Public}}}), y)\rangle \\ &\qquad \mid \mathit{if}\ x_{pK_B} = pK_A\ \mathit{then}\ (\nu a''_{\text{Public}})(\nu a''_{\text{EK}^{\text{Public}}}) \\ &\qquad\ \ \overline{e}\langle \mathit{sencrypt}((a''_{\text{Public}}, s_A, a''_{\text{EK}^{\text{Public}}}), y)\rangle) \\ B' &\triangleq e(z).\mathit{let}\ (x_1, x, y) = \mathit{pdecrypt}(z, sK_B)\ \mathit{in} \\ &\qquad (\nu K_{AB})(\nu a'_{\text{Public}})(\nu a'_{\text{EK}^{\text{Public}}}) \\ &\qquad \overline{e}\langle \mathit{pencrypt}((a'_{\text{Public}}, (x, K_{AB}), a'_{\text{EK}^{\text{Public}}}), y)\rangle. \\ &\qquad e(z').\mathit{let}\ (x''_1, s', x''_3) = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0 \\ B'' &\triangleq e(z).\mathit{let}\ (x_1, x, y) = \mathit{pdecrypt}(z, sK_A)\ \mathit{in} \\ &\qquad (\nu K_{AB})(\nu a'_{\text{Public}})(\nu a'_{\text{EK}^{\text{Public}}}) \\ &\qquad \overline{e}\langle \mathit{pencrypt}((a'_{\text{Public}}, (x, K_{AB}), a'_{\text{EK}^{\text{Public}}}), y)\rangle. \\ &\qquad e(z').\mathit{let}\ (x''_1, s', x''_3) = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0 \end{aligned}$$

We can show that this variant is well-typed in this instance of the type system: $e : \text{Public}, s_A : \text{Secret}, s_B : \text{Secret} \vdash P'$. Thus, we obtain that this process (but not the original process P' of Section 2.2) preserves the secrecy of s_A and s_B from $\{e\}$.

As in these examples, this finitary instance of the type system requires a rather strong discipline in the format of data encrypted under secret keys or sent on secret channels. While this discipline may not be hard to follow in writing new processes, it typically requires rewriting other processes before they can be typechecked. Even when the rewriting may appear simple, it may strengthen the processes in question. For example (as suggested above and explained fully in Section 7) the original process P' of Section 2.2 does not satisfy the secrecy properties that hold for its well-typed variant. What might be perceived as a disappointment if one is interested in the properties of the original process is a positive outcome if one aims to obtain security guarantees.

We return to the analysis of the original processes P and P' of Section 2.2 in Section 7.

7. THE 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 5.1.3).

As explained in the introduction, 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 [Blanchet 2001]. Interested readers may consult that work for further explanations of the material in the early part of Section 7.1.

In our definition and study of the checker, we emphasize its use for proving secrecy properties, in particular that names remain secret in the sense defined in Section 3. However, the checker has also been used for establishing other security properties. In particular, it has been quite effective in proofs of authenticity properties, expressed as correspondences between events [Blanchet 2002]. Recently, it has also been used in establishing certain process equivalences that capture strong secrecy properties [Blanchet 2004].

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.

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. Without loss of generality, 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. Thus, in the checker, a new name behaves as a function of the inputs that take place (lexically) before its creation. For instance, when we represent a process of the form $(\nu b)a(x).(\nu c)Q$, we use the pattern $a[]$ for the name a, $b[]$ for b, and $c[x]$ for c. Basically, we map a and b to constants, and c to a function of the input x.

We use the same patterns even when we treat processes with more replications, such as $!(\nu b)a(x).!(\nu c)Q$. Despite the replications, we use a single pattern for b, and one that depends only on x for c. Thus, 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:

$$\begin{array}{ll} \text{For each constructor } f \text{ of arity } n, & \\ \quad \text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n)) & \text{(Rf)} \\ \text{For each destructor } g, & \\ \quad \text{for each equation } g(M_1, \ldots, M_n) = M \text{ in def}(g), & \text{(Rg)} \\ \quad \text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M) & \\ \text{message}(x, y) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(y) & \text{(Rl)} \\ \text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{message}(x, y) & \text{(Rs)} \end{array}$$

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 def(g) is finite; handling this set is easiest in this finite case. In (Rg), notice that the terms $M_1, \ldots, M_n, M$ do not contain destructors, that equations in 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 the environment ρ is a function that associates a pattern with each name and variable, and h is a sequence of facts of the form 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 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.
—The sequence h 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 the translation avoids having to determine whether the destructor 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 the closed process P_0 as:

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

As an example, Figure 7 gives the rule base for the process P of the end of Section 2.1. In this rule base, all occurrences of message$(c[], M)$ where $c \in S$ are replaced by attacker(M). These two facts are equivalent by the rules (Rl) and (Rs). The rules for tuples are omitted; these rules are built-in in the protocol checker [Blanchet 2001].

We have the following secrecy result. Let $s \in \mathit{fn}(P_0)$. If attacker$(s[])$ cannot be derived from $B_{P_0,S}$, then P_0 preserves the secrecy of s from S. This result is the

$$
\begin{array}{l}
\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{attacker}(\mathit{pencrypt}(x, y))\\
\text{attacker}(x) \Rightarrow \text{attacker}(\mathit{pk}(x))\\
\text{attacker}(\mathit{pencrypt}(m, \mathit{pk}(k))) \wedge \text{attacker}(k) \Rightarrow \text{attacker}(m)\\
\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{attacker}(\mathit{sencrypt}(x, y))\\
\text{attacker}(\mathit{sencrypt}(m, k)) \wedge \text{attacker}(k) \Rightarrow \text{attacker}(m)\\
\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{message}(x, y)\\
\text{message}(x, y) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(y)\\
\text{attacker}(e[])\\
\text{attacker}(\mathit{pk}(sK_A[]))\\
\text{attacker}(\mathit{pk}(sK_B[]))\\
\text{attacker}(\mathit{pencrypt}((k[], \mathit{pk}(sK_A[])), \mathit{pk}(sK_B[])))\\
\text{attacker}(\mathit{pencrypt}((k[], x), \mathit{pk}(sK_A[]))) \Rightarrow \text{attacker}(\mathit{sencrypt}(s[], x))\\
\text{attacker}(\mathit{pencrypt}((x, y), \mathit{pk}(sK_B[])))\\
\quad \Rightarrow \text{attacker}(\mathit{pencrypt}((x, K_{AB}[\mathit{pencrypt}((x, y), \mathit{pk}(sK_B[]))]), y))
\end{array}
$$

Fig. 7. Rules for the process P of Section 2.2

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 resolution with free selection, as in [Blanchet 2001]. It has been shown that this algorithm always terminates for a class of protocols called tagged protocols [Blanchet and Podelski 2003]. Intuitively, a tagged protocol is a protocol in which each application of a cryptographic constructor (in particular, each encryption and each signature) is distinguished from others by a constant tag. For instance, to encrypt m under k, we write $\mathit{sencrypt}((c, m), k)$ instead of $\mathit{sencrypt}(m, k)$, where c is a constant tag. Different encryptions in the protocol use different tags, and the receiver of a message always checks the tags. Experimentally, the algorithm also terminates on many non-tagged protocols. Comon and Cortier show that an algorithm using ordered binary resolution, ordered factorization, and splitting terminates on protocols which blindly copy at most one term in each message [Comon-Lundh and Cortier 2003]. (A blind copy happens when a participant sends back part of a message it received without looking at what is contained inside this part.)

