A Type Discipline for Authorization in Distributed Systems

Cédric Fournet
Microsoft Research

Andrew D. Gordon
Microsoft Research

Sergio Maffeis
Imperial College

Abstract

We consider the problem of statically verifying the conformance of the code of a system to an explicit authorization policy. In a distributed setting, some part of the system may be compromised, that is, some nodes of the system and their security credentials may be under the control of an attacker. To help predict and bound the impact of such partial compromise, we advocate logic-based policies that explicitly record dependencies between principals. We propose a conformance criterion, safety despite compromised principals*, such that an invalid authorization decision at an uncompromised node can arise only when nodes on which the decision logically depends are compromised. We formalize this criterion in the setting of a process calculus, and present a verification technique based on a type system. Hence, we can verify policy conformance of code that uses a wide range of the security mechanisms found in distributed systems, ranging from secure channels down to cryptographic primitives, including encryption and public-key signatures.*

1 Introduction

We are given implementation code for a distributed system, with security relevant events suitably annotated. We are given a target authorization policy, expressed in a suitable logic. How can we check statically whether the code correctly implements the policy?

Our approach, begun in a previous paper [23], is to represent code in a process calculus, and to develop a type system such that successful typechecking implies that code conforms to the policy. We treat policies for authorization (access control) and their formulation as a parameter—developing appropriate languages and tools for policies is a separate issue [6, 16, 42, 22, 7, 34, 33, 11, 19, 45, 35, 8, 20, 9, 10, 32]. This paper adds a range of features needed for modelling systems where trust is decentralized. These features include: a more comprehensive range of cryptographic primitives; a connection between the logical treatments of principals (people, computers, services), using the modality a **says** C, and their representations as formal processes; and a study of the meaning of policy conformance in the presence of principal compromise. An aim is to discover a unified logical treatment of both authorization and authentication mechanisms built on top of secure channels and cryptography.

An Example To illustrate our approach, consider a simple distributed system consisting of code running on behalf of three principals: an online music store, a user wishing to download music, and a proxy device with which the user is registered.

We assume a logical policy governing authorization decisions (expressed in a logic defined in Section 2): a song can be downloaded to a user if the user has ordered the song and the user is registered at the proxy:

$$\forall u, s.((\mathit{proxy}\ \mathbf{says}\ \mathit{Registered}(u) \wedge u\ \mathbf{says}\ \mathit{Order}(s)) \rightarrow \mathit{CanDownload}(u,s)) \quad (1)$$

We assume that each piece of code reacting to a security-related event is annotated with a logical formula representing the event. For example, the point at which the code acting for *user* receives an instruction to download *song* is annotated with:

$$\mathit{user}\ \mathbf{says}\ \mathit{Order}(\mathit{song}) \quad (2)$$

Similarly, the point in the code for *proxy* where it registers *user* is marked with:

$$\mathit{proxy}\ \mathbf{says}\ \mathit{Registered}(\mathit{user}) \quad (3)$$

We also assume that each piece of code guarding a sensitive action is annotated with an *expectation*, a formula that should hold in any run. For example, we annotate the point at which the authorization decision has been made, and the store is about to start the transfer of a *song* to *user*, with the expectation:

$$\mathit{store}\ \mathbf{says}\ \mathit{CanDownload}(\mathit{user}, \mathit{song}) \quad (4)$$

Let an implementation be *robustly safe* if and only if all its expectations hold in every run of the system, even in the presence of an active adversary [40, 21].

In our example, suppose we implement the following protocol, where k_{up} is a symmetric key shared between *user* and *proxy*, and k_p is the private signing key of *proxy*.

To appear in the proceedings of the *20th IEEE Computer Security Foundations Symposium (CSF 2007)*, July 7–9, 2007, Venice, Italy.

Message 1 $\quad user \rightarrow proxy \;:\; \mathbf{senc}(song, k_{up})$

Message 2 $\quad proxy \rightarrow store \;:\; \mathbf{sign}(\langle user, store\rangle, k_p)$

Assuming neither k_{up} nor k_p is known to the attacker, we can show this implementation is robustly safe. In the context of the policy (1), the invariant that formula (4) holds in every run assures us that, every time a song is downloaded to a user, a registered user has indeed asked for it. (For the sake of a simple example, we omit any measures to prevent replay attacks.)