The secrecy result discussed above can be proved directly. Instead, below 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 5.2.1. 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
$\quad a[T_1, \ldots, T_n]$	name
$\quad f(T_1, \ldots, T_n)$	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 def(g), σ maps variables to types, and for all $i \in \{1, \ldots, n\}, \sigma M_i = T_i\}$.
(Notice that this definition is compatible with the definition of O_{id} and O_{equals} in the encoding of let and conditionals of Section 4.)

We have the following two results:

Proposition 7.2.1. *The checker's type system satisfies the constraints (P0, P1, P2, P3) of the general type system.*

Lemma 7.2.2. *Let P_0 be a closed process and $E = \{a : a[] \mid a \in \mathit{fn}(P_0)\}$. Then $E \vdash P_0$.*

The proofs of these results are in an appendix.

The secrecy theorem for the protocol checker follows from these results and the secrecy theorem for the general type system (Theorem 5.2.1):

Theorem 7.2.3 (Secrecy). *Let P_0 be a closed process and $s \in \mathit{fn}(P_0)$. If* attacker$(s[])$ *cannot be derived from $B_{P_0,S}$, then P_0 preserves the secrecy of s from S.*

Proof. Let $E = \{a : a[] \mid a \in \mathit{fn}(P_0)\}$, and $E' = \{a : a[] \mid a \in \mathit{fn}(P_0) \cup S\}$. By Lemma 7.2.2, $E \vdash P_0$, so $E' \vdash P_0$. Since attacker$(s[])$ cannot be derived from $B_{P_0,S}$, we have $s[] \notin T_{\text{Public}}$. Let $S' = \{b \mid E' \vdash b : T \text{ and } T \in T_{\text{Public}}\}$. By Theorem 5.2.1 (and Proposition 7.2.1), P_0 preserves the secrecy of s from S'. We have $S \subseteq S'$. (If $b \in S$, then attacker$(b[]) \in B_{P_0,S}$, so $b[] \in T_{\text{Public}}$ and $E' \vdash b : b[]$, so $b \in S'$.) Therefore, a fortiori, P_0 preserves the secrecy of s from S. $\square$

For example, attacker$(s[])$ is not derivable from $B_{P,\{e\}}$ where P is the process of Section 2.2, so we can show using this theorem that P preserves the secrecy of s from $\{e\}$. We can also show that the process P' preserves the secrecy s_B from $\{e\}$. However, attacker$(s_A[])$ is derivable from $B_{P',\{e\}}$, so we cannot prove that P' preserves the secrecy of s_A from $\{e\}$. More precisely, we can derive attacker$(s_A[])$ as follows. The clauses

$$\text{attacker}(\mathit{pk}(sK_A[])) \tag{1}$$

$$\text{attacker}(x_{pK_B}) \Rightarrow \text{attacker}(\mathit{pencrypt}((k[x_{pK_B}], \mathit{pk}(sK_A[])), x_{pK_B})) \tag{2}$$

$$\begin{aligned}&\text{attacker}(\mathit{pencrypt}((k[\mathit{pk}(sK_A[])], y), \mathit{pk}(sK_A[]))) \wedge \text{attacker}(\mathit{pk}(sK_A[])) \\ &\quad \Rightarrow \text{attacker}(\mathit{sencrypt}(s_A[], y))\end{aligned} \tag{3}$$

$$\text{attacker}(\mathit{sencrypt}(x, y)) \wedge \text{attacker}(y) \Rightarrow \text{attacker}(x) \tag{4}$$

are in $B_{P',\{e\}}$: (1) comes from the output $\overline{e}\langle pK_A\rangle$, (2) comes from the output of message 1 by A $\overline{e}\langle \mathit{pencrypt}((k, pK_A), x_{pK_B})\rangle$, (3) comes from the output of message 3 by A $\overline{e}\langle \mathit{sencrypt}(s_A, y)\rangle$, and (4) means that the adversary can decrypt when it has the key; it is (Rg) for the destructor *sdecrypt*. By (1), attacker($pk(sK_A[])$) is true; by (2), we derive attacker($\mathit{pencrypt}((k[pk(sK_A[])], pk(sK_A[])), pk(sK_A[]))$); by (3), we derive attacker($\mathit{sencrypt}(s_A[], pk(sK_A[]))$); and by (4), we finally obtain attacker($s_A[]$). This derivation corresponds to an attack against the protocol:

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

First A sends message 1 to itself playing the role of B. (This corresponds to applying the clause (2).) The attacker C intercepts this message and sends it back to A as message 2. A then replies with message 3 $\mathit{sencrypt}(s, pK_A)$. (This corresponds to applying the clause (3).) The attacker can decrypt this reply. (This corresponds to applying the clause (4).) This attack depends on A mistaking its own public key for a session key. Such "type confusions" are not always possible in concrete implementations (for example, because public keys and session keys may have different lengths). When they are, they can be prevented by tagging data with type tags. The "type confusions" can also be prevented through discipline: for example, in Section 6.2, the constraint that encryptions must take plaintexts of type Public-Secret-EK$^{\text{Public}}$ prevents the attack on a variant of P'. In this and many similar cases, type systems can support the prudent design of protocols.

We may note that this protocol is also subject to another attack, which does not compromise the secrecy of s_A and s_B and which resembles Lowe's attack against the Needham-Schroeder public-key protocol [Lowe 1996]:

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

In this attack, A executes a run with the adversary C, and C uses this run to execute a run of the protocol with B as if it were A. C decrypts the first message received from A, encrypts it with B's public key, and sends it to B. B then replies with the second message, which C simply forwards to A. A replies with the last message, which C forwards to B to complete the run. A then believes that k, K_{AB}, and s are secrets shared with C, while B believes that they are secrets shared with A. C can obtain k (but not s and K_{AB}). We can exhibit this attack by adding one more message $B \rightarrow A : \mathit{sencrypt}(s', k)$. The fact attacker($s'[]$) is then derivable from the clauses that represent the protocol and the adversary, as expected since the resulting protocol does not preserve the secrecy of s'. This attack can be prevented by adding the public key pK_B of B in the second message.

With this addition and the addition of tags (discussed above), we obtain the following exchange:

Message 1. $A \rightarrow B : \mathit{pencrypt}((c_1, k, pK_A), pK_B)$
Message 2. $B \rightarrow A : \mathit{pencrypt}((c_2, k, K_{AB}, pK_B), pK_A)$
Message 3. $A \rightarrow B : \mathit{sencrypt}(s, K_{AB})$
Message 4. $B \rightarrow A : \mathit{sencrypt}(s', k)$

where c_1 and c_2 are tags for messages 1 and 2, respectively. We have studied a process that represents this exchange. Using the checker, we have proved that this process preserves the secrecy of s and s', as desired. (We omit details of this analysis for the sake of brevity.)

Despite what the previous examples might suggest, a derivation of the fact attacker($s[]$) does not always correspond to an actual attack that compromises the secrecy of the corresponding name s. For instance, the process $P_0 = (\nu c)(\overline{c}\langle s\rangle \mid c(x).\overline{d}\langle c\rangle)$ preserves the secrecy of s from $\{d\}$, but the checker cannot establish it because attacker($s[]$) is derivable from the clauses (Rl), message($c[], s[]$), and message($c[], x$) $\Rightarrow$ attacker($c[]$) that are in $B_{P_0,\{d\}}$. (Note that message($d[], c[]$) is equivalent to attacker($c[]$) since d is a public channel.) These clauses do not take into account that the output $\overline{c}\langle s\rangle$ must have been executed before the adversary gets the channel c. This incompleteness is not specific to the checker. In particular, our relative completeness result (below) implies that no instance of our general type system can prove that P_0 preserves the secrecy of s from $\{d\}$. Furthermore, in practice, the checker rarely signals false attacks when applied to processes that correspond to actual protocols.