This formulation of policy conformance as robust safety generalizes specifications of authentication protocols. In particular, our use of events and expectations generalizes the begin- and end-events introduced by Woo and Lam [48], or the running- and commit-events of Lowe [36]. An end-event labelled with the parameters of a run of a protocol (initiator, responder, session key, and so on) can be understood as an invariant stating that a begin-event with the same label has previously occurred.

Policy Conformance Despite Partial Compromise We say a principal is *compromised* to mean that its privileges can be exercised by the attacker. (Alternative terminology from the literature includes saying such principals are "dishonest" or "non-compliant".) A realistic threat model for a distributed system must include *partial compromise*, the possibility that some of the principals in the system are compromised. Since privileges are often exercised using knowledge of secrets such as passwords, shared encryption keys, and private keys, we assume all such secrets known to a compromised principal are disclosed to the attacker. Partial compromise covers deliberate insider attacks as well as external attackers taking ownership of insiders' assets.

Robust safety essentially requires that no invalid authorization decisions arise. This is too demanding in the presence of partial compromise. Instead, for an implementation to conform to a logical authorization policy, in the presence of partial compromise, we propose the following principle:

> An invalid authorization decision by an uncompromised principal can only arise if principals on which the decision logically depends are compromised.

Hence, the logical policy can be scrutinized *independently of the implementation* to understand the impact of partial compromise. Our principle requires the policy to record all dependencies between principals in the implementation. It may require the policy to be augmented with explicit rules to manifest previously hidden dependencies.

Partial Compromise for the Example Consider the situation where *user* is pre-registered with the *proxy*, but the code running on behalf of *user* receives no instruction to order a song, so that (3) holds but not (2). Partial compromise can lead to the two following attacks. In both, after receiving Message 2, the store incorrectly deduces $user$ **says** $Order(song)$ and hence that $store$ **says** $CanDownload(user, song)$, thus making an invalid authorization decision.

- If *user* but not *proxy* is compromised, the attacker knows k_{up} and can use it to fake Message 1, which causes *proxy* to send Message 2.
- If *proxy* but not *user* is compromised, the attacker knows both k_{up} and k_p and can use k_p to fake Message 2 directly.

Unavoidably, if a system is partially compromised, an authorization decision may be invalid if by policy it depends on a compromised principal. The first attack is an example: the incorrect deduction $user$ **says** $Order(song)$ clearly depends on *user*. Our principle discounts such attacks; the best we can do in such cases is document the possibility, and consider the consequences in security reviews.

On the other hand, an implementation is faulty if it reaches invalid conclusions that do not logically depend on compromised principals. The second attack is an example: the incorrect deduction $user$ **says** $Order(song)$ does not depend on *proxy*. To comply with our principle, either the implementation should be corrected to avoid this dependency, or the policy should be augmented to record it.

Safety Despite Compromised Principals To formalize our principle we consider that all propositions hold at a compromised principal: if b is compromised we assume b **says** C for every C. Let an implementation be *safe despite compromised principals* if and only if, for every choice $\widetilde{b}$ of compromised principals, the system is robustly safe when for every $b \in \{\widetilde{b}\}$,

(1) the principal b is compromised, that is, the attacker knows the secrets of b, and

(2) we augment our policy to allow all propositions at b, that is, we include the formula $\forall X.b$ **says** X.

In our example, the case $\widetilde{b} = user$ corresponds to the first attack above, which we discount: the implementation is robustly safe even though the attacker knows k_{up}, provided we allow $\forall X.user$ **says** X. On the other hand, the case $\widetilde{b} = proxy$ corresponds to the second attack above, which violates our principle: the implementation is not robustly safe if the attacker knows k_{up} and k_p, even if we allow $\forall X.proxy$ **says** X.

The case $\widetilde{b} = proxy$ uncovers a dependency on *proxy* absent from policy (1). Let a **controls** C be short for $(a$ **says** $C) \rightarrow C$. The following formula asserts that, as far as *store* is concerned, the *proxy* has authority over the predicate u **says** $Order(s)$ for all u and s.

$$\forall u, s.\, store \textbf{ says } (proxy \textbf{ controls } (u \textbf{ says } Order(s))) \quad (5)$$

This logical policy clearly documents the implementation dependency. Indeed, our implementation is safe despite compromised principals if the policy consists of both (1) and (5).

Our Contributions Our previous work [23] presents a type system for checking the robust safety of implementations in a process calculus. We address two significant limitations in our previous work: that it does not consider the problem of partial compromise, and that it supports only secure channels and symmetric cryptography. Our main contributions are as follows:

- A systematic study of safety properties despite compromise (in Section 4), including a formulation of *safety despite compromised principals*. Although we phrase our definitions in a process calculus, they could easily be adapted to other formalisms.
- A type system for proving safety despite compromised principals. As previously [23], our type system is phrased in terms of an arbitrary authorization logic. The presentation of our type system is parametric in the choice of constructor and selector operations; the version in this paper includes both public key and symmetric key operations. Typing is largely driven by the reduction rules for cryptography, independently of rules for concurrency.

We emphasise the generality of these definitions, and especially that neither depends on a primitive syntax for principals within the process calculus.

Our main technical results are as follows. Theorem 1 and Theorem 2 are safety and robust safety properties for the type system, generalizing previous results [23]; the notions of safety and robust safety, and these theorems, apply to any choice of authorization logic. Theorem 3 and Theorem 4, with no counterparts in our previous paper or related work [31], explain how we can apply our type system to prove safety despite compromised principals; our notions of safety despite compromise, and these theorems, apply to the example logic of Section 2, and should also apply to any authorization logic with the a **says** C modality [6].

Contents of the Paper Section 2 recalls our notion of authorization logics, and introduces an example. Section 3 recalls a variant of the applied pi calculus, our language for distributed implementations. Section 4 gives a direct, general definition of safety despite compromised principals, and illustrates the definition via a series of examples. Section 5 defines our type system and states its main properties. Section 6 establishes some principles for proving safety despite compromised principals, by typing. Section 7 discusses related work, and Section 8 concludes.

An extended version of this paper, with full proofs, is available as a technical report [24].

2 Review: Authorization Logics

Our process calculus and type system rely on a formal logic, but are largely independent of the exact choice of logic. For the sake of generality, as in earlier work [23], we parameterize our development on a notion of *authorization logic*. This is defined in terms of the names and messages of our process calculus, which is defined in Section 3. The set of messages includes a set of atomic names, as well as a (disjoint) set of variables.

Variables, Messages:

x, y, z	(message) variable
M, N	message, principal

Authorization Logic: $(\mathscr{C}, fn, fv, \vdash)$

An *authorization logic* $(\mathscr{C}, fn, fv, \vdash)$ is a set of formulas $C \in \mathscr{C}$ closed by substitutions σ of messages for variables, with finite sets of *free names* $fn(C)$ and *free variables* $fv(C)$ such that $C\sigma = C$ if $dom(\sigma) \cap fv(C) = \varnothing$ and $fv(C\sigma) \subseteq (fv(C) \setminus dom(\sigma)) \cup fv(\sigma)$; and with an *entailment relation* $S \vdash C$, between sets of formulas $S \subseteq \mathscr{C}$ and formulas $C, C' \in \mathscr{C}$, such that $S \vdash C \Rightarrow S \cup \{C'\} \vdash C$ and $S \vdash C \Rightarrow S\sigma \vdash C\sigma$ and $S \vdash C \wedge S \cup \{C\} \vdash C' \Rightarrow S \vdash C'$.

We present an example authorization logic consisting of CDD [2] augmented with primitive predicates and message quantification. CDD is a cut-down version of Abadi's reading of the Dependency Core Calculus (DCC) [3] as a constructive logic of access control, with constructs for describing principals, access requests, and delegations of authority. These logics include formulas of the form M **says** C, meaning intuitively that principal M has caused formula C to have been said, as in early logical calculi of access control [6]. Garg and Pfenning [25] describe a related constructive authorization logic; in fact, constructivity is incidental to our present purposes. CDD is cut-down in the sense that the principals are a set, while in DCC, they are a lattice. Hence, fewer theorems are provable; for instance, M **says** N **says** C implies N **says** M **says** C in DCC but not in CDD. In our logic, we take the set of principals to be the set of messages of our process calculus. The expected usage is that a principal is either a name or a variable.

Our logic includes formulas of propositional logic, but not negation. The formula X is a propositional variable. The formula $\forall X.C$ is propositional quantification. The formula $\forall x.C$ is message quantification, where x is a variable ranging over messages. The formula $p(M_1, \ldots, M_n)$ is a primitive predicate, parameterized by the messages $M_1, \ldots, M_n$, where p is a predicate symbol. This logic can encode Datalog, the example of our previous work [23].