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 7.3.1 (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 5.2.1) 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 completeness result shows the power of the protocol checker. This power is not only theoretical: it has been demonstrated in practice on several examples [Blanchet 2001], from simple protocols like variants of the Needham-Schroeder protocols [Needham and Schroeder 1978] to Skeme [Krawczyk 1996], a certified email protocol [Abadi et al. 2002; Abadi and Blanchet 2003a], and JFK [Aiello et al. 2002; Abadi et al. 2004].

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 [Gordon and Jeffrey 2001], which deals with authenticity properties).

The proof of the theorem requires establishing a correspondence between types T of an instance of the general type system and closed patterns T_{c} (which are the types of the checker according to Section 7.2): we define a partial function ϕ that maps T_{c} to T. Then we prove that all rules of $B_{P_0,S}$ are satisfied, in the following sense:

Definition 7.3.2. The closed fact attacker(T_{c}) is said to be satisfied if $\phi(T_{\mathrm{c}})$ is defined and $\phi(T_{\mathrm{c}}) \in T_{\mathrm{Public}}$. The closed fact message$(T_{\mathrm{c}}, T'_{\mathrm{c}})$ is satisfied if $\phi(T'_{\mathrm{c}}) \in \mathit{conveys}(\phi(T_{\mathrm{c}}))$. The sequence of closed facts $F_1 \wedge \ldots \wedge F_n$ is satisfied if for all $i \in \{1, \ldots, n\}, F_i$ is satisfied. The rule $F_1 \wedge \ldots \wedge F_n \Rightarrow F$ is satisfied if, for every closed substitution σ such that $\sigma(F_1 \wedge \ldots \wedge F_n)$ is satisfied, σF is also 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.

The rest of this section gives a more detailed explanation of the proof. We consider a closed process P_0, a name s, and a set of names S. We also consider an instance of the general type system, and assume that this instance proves (by Theorem 5.2.1) 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 $T \notin T_{\mathrm{Public}}$, and $S = \{a \mid E_0 \vdash a : T \text{ and } T \in T_{\mathrm{Public}}\}$. 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.

Now we consider the protocol checker. All values concerning this system have index c. The set of types is:

$T_{\mathrm{c}} ::=$	types
$a[T_{\mathrm{c}1}, \ldots, T_{\mathrm{c}n}]$	name
$f(T_{\mathrm{c}1}, \ldots, T_{\mathrm{c}n})$	constructor application

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 $a[T_{\mathrm{c}1}, \ldots, T_{\mathrm{c}n}]$, where $T_{\mathrm{c}1}, \ldots, T_{\mathrm{c}n}$ are the types of the variables bound by inputs above $(\nu a)Q$ in P_0's syntax tree.

A $(T_{\mathrm{c}1}, \ldots, T_{\mathrm{c}n})$-well-chosen environment for P is similar, except that the parameters $(T_{\mathrm{c}1}, \ldots, T_{\mathrm{c}n})$ indicate which types should be chosen for the variables bound by inputs. Note that a $(T_{\mathrm{c}1}, \ldots, T_{\mathrm{c}n})$-well-chosen environment for P does not always exist, for example when the number of parameters $(T_{\mathrm{c}1}, \ldots, T_{\mathrm{c}n})$ does not correspond to the number of variables bound by inputs above P in P_0.

Definition 7.3.3. Let $T_{\mathrm{c}1}, \ldots, T_{\mathrm{c}n}$ be closed patterns. A $(T_{\mathrm{c}1}, \ldots, T_{\mathrm{c}n})$-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, for all T_{cn+1}.
—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 exist an equation $g(M'_1, \dots, M'_n) = M'$ in $\mathrm{def}(g)$ 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.
(In writing $E_c(M_i)$, we view E_c as a function on atoms and extend it to terms as a substitution.)

A pair (ρ, h) is a well-chosen pair for P if $h = \mathrm{message}(c_1, p_1) \wedge \dots \wedge \mathrm{message}(c_n, p_n)$ and, for every closed substitution σ, $\sigma\rho$ is a $(\sigma p_1, \dots, \sigma p_n)$-well-chosen environment for P.

A $(T_{c1}, \dots, T_{cn})$-well-chosen environment for P depends not only on the process P, but on its occurrence in P_0. However, notice that if $P = (\nu a)P'$ and we fix the bound name a, the occurrence of the process P is unique, since different restrictions in P_0 must create different names. We will have that, if $[\![P]\!]\rho h$ is called during the evaluation of $[\![P_0]\!]\rho_0\emptyset$ for $\rho_0 = \{a \mapsto a[] \mid (a : T) \in E_0\}$, then (ρ, h) is a well-chosen pair for P.

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 7.3.4. 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. 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 or decreases 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 with the computation of the well-chosen environment are free names or names bound above a (therefore names smaller than a).

In an appendix, we establish the following three lemmas:

Lemma 7.3.5. *Let $a \in S$. The fact* attacker($a[]$) *is satisfied.*

Lemma 7.3.6. *The rules for the attacker are satisfied.*

Lemma 7.3.7. *Let P be an occurrence of a subprocess of P_0, and (ρ, h) be a well-chosen pair for P. If, for every closed substitution σ such that σh is satisfied, $\phi \circ \sigma\rho \vdash P$ has been proved to obtain $E_0 \vdash P_0$, then the rules in $[\![P]\!]\rho h$ are satisfied. In particular, the rules in $[\![P_0]\!]\rho_0\emptyset$ are satisfied, where $\rho_0 = \{a \mapsto a[] \mid (a:T) \in E_0\}$.*

Using these lemmas, we obtain the theorem as indicated above:

Proof of Theorem 7.3.1. All the rules in $B_{P_0,S}$ are satisfied, by Lemmas 7.3.5, 7.3.6, and 7.3.7. By induction on derivations, we easily see that all facts derived from $B_{P_0,S}$ are satisfied. Moreover, $E_0 \vdash s : T$, with $T \notin T_{\text{Public}}$. By definition of ϕ, $\phi(s[]) = T \notin T_{\text{Public}}$. Therefore, attacker($s[]$) is not satisfied, so attacker($s[]$) cannot be derived from $B_{P_0,S}$, that is, the checker claims that P_0 preserves the secrecy of s from S. $\square$

8. TREATMENT OF GENERAL EQUATIONAL THEORIES

As Section 2.1 indicates, the classification of functions into constructors and destructors has limitations; for example, XOR does not fit in either class, so it is hard to treat. A convenient way to overcome these limitations is to allow more general equational theories, as in the applied pi calculus [Abadi and Fournet 2001]. This section briefly describes one treatment of those equational theories.

In this treatment, we assume that terms are subject to an equational theory $\mathcal{T}$, defined by a set of equations $M = N$ in which the terms M and N do not contain free names. The equational theory is the smallest congruence relation that includes this set of equations and that is preserved by substitution of terms for variables. We write $\mathcal{T} \vdash M = N$ when M equals N in the equational theory.

We can extend the semantics of our calculus to handle equational theories. For this purpose, we can either require that the definitions of destructors be invariant under the equational theory, or allow destructors that can non-deterministically yield several values. In either case, we add the structural congruence $P\{M/x\} \equiv$

$P\{N/x\}$ when $\mathcal{T} \vdash M = N$, and make sure that an *else* branch of a destructor is selected only when no equation makes it possible to apply the destructor. Similarly, for a conditional, the *else* branch should be selected only when the corresponding terms are not equal modulo $\mathcal{T}$.

It is fairly straightforward to extend the generic type system to equational theories. It suffices to add the condition that if two terms are equal then they have the same types:

(P4) If $E \vdash M : T$ and $\mathcal{T} \vdash M = N$, then $E \vdash N : T$.

On the other hand, defining useful instances of the generic type system (in the style of Section 6.1) can sometimes be difficult. For instance, it is not clear what types should be used for XOR or for Diffie-Hellman key-agreement operations, though we have ideas on the latter.

In extending the protocol checker of Section 7, we can use essentially the same Horn clauses to represent a protocol, but these Horn clauses have to be considered modulo an equational theory, and that raises difficult issues. We have to perform unifications modulo an equational theory, or to use other techniques for reasoning on Horn clauses modulo equations, such as paramodulation [Bachmair and Ganzinger 1998]. (Correspondingly, in our proofs, the types that correspond to the checker would be quotients of closed patterns by an equational theory.)

For simplicity, the current implementation of the checker includes only a simple treatment of equations. To each constructor f is attached a finite set of equations $f(M_1, \dots, M_n) = M$ which is required to satisfy certain closure conditions. It is then easy to generate appropriate Horn clauses for representing a protocol. Obviously, this approach limits which equational theories can be handled. For instance, this approach permits the equation $f(x, g(y)) = f(y, g(x))$, which can be used to model Diffie-Hellman operations [Abadi and Fournet 2001], but unification modulo an equational theory could yield a more detailed model [Meadows and Narendran 2002; Goubault-Larrecq et al. 2004].

9. 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. It focuses on techniques based on types and on logic programs because of their effectiveness and their popularity, illustrated by the many references given in the introduction. Previous work (in particular [Durgin and Mitchell 1999]) 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.

APPENDIX: ADDITIONAL PROOFS

This appendix contains a few proofs omitted in the main body of the paper.

A.1 Proofs of Proposition 7.2.1 and Lemma 7.2.2

If a finite function E maps atoms to types, we write E also for the environment that binds each atom u in $dom(E)$ with $u : E(u)$. The bindings can be in any order. In addition, the function E is extended to all terms as a substitution.

Lemma A.1.1. *In the type system of Section 7.2, if E binds all names and variables in M to types (that is, closed patterns), then*

$$E \vdash M : E(M)$$

Proof. The proof is by induction on the term M.

—For an atom u, we have $E \vdash u : E(u)$ by (Atom), hence the result.

—For a composite term $f(M_1, \dots, M_n)$, we have $E \vdash M_i : E(M_i)$ by induction hypothesis. Therefore, by (Constructor application), we obtain $E \vdash f(M_1, \dots, M_n)$: $E(f(M_1, \dots, M_n))$ since

$$\begin{aligned} O_f(E(M_1), \dots, E(M_n)) &= f(E(M_1), \dots, E(M_n)) \\ &= E(f(M_1, \dots, M_n)) \end{aligned}$$

□

Proof of Proposition 7.2.1. The proof relies on the rules that represent the attacker in the checker.

(P0) The rule attacker$(x) \wedge$ attacker$(y) \Rightarrow$ message(x, y) is in $B_{P_0,S}$. If $T \in T_{\text{Public}}$ and $T' \in T_{\text{Public}}$, then attacker(T) and attacker(T') can be derived from $B_{P_0,S}$. So message(T, T') can also be derived from $B_{P_0,S}$ and $T' \in$ *conveys*(T). Therefore, $T \in T_{\text{Public}}$ implies $T_{\text{Public}} \subseteq$ *conveys*(T).
Conversely, the rule attacker$(x) \wedge$ message$(x, y) \Rightarrow$ attacker(y) is also in $B_{P_0,S}$. If $T \in T_{\text{Public}}$ and $T' \in$ *conveys*(T) then $T' \in T_{\text{Public}}$. Therefore, $T \in T_{\text{Public}}$ implies $T_{\text{Public}} \supseteq$ *conveys*(T).

(P1) The rule attacker$(x_1) \wedge \dots \wedge$ attacker$(x_n) \Rightarrow$ attacker$(f(x_1, \dots, x_n))$ is in $B_{P_0,S}$. Therefore, if $T_1 \in T_{\text{Public}}, \dots, T_n \in T_{\text{Public}}$, then $O_f(T_1, \dots, T_n) \in T_{\text{Public}}$.

(P2) Assume that $T \in O_g(T_1, \dots, T_n)$. Then there exists an equation $g(M_1, \dots, M_n) = M$ in def(g) and a substitution σ such that $T_i = \sigma M_i$ for all i and $T = \sigma M$. The rule attacker$(M_1) \wedge \dots \wedge$ attacker$(M_n) \Rightarrow$ attacker(M) is in $B_{P_0,S}$. If attacker$(T_1) \wedge \dots \wedge$ attacker(T_n) can be derived from $B_{P_0,S}$, then attacker(T) can also be derived from $B_{P_0,S}$; therefore, if $T_i \in T_{\text{Public}}$ for all $i \in \{1, \dots, n\}$, then $T \in T_{\text{Public}}$.

(P3) If $g(M_1, \dots, M_n) = M$ is in def(g), and $E \vdash M_i : T_i$ for all i, then $T_i = E(M_i)$ by Lemma A.1.1 and the uniqueness of the type of a term. So, taking $T = E(M)$, we have $T \in O_g(T_1, \dots, T_n)$, by definition of O_g, and $E \vdash M : T$, by Lemma A.1.1.

□

Proof of Lemma 7.2.2. We prove by induction on the process P that, if

(1) ρ binds all free names and variables of P to patterns,
(2) $B_{P_0,S} \supseteq [\![P]\!]\rho h$,
(3) σ is a closed substitution, mapping all variables of h and of the image of ρ to patterns,
(4) for all p and p', if message$(p, p') \in h$ then $\sigma p' \in \mathit{conveys}(\sigma p)$,

then $\sigma\rho \vdash P$.

—Case 0: $\sigma\rho \vdash 0$ is always true (since $\sigma\rho$ is well-formed).

—Case $P \mid Q$: Assume that $[\![P \mid Q]\!]\rho h = [\![P]\!]\rho h \cup [\![Q]\!]\rho h \subseteq B_{P_0,S}$. Assume that σ satisfies (3) and (4). By induction hypothesis, $\sigma\rho \vdash P$ and $\sigma\rho \vdash Q$, so $\sigma\rho \vdash P \mid Q$ by (Parallel composition).

—Case $!P$: Assume that $[\![!P]\!]\rho h = [\![P]\!]\rho h \subseteq B_{P_0,S}$. Assume that σ satisfies (3) and (4). By induction hypothesis, $\sigma\rho \vdash P$, so $\sigma\rho \vdash !P$ by (Replication).

—Case $(\nu a)P$: Let $h = \text{message}(c_1, p_1) \wedge \dots \wedge \text{message}(c_n, p_n)$. Assume that

$$[\![(\nu a)P]\!]\rho h = [\![P]\!](\rho[a \mapsto a[p_1, \dots, p_n]])h \subseteq B_{P_0,S}$$

Assume that σ satisfies (3) and (4). By induction hypothesis, $\sigma\rho, a : \sigma(a[p_1, \dots, p_n]) \vdash P$. Therefore, $\sigma\rho \vdash (\nu a)P$ by (Restriction).