Syntax of the Example Logic:

$C ::=$	formula
$\quad \mathbf{true}$	true constant
$\quad (C \to C')$	implication
$\quad (C \wedge C')$	conjunction
$\quad (C \vee C')$	disjunction
$\quad (M \ \mathbf{says}\ C)$	says modality
$\quad X$	propositional variable
$\quad (\forall X.C)$	propositional quantification
$\quad (\forall x.C)$	message quantification
$\quad p(M_1,\ldots,M_n)$	primitive predicate
$S ::= \{C_1,\ldots,C_n\}$	set of formulas

We rely on some common abbreviations. We can define falsity via propositional quantification; let **false** be short for $\forall X.X$. The "controls" predicate [6] expresses one form of delegation: let M **controls** C be short for $(M \ \mathbf{says}\ C) \to C$.

Throughout the paper, we identify any phrase of syntax ϕ up to consistent renaming of bound names and variables. We write $fn(\phi)$ and $fv(\phi)$ for the sets of names and variables occurring free in ϕ. We write $fnfv(\phi)$ for $fn(\phi) \cup fv(\phi)$. We write $\phi\{\phi'/x\}$ for the outcome of a capture-avoiding substitution of ϕ' for each free occurrence of x in ϕ. We say ϕ is *closed* when it has no free variables.

We define the entailment relation $S \vdash C$ for our logic by the rules below. We write $\vdash C$ when $\varnothing \vdash C$ is derivable.

Entailment Relation of the Example Logic: $S \vdash C$

$$\frac{C \in S}{S \vdash C} \qquad \frac{}{S \vdash \mathbf{true}}$$

$$\frac{S \cup \{C_1\} \vdash C_2}{S \vdash (C_1 \to C_2)} \qquad \frac{S \vdash (C_1 \to C_2) \quad S \vdash C_1}{S \vdash C_2}$$

$$\frac{S \vdash C_1 \quad S \vdash C_2}{S \vdash (C_1 \wedge C_2)} \qquad \frac{S \vdash (C_1 \wedge C_2)}{S \vdash C_1} \qquad \frac{S \vdash (C_1 \wedge C_2)}{S \vdash C_2}$$

$$\frac{S \vdash C_1}{S \vdash (C_1 \vee C_2)} \qquad \frac{S \vdash C_2}{S \vdash (C_1 \vee C_2)}$$

$$\frac{S \vdash (C_1 \vee C_2) \quad S \cup \{C_1\} \vdash C \quad S \cup \{C_2\} \vdash C}{S \vdash C}$$

$$\frac{S \vdash C}{S \vdash M \ \mathbf{says}\ C} \qquad \frac{S \vdash M \ \mathbf{says}\ C \quad S \cup \{C\} \vdash M \ \mathbf{says}\ C'}{S \vdash M \ \mathbf{says}\ C'}$$

$$\frac{S \vdash C}{S \vdash \forall X.C}\ (X \text{ not free in } S) \qquad \frac{S \vdash \forall X.C}{S \vdash C\{C'/X\}}$$

$$\frac{S \vdash C}{S \vdash \forall x.C}\ (x \text{ not free in } S) \qquad \frac{S \vdash \forall x.C}{S \vdash C\{M/x\}}$$

For instance, we have the following logical entailments:

$\vdash C \to M \ \mathbf{says}\ C$
$\vdash \mathbf{false} \to C$
$\vdash (M \ \mathbf{says}\ M \ \mathbf{says}\ C) \to (M \ \mathbf{says}\ C)$
$\vdash M \ \mathbf{says}\ \mathbf{false} \to M \ \mathbf{says}\ C$

The structure $(\mathscr{C}, fn, fv, \vdash)$, where $\mathscr{C}$ is the set of formulas, is an example authorization logic; we can easily establish the properties (Mon), (Subst), and (Cut).

3 Review: An Applied Pi Calculus

We now define our implementation language, as a variant of the applied pi calculus. Processes are augmented with statements and expectations of logical formulas, whereas messages, in addition to names, can contain a selection of cryptographic functions, a symbolic model of cryptographic primitives [21].

Messages The set of messages is the free algebra built from variables, names and constructors applied to arguments. Usual pi calculus names are, in the technical development, considered as nullary constructors. The **ok** token is a constant used to propagate logical assumptions, by means of the **exercise** destructor. Pairs are modelled by the binary constructor **pair** and the two destructors **fst** and **snd**. To this core, we add a selection of cryptographic operations. The language can be easily extended to include others. Signature checking is modelled by a destructor $\mathbf{verify}(\mathbf{sign}(msg,k),\mathbf{vk}(k))$ which, applied to the signature of msg under k and to the correct verification key for k, yields the message msg. Symmetric encryption is modelled by a destructor $\mathbf{sdec}(\mathbf{senc}(msg,k),k)$ which, applied to the encryption of msg under k and to the symmetric key k, yields the message msg.