—Case $M(x).P$: Assume that

$$[\![M(x).P]\!]\rho h = [\![P]\!](\rho[x \mapsto x])(h \wedge \text{message}(\rho(M), x)) \subseteq B_{P_0,S}$$

Assume that σ satisfies (3) and (4). By Lemma A.1.1, $\sigma\rho \vdash M : \sigma\rho(M)$. Let $h' = h \wedge \text{message}(\rho(M), x)$. Let $T \in \mathit{conveys}(\sigma\rho(M))$. Let $\sigma' = \sigma[x \mapsto T]$. Then $\sigma' x \in \mathit{conveys}(\sigma'\rho(M))$, then message$(p, p') \in h'$ implies $\sigma' p' \in \mathit{conveys}(\sigma' p)$. By induction hypothesis, $\sigma'\rho, x : \sigma' x \vdash P$. So for all $T \in \mathit{conveys}(\sigma\rho(M))$, $\sigma\rho, x : T \vdash P$. By (Input), $\sigma\rho \vdash M(x).P$.

—Case $\overline{M}\langle N\rangle.P$: Assume that

$$[\![\overline{M}\langle N\rangle.P]\!]\rho h = [\![P]\!]\rho h \cup \{h \Rightarrow \text{message}(\rho(M), \rho(N))\} \subseteq B_{P_0,S}$$

Assume that σ satisfies (3) and (4). By induction hypothesis, $\sigma\rho \vdash P$. By Lemma A.1.1, $\sigma\rho \vdash M : \sigma\rho(M)$ and $\sigma\rho \vdash N : \sigma\rho(N)$. The rule $R = h \Rightarrow \text{message}(\rho(M), \rho(N))$ is in $B_{P_0,S}$. By condition (4), for each message(p, p') in h, $\sigma p' \in \mathit{conveys}(\sigma p)$, so message$(\sigma p, \sigma p')$ is derivable from $B_{P_0,S}$. Using the rule R, the fact message$(\sigma\rho(M), \sigma\rho(N))$ is also derivable from $B_{P_0,S}$. Therefore, we have $\sigma\rho(N) \in \mathit{conveys}(\sigma\rho(M))$. By (Output), $\sigma\rho \vdash \overline{M}\langle N\rangle.P$.

—Case *let* $x = g(M_1, \dots, M_n)$ *in* P *else* Q: Assume that

$$\begin{aligned}&[\![let\ x = g(M_1, \dots, M_n)\ in\ P\ else\ Q]\!]\rho h = \\ &\quad \cup \{[\![P]\!]((\sigma_1\rho)[x \mapsto \sigma'_1 p'])(\sigma_1 h) \\ &\qquad\quad \mid g(p'_1, \dots, p'_n) = p' \text{ is in } \mathrm{def}(g)\} \cup [\![Q]\!]\rho h \\ &\quad \subseteq B_{P_0,S}\end{aligned}$$

where (σ_1, σ'_1) is a most general pair of substitutions such that $\sigma_1\rho(M_1) = \sigma'_1 p'_1, \dots, \sigma_1\rho(M_n) = \sigma'_1 p'_n$. Assume that σ satisfies (3) and (4). By Lemma A.1.1, $\sigma\rho \vdash M_i : \sigma\rho(M_i)$ for all $i \in \{1, \dots, n\}$.
If $T \in O_g(\sigma\rho(M_1), \dots, \sigma\rho(M_n))$, then there exist an equation $g(p'_1, \dots, p'_n) = p'$ in $\mathrm{def}(g)$ and a substitution σ' such that, for all i, $\sigma\rho(M_i) = \sigma' p'_i$ and $T = \sigma' p'$. Then there exists σ'' such that $\sigma = \sigma''\sigma_1$ and $\sigma' = \sigma''\sigma'_1$. Moreover, we have

$$[\![P]\!](\sigma_1\rho[x \mapsto \sigma'_1 p'])(\sigma_1 h) \subseteq B_{P_0,S}$$

For all message$(p_1, p_2) \in \sigma''\sigma_1 h = \sigma h$, we have $p_2 \in \mathit{conveys}(p_1)$. By induction hypothesis on P, we have $\sigma''\sigma_1\rho, x : \sigma''\sigma'_1 p' \vdash P$, that is, $\sigma\rho, x : \sigma' p' \vdash P$.
Therefore, if $T \in O_g(\sigma\rho(M_1), \dots, \sigma\rho(M_n))$, then $\sigma\rho, x : T \vdash P$. Finally, by induction hypothesis on Q, $\sigma\rho \vdash Q$. By (Destructor application), $\sigma\rho \vdash$ *let* $x = g(M_1, \dots, M_n)$ *in* P *else* Q.

In particular, $B_{P_0,S} \supseteq [\![P_0]\!]\rho\emptyset$, where $\rho = \{a \mapsto a[] \mid a \in \mathit{fn}(P_0)\}$. Then, with $E = \sigma\rho = \{a : a[] \mid a \in \mathit{fn}(P_0)\}$, we obtain $E \vdash P_0$. □

A.2 Proof of Lemmas 7.3.5, 7.3.6, and 7.3.7

Proof of Lemma 7.3.5. Since $a \in S$, $(a : T) \in E_0$, with $T \in T_{\text{Public}}$. By definition of ϕ, $\phi(a[]) = T \in T_{\text{Public}}$. Therefore, attacker$(a[])$ is satisfied. □

Lemma A.2.1. *Let E_c be a partial function from atoms to closed patterns, defined for all names and variables of M. The function E_c is extended to a substitution.*

(1) If $\phi \circ E_c \vdash M : T$ then $T = \phi(E_c(M))$ (in particular, $\phi(E_c(M))$ is defined).

(2) If $\phi(E_c(M))$ is defined, then $\phi \circ E_c \vdash M : \phi(E_c(M))$.
(If $\phi(E_c(M))$ is defined, then ϕ is defined on $E_c(u)$ for all $u \in \mathit{fn}(M) \cup \mathit{fv}(M)$.)

Proof. The proof of (1) is by induction on M.

—Case M is an atom u. Since $\phi \circ E_c \vdash u : T$ must have been derived by (Atom), $T = \phi(E_c(u))$.

—Case M is a composite term $f(M_1, \dots, M_n)$. Since $\phi \circ E_c \vdash M : T$ can be obtained only by (Constructor), for each $i \in \{1, \dots, n\}$, $\phi \circ E_c \vdash M_i : T_i$ and $T = O_f(T_1, \dots, T_n)$. Therefore, by induction hypothesis, $T_i = \phi(E_c(M_i))$ and, by definition of ϕ, $T = O_f(\phi(E_c(M_1)), \dots, \phi(E_c(M_n))) = \phi(f(E_c(M_1), \dots, E_c(M_n))) = \phi(E_c(M))$.

The proof of (2) is also by induction on M.

—Case M is an atom u. By (Atom), $\phi \circ E_c \vdash u : \phi(E_c(u))$.

—Case M is a composite term $f(M_1, \dots, M_n)$. Since $\phi(E_c(M)) = \phi(f(E_c(M_1), \dots, E_c(M_n))) = O_f(\phi(E_c(M_1)), \dots, \phi(E_c(M_n)))$ is defined, $\phi(E_c(M_i))$ is defined for all $i \in \{1, \dots, n\}$. By induction hypothesis, we have $\phi \circ E_c \vdash M_i : \phi(E_c(M_i))$. Moreover, $O_f(\phi(E_c(M_1)), \dots, \phi(E_c(M_n)))$ is defined, therefore, by (Constructor), $\phi \circ E_c \vdash M : \phi(E_c(M))$.

□

Proof of Lemma 7.3.6. Let us prove first that $\text{attacker}(x) \wedge \text{message}(x, y) \Rightarrow \text{attacker}(y)$ is satisfied. Let σ be any closed substitution. If $\text{attacker}(\sigma x)$ and $\text{message}(\sigma x, \sigma y)$ are satisfied, then $\phi(\sigma x) \in T_{\text{Public}}$, so by (P0), $\mathit{conveys}(\phi(\sigma x)) = T_{\text{Public}}$ and $\phi(\sigma y) \in \mathit{conveys}(\phi(\sigma x)) = T_{\text{Public}}$. Then $\text{attacker}(\sigma y)$ is satisfied. Therefore, the rule $\text{attacker}(x) \wedge \text{message}(x, y) \Rightarrow \text{attacker}(y)$ is satisfied.

Similarly, $\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{message}(x, y)$ is satisfied.

Let f be a constructor. Let us prove that $\text{attacker}(x_1) \wedge \dots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \dots, x_n))$ is satisfied. Let σ be any closed substitution. Assume that $\text{attacker}(\sigma x_1), \dots, \text{attacker}(\sigma x_n)$ are satisfied. Then for all $i \in \{1, \dots, n\}$, $\phi(\sigma x_i) \in T_{\text{Public}}$, therefore $\phi(f(\sigma x_1, \dots, \sigma x_n)) = O_f(\phi(\sigma x_1), \dots, \phi(\sigma x_n)) \in T_{\text{Public}}$ by (P1). Then $\text{attacker}(f(\sigma x_1, \dots, \sigma x_n))$ is satisfied. Therefore, the rule $\text{attacker}(x_1) \wedge \dots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \dots, x_n))$ is satisfied.

Assume that there is an equation $g(M_1, \dots, M_n) = M$ in $\text{def}(g)$, and let us prove that $\text{attacker}(M_1) \wedge \dots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M)$ is satisfied. For every closed substitution σ, if $\text{attacker}(\sigma M_1), \dots, \text{attacker}(\sigma M_n)$ are satisfied, then for all $i \in \{1, \dots, n\}$, $\phi(\sigma M_i) \in T_{\text{Public}}$, so $O_g(\phi(\sigma M_1), \dots, \phi(\sigma M_n)) \subseteq T_{\text{Public}}$ by (P2). Moreover, for all $i \in \{1, \dots, n\}$, $\phi \circ \sigma \vdash M_i : \phi(\sigma M_i)$ by Lemma A.2.1(2), therefore $\phi \circ \sigma \vdash M : T$ and $T \in O_g(\phi(\sigma M_1), \dots, \phi(\sigma M_n))$ for some T by (P3). By Lemma A.2.1(1), $T = \phi(\sigma M)$, so $\phi(\sigma M) \in O_g(\phi(\sigma M_1), \dots, \phi(\sigma M_n))$. Hence $\phi(\sigma M) \in T_{\text{Public}}$, so $\text{attacker}(\sigma M)$ is satisfied. Therefore, the rule $\text{attacker}(M_1) \wedge \dots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M)$ is satisfied. □

Proof of Lemma 7.3.7. By induction on P.

—Case 0: $[\![0]\!]\rho h = \emptyset$, so the result is obvious.

—Case $P \mid Q$: Let (ρ, h) be a well-chosen pair for $P \mid Q$. Let σ be such that σh is satisfied, and $E = \phi \circ \sigma\rho$. If $E \vdash P \mid Q$ has been proved to obtain $E_0 \vdash P_0$, this must have been derived by (Parallel composition), therefore $E \vdash P$ and $E \vdash Q$ have been proved to obtain $E_0 \vdash P_0$. Since this is true for any σ such that σh is satisfied, and (ρ, h) is also a well-chosen pair for P and Q, by induction hypothesis, the rules in $[\![P]\!]\rho h$ and in $[\![Q]\!]\rho h$ are satisfied. Therefore, the rules in $[\![P \mid Q]\!]\rho h = [\![P]\!]\rho h \cup [\![Q]\!]\rho h$ are satisfied.

—Case $!P$: Let (ρ, h) be a well-chosen pair for $!P$. Let σ such that σh is satisfied, and $E = \phi \circ \sigma\rho$. If $E \vdash !P$ has been proved to obtain $E_0 \vdash P_0$, this must have been derived by (Replication), then $E \vdash P$ has been proved to obtain $E_0 \vdash P_0$. Since this is true for any σ such that σh is satisfied, and (ρ, h) is also a well-chosen pair for P, by induction hypothesis, the rules in $[\![P]\!]\rho h$ are satisfied. Therefore, the rules in $[\![!P]\!]\rho h = [\![P]\!]\rho h$ are satisfied.

—Case $(\nu a)P$: Let (ρ, h) be a well-chosen pair for $(\nu a)P$. Let σ be such that σh is satisfied, and $E = \phi \circ \sigma\rho$. If $E \vdash (\nu a)P$ has been proved to obtain $E_0 \vdash$

P_0, this must have been derived by (Restriction), then there exists T such that $E, a : T \vdash P$ has been proved to obtain $E_0 \vdash P_0$. By definition of ϕ, $T = \phi(a[\sigma p_1, \ldots, \sigma p_n])$, where $h = \text{message}(c_1, p_1) \wedge \ldots \wedge \text{message}(c_n, p_n)$, since $\sigma\rho$ is a $(\sigma p_1, \ldots, \sigma p_n)$-well-chosen environment for $(\nu a)P$. We have that $(\rho[a \mapsto a[p_1, \ldots, p_n]], h)$ is a well-chosen pair for P, and for any σ such that σh is satisfied, $\phi \circ \sigma\rho, a : \phi(a[\sigma p_1, \ldots, \sigma p_n]) \vdash P$ has been proved to obtain $E_0 \vdash P_0$. By induction hypothesis, the rules in $[\![(\nu a)P]\!]\rho h = [\![P]\!](\rho[a \mapsto a[p_1, \ldots, p_n]])h$ are satisfied.

—Case $M(x).P$: Let (ρ, h) be a well-chosen pair for $M(x).P$. We assume that for all σ such that σh is satisfied, and $E = \phi \circ \sigma\rho$, $E \vdash M(x).P$ has been proved to obtain $E_0 \vdash P_0$. Then this must have been derived by (Input), therefore $E \vdash M : T$ and $\forall T' \in \mathit{conveys}(T), E, x : T' \vdash P$. By Lemma A.2.1(1), $T = \phi(\sigma\rho(M))$. Let $h' = h \wedge \text{message}(\rho(M), x)$. If σ' is such that $\sigma' h'$ is satisfied, then $\text{message}(\sigma'\rho(M), \sigma' x)$ is satisfied, and $\phi(\sigma' x) \in \mathit{conveys}(\phi(\sigma'\rho(M))) = \mathit{conveys}(T)$. Moreover, $\sigma' h$ is satisfied, so we can apply the reasoning above to σ' instead of σ, therefore $E, x : \phi(\sigma' x) \vdash P$ for $E = \phi \circ \sigma'\rho$. Let $\rho' = \rho[x \mapsto x]$. Then (ρ', h') is a well-chosen pair for P, and $\phi \circ \sigma'\rho' \vdash P$ has been proved to obtain $E_0 \vdash P_0$. By induction hypothesis, the rules in $[\![M(x).P]\!]\rho h = [\![P]\!]\rho' h'$ are satisfied.

—Case $\overline{M}\langle N\rangle.P$: Let (ρ, h) be a well-chosen pair for $\overline{M}\langle N\rangle.P$. Let σ be such that σh is satisfied, and $E = \phi \circ \sigma\rho$. If $E \vdash \overline{M}\langle N\rangle.P$ has been proved to obtain $E_0 \vdash P_0$, then this must have been derived by (Output), therefore $E \vdash M : T$, $E \vdash N : T'$, $T' \in \mathit{conveys}(T)$, and $E \vdash P$. By Lemma A.2.1(1), $T = \phi(\sigma\rho(M))$ and $T' = \phi(\sigma\rho(N))$, therefore $\phi(\sigma\rho(N)) \in \mathit{conveys}(\phi(\sigma\rho(M)))$.
Let $R = h \Rightarrow \text{message}(\rho(M), \rho(N))$, and let σ' be any closed substitution. If $\sigma' h$ is satisfied, the argument of the paragraph above can be applied to σ'. Then $\phi(\sigma'\rho(N)) \in \mathit{conveys}(\phi(\sigma'\rho(M)))$, so $\text{message}(\sigma'\rho(M), \sigma'\rho(N))$ is satisfied. Therefore, R is satisfied.
We have that (ρ, h) is a well-chosen pair for P, and for all σ such that σh is satisfied, $E \vdash P$ has been proved to obtain $E_0 \vdash P_0$. By induction hypothesis on P, the rules in $[\![P]\!]\rho h$ are satisfied.
Hence the rules in $[\![\overline{M}\langle N\rangle.P]\!]\rho h = [\![P]\!]\rho h \cup \{R\}$ are satisfied.

—Case *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* Q: Let (ρ, h) be a well-chosen pair for *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* Q. We assume that for all σ such that σh is satisfied, and $E = \phi \circ \sigma\rho$, $E \vdash$ *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* Q has been proved to obtain $E_0 \vdash P_0$. This must have been derived by (Destructor application), then $\forall i \in \{1, \ldots, n\}$, $E \vdash M_i : T_i$, $\forall T \in O_g(T_1, \ldots, T_n), E, x : T \vdash P$, and $E \vdash Q$. By Lemma A.2.1(1), $T_i = \phi(\sigma\rho(M_i))$.
Assume that there is an equation $g(p'_1, \ldots, p'_n) = p'$ in $\text{def}(g)$. Let $\rho' = \sigma_1\rho[x \mapsto \sigma'_1 p']$ and $h' = \sigma_1 h$ where (σ_1, σ'_1) is the most general pair of substitutions such that $\sigma_1\rho(M_1) = \sigma'_1 p'_1, \ldots, \sigma_1\rho(M_n) = \sigma'_1 p'_n$. Let σ'' be such that $\sigma'' h'$ is satisfied. Then $\sigma = \sigma''\sigma_1$ is such that σh is satisfied, so the argument of the paragraph above can be applied to σ. Moreover $\sigma\rho(M_i) = \sigma''\sigma'_1 p'_i$. We have $\phi \circ \sigma''\sigma'_1 \vdash p'_i : \phi(\sigma''\sigma'_1 p'_i)$ (by Lemma A.2.1(2)). Therefore, by (P3), $\phi \circ \sigma''\sigma'_1 \vdash p' : \phi(\sigma''\sigma'_1 p')$ with $\phi(\sigma''\sigma'_1 p') \in O_g(\phi(\sigma''\sigma'_1 p'_1), \ldots, \phi(\sigma''\sigma'_1 p'_n))$. That is, $\phi(\sigma''\sigma'_1 p') \in O_g(T_1, \ldots, T_n)$. Therefore $E, x : \phi(\sigma''\sigma'_1 p') \vdash P$. That is, $\phi \circ \sigma''\rho' \vdash P$. This is true for any σ'' such that $\sigma'' h'$ is satisfied, and (ρ', h') is a well-chosen

pair for P, therefore by induction hypothesis, the rules in $[\![P]\!]\rho' h'$ are satisfied. Moreover, (ρ, h) is also a well-chosen pair for Q, then by induction hypothesis, the rules in $[\![Q]\!]\rho h$ are satisfied.
Therefore, the rules in $[\![\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho h$ are satisfied.

In particular, for $[\![P_0]\!]\rho_0\emptyset$, $(\rho_0, \emptyset)$ is a well-chosen pair for P_0, and $E_0 = \{a : \phi(a[\,]) \mid (a : T) \in E_0\} = \phi \circ \sigma\rho_0$, for any σ. Therefore, $\phi \circ \sigma\rho_0 \vdash P_0$ has been proved to obtain $E_0 \vdash P_0$. Then the rules in $[\![P_0]\!]\rho_0\emptyset$ are satisfied. $\square$

ACKNOWLEDGMENTS

This work was partly done while Martín Abadi was at Bell Labs Research, Lucent Technologies, and at InterTrust's Strategic Technologies and Architectural Research Laboratory, and while Bruno Blanchet was at INRIA Rocquencourt and at Max-Planck-Institut für Informatik. Martín Abadi's research was partly supported by faculty research funds granted by the University of California, Santa Cruz, and by the National Science Foundation under Grants CCR-0204162 and CCR-0208800. We would like to thank the anonymous referees, the Area Editor, and Xavier Allamigeon for their helpful comments on this paper.

REFERENCES

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

Abadi, M. 2000. Security protocols and their properties. In *Foundations of Secure Computation*, F. Bauer and R. Steinbrueggen, Eds. NATO Science Series. IOS Press, Amsterdam, The Netherlands, 39–60. Volume for the 20th International Summer School on Foundations of Secure Computation, held in Marktoberdorf, Germany (1999).

Abadi, M., Banerjee, A., Heintze, N., and Riecke, J. G. 1999. A core calculus of dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages.* ACM Press, New-York, NY, 147–160.

Abadi, M. and Blanchet, B. 2003a. Computer-assisted verification of a protocol for certified email. In *Static Analysis, 10th International Symposium (SAS'03)* (San Diego, California), R. Cousot, Ed. Lecture Notes in Computer Science, vol. 2694. Springer-Verlag, Berlin, Germany, 316–335.

Abadi, M. and Blanchet, B. 2003b. Secrecy types for asymmetric communication. *Theoretical Computer Science 298,* 3 (Apr.), 387–415.

Abadi, M., Blanchet, B., and Fournet, C. 2004. Just Fast Keying in the pi calculus. In *Programming Languages and Systems: 13th European Symposium on Programming (ESOP 2004)* (Barcelona, Spain), D. Schmidt, Ed. Lecture Notes in Computer Science, vol. 2986. Springer-Verlag, Berlin, Germany, 340–354.

Abadi, M. and Fournet, C. 2001. Mobile values, new names, and secure communication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Programming Languages (POPL'01).* ACM Press, New-York, NY, 104–115.

Abadi, M., Glew, N., Horne, B., and Pinkas, B. 2002. Certified email with a light on-line trusted third party: Design and implementation. In *11th International World Wide Web Conference* (Honolulu, Hawaii). ACM Press, New-York, NY, 387–395.

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

Aiello, W., Bellovin, S., Blaze, M., Canetti, R., Ionnidis, J., Keromytis, A., and Reingold, O. 2002. Efficient, DoS-resistant, secure key exchange for internet protocols. In *ACM Conference on Computer and Communications Security (CCS'02)*, R. Sandhu, Ed. ACM, New-York, NY, 48–58.

Amadio, R. M. and Lugiez, D. 2000. On the reachability problem in cryptographic protocols. In *CONCUR 2000: Concurrency Theory (11th International Conference)*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 1877. Springer-Verlag, Berlin, Germany, 380–394.

Bachmair, L. and Ganzinger, H. 1998. Equational reasoning in saturation-based theorem proving. In *Automated Deduction — A Basis for Applications*, W. Bibel and P. Schmitt, Eds. Vol. I. Kluwer, Dordrecht, The Netherlands, Chapter 11, 353–397.

Blanchet, B. 2001. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. IEEE Computer Society, Los Alamitos, CA, 82–96.

Blanchet, B. 2002. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)* (Madrid, Spain), M. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, Berlin, Germany, 342–359.

Blanchet, B. 2004. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy* (Oakland, California). IEEE Computer Society, Los Alamitos, CA, 86–100.

Blanchet, B. and Podelski, A. 2003. Verification of cryptographic protocols: Tagging enforces termination. In *Foundations of Software Science and Computation Structures (FoSSaCS'03)* (Warsaw, Poland), A. Gordon, Ed. Lecture Notes in Computer Science, vol. 2620. Springer-Verlag, Berlin, Germany, 136–152.

Bodei, C. 2000. Security issues in process calculi. Ph.D. thesis, Università di Pisa.

Bodei, C., Degano, P., Nielson, F., and Nielson, H. 1998. Control flow analysis for the π-calculus. In *CONCUR'98: Concurrency Theory.* Lecture Notes in Computer Science, vol. 1466. Springer Verlag, Berlin, Germany, 84–98.

Cardelli, L. 1997. Type systems. In *The Computer Science and Engineering Handbook*, A. B. Tucker, Ed. CRC Press, Boca Raton,FL, Chapter 103, 2208–2236.

Cardelli, L., Ghelli, G., and Gordon, A. D. 2000. Secrecy and group creation. In *CONCUR 2000: Concurrency Theory*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 1877. Springer-Verlag, Berlin, Germany, 365–379.

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

Comon-Lundh, H. and Cortier, V. 2003. New decidability results for fragments of first-order logic and application to cryptographic protocols. In *14th Int. Conf. Rewriting Techniques and Applications (RTA'2003)* (Valencia, Spain), R. Nieuwenhuis, Ed. Lecture Notes in Computer Science, vol. 2706. Springer-Verlag, Berlin, Germany, 148–164.

Compton, K. J. and Dexter, S. 1999. Proof techniques for cryptographic protocols. In *Automata, Languages and Programming, 26th International Colloquium, ICALP'99* (Prague, Czech Republic), J. Wiedermann, P. van Emde Boas, and M. Nielsen, Eds. Lecture Notes in Computer Science, vol. 1644. Springer-Verlag, Berlin, Germany, 25–39.

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

Debbabi, M., Mejri, M., Tawbi, N., and Yahmadi, I. 1997. 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.

Denker, G., Meseguer, J., and Talcott, C. 1998. Protocol specification and analysis in Maude. In *Proc. of Workshop on Formal Methods and Security Protocols* (Indianapolis, Indiana), N. Heintze and J. Wing, Eds.

Denning, D. E. 1982. *Cryptography and Data Security.* Addison-Wesley, Reading, Mass.

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

Durgin, N., Mitchell, J., and Pavlovic, D. 2001. A compositional logic for protocol correctness. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. IEEE Computer Society, Los Alamitos, CA, 241–255.

Durgin, N. A. and Mitchell, J. C. 1999. Analysis of security protocols. In *Calculational System Design*, M. Broy and R. Steinbruggen, Eds. IOS Press, Amsterdam, The Netherlands, 369–395.

Focardi, R. and Gorrieri, R. 1997. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering 23,* 9 (Sept.), 550–571.

Gordon, A. and Jeffrey, A. 2001. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. IEEE Computer Society, Los Alamitos, CA, 145–159.

Gordon, A. and Jeffrey, A. 2002. Types and effects for asymmetric cryptographic protocols. In *15th IEEE Computer Security Foundations Workshop (CSFW-15)*. IEEE Computer Society, Los Alamitos, CA, 77–91.

Goubault-Larrecq, J. 2002. Protocoles cryptographiques: la logique à la rescousse! In *Atelier SEcurité des Communications sur Internet (SECI'02)* (Tunis, Tunisie).

Goubault-Larrecq, J. 2004. Une fois qu'on n'a pas trouvé de preuve, comment le faire comprendre à un assistant de preuve ? In *Actes 15èmes journées francophones sur les langages applicatifs (JFLA'04)* (Sainte-Marie-de-Ré, France). INRIA, Rocquencourt, France.

Goubault-Larrecq, J., Roger, M., and Verma, K. N. 2004. Abstraction and resolution modulo AC: How to verify Diffie-Hellman-like protocols automatically. *Journal of Logic and Algebraic Programming*. To appear.

Heintze, N. and Riecke, J. G. 1998. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages.* ACM Press, New-York, NY, 365–377.

Hennessy, M. and Riely, J. 2000. 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. Springer-Verlag, Berlin, Germany, 415–427.

Honda, K., Vasconcelos, V., and Yoshida, N. 2000. Secure information flow as typed process behaviour. In *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)*, G. Smolka, Ed. Lecture Notes in Computer Science, vol. 1782. Springer-Verlag, Berlin, Germany, 180–199.

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

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

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

Lowe, G. 1996. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems.* Lecture Notes in Computer Science, vol. 1055. Springer Verlag, Berlin, Germany, 147–166.

Meadows, C. 1997. Panel on languages for formal specification of security protocols. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop.* IEEE Computer Society, Los Alamitos, CA, 96.

Meadows, C. and Narendran, P. 2002. A unification algorithm for the group Diffie-Hellman protocol. In *Workshop on Issues in the Theory of Security (WITS'02)* (Portland, Oregon).

Milner, R. 1999. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, Cambridge, United Kingdom.

Morris, J. H. 1973. Protection in programming languages. *Commun. ACM 16,* 1 (Jan.), 15–21.

Myers, A. C. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages.* ACM Press, New-York, NY, 228–241.

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

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

Selinger, P. 2001. Models for an adversary-centric protocol logic. In *Proceedings of the 1st Workshop on Logical Aspects of Cryptographic Protocol Verification* (Paris, France), J. Goubault-Larrecq, Ed. Electronic Notes in Theoretical Computer Science, vol. 55(1). Elsevier, Amsterdam, The Netherlands, 73–88.

Sumii, E. and Pierce, B. C. 2001. Logical relations and encryption (Extended abstract). In *14th IEEE Computer Security Foundations Workshop (CSFW-14).* IEEE Computer Society, Los Alamitos, CA, 256–269.

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

Weidenbach, C. 1999. Towards an automatic analysis of security protocols in first-order logic. In *16th International Conference on Automated Deduction (CADE-16)*, H. Ganzinger, Ed. Lecture Notes in Artificial Intelligence, vol. 1632. Springer-Verlag, Berlin, Germany, 314–328.

Received September 2002; revised April and September 2004; accepted August 2004