Syntax for Messages:

$a,b,c,k,\ldots$	name
$f ::= \mathbf{ok},\mathbf{pair},\mathbf{vk},\mathbf{sign},\mathbf{senc}$	constructor
$g ::= \mathbf{exercise},\mathbf{fst},\mathbf{snd},\mathbf{verify},$ $\mathbf{sdec},\mathbf{check},\mathbf{eq}$	destructor
$M,N ::=$	message, principal
$\quad x$	variable
$\quad a$	name
$\quad f(M_1,\ldots,M_n)$	constructor f applied to $M_1,\ldots,M_n$

Convention: u,v range over names and variables.

The evaluation of destructors is defined by an extensible set of symbolic rules. If it exists, we write $mgu(\widetilde{M} = \widetilde{N})$ for the substitution that is the most general unifier between

the messages $\widetilde{M}$ and $\widetilde{N}$. The relation $g(\widetilde{M}) \mapsto M$ gives base rewrite rules on messages with variables, while the relation $g(\widetilde{M}) \Downarrow M$ is its instantiation.

Most General Unifier: $mgu(\widetilde{M} = \widetilde{N})$

$$mgu(M, \widetilde{M} = N, \widetilde{N}) = \sigma mgu(\widetilde{M}\sigma = \widetilde{N}\sigma) \quad \text{where } \sigma = mgu(M = N)$$
$$mgu(f(\widetilde{M}) = f(\widetilde{N})) = mgu(\widetilde{M} = \widetilde{N})$$
$$mgu(x = N) = mgu(N = x) = \{N/x\} \quad \text{where } x \notin fv(\mathrm{N})$$

Rules for Evaluation: $g(\widetilde{M}) \mapsto M$, $g(\widetilde{M}) \Downarrow M$

$$\mathbf{fst}(\mathbf{pair}(x_1, x_2)) \mapsto x_1$$
$$\mathbf{snd}(\mathbf{pair}(x_1, x_2)) \mapsto x_2$$
$$\mathbf{verify}(\mathbf{sign}(x_1, x_2), \mathbf{vk}(x_2)) \mapsto x_1$$
$$\mathbf{sdec}(\mathbf{senc}(x_1, x_2), x_2) \mapsto x_1$$
$$\mathbf{eq}(x, x) \mapsto x$$
$$\mathbf{exercise}(x) \mapsto x$$
$$g(\widetilde{M}) \mapsto M \wedge \sigma = mgu(\widetilde{M} = \widetilde{N}) \Rightarrow g(\widetilde{N}) \Downarrow M\sigma$$

Processes Our language is essentially the process calculus of Abadi and Blanchet [5], augmented with statements and expectations of logical formulas. A *statement* is simply a formula C that marks a security-related event, such as the instruction to download a song, or the registration of a user, formulas (2) and (3) in Section 1. An *expectation* **expect** C asserts that the formula C should hold in every run. The expectation of formula (4) is the process **expect** $store$ **says** $CanDownload(user, song)$.

To help typechecking, each restricted name is annotated with a type T; the syntax of types is in Section 5.

Syntax for Processes:

$P, Q, R ::=$	process
out $M(N); P$	output of N to channel M
in $M(x); P$	input of x from channel M
!in $M(x); P$	replicated input
new $a{:}T; P$	name restriction
$P \mid Q$	parallel composition
$\mathbf{0}$	inactivity
let $x = g(\widetilde{M})$ **in** P **else** Q	destructor evaluation
C	statement of C
expect C	expectation that C holds

Notations: **new** $\widetilde{a}{:}\widetilde{T}; P \triangleq$ **new** $a_1{:}T_1; \ldots$ **new** $a_n{:}T_n; P$
Let $S = \{C_1, \ldots, C_n\}$. We write $S \mid P$ for $C_1 \mid \ldots \mid C_n \mid P$.

In the processes **in** $M(x); P$ and **!in** $M(x); P$ and **let** $x = g(\widetilde{M})$ **in** P **else** Q, variable x is bound with scope P. In the restriction **new** $a{:}T; P$, name a is bound with scope P.

In examples, we use $\langle M_1, \ldots M_n \rangle$ as a shorthand for $\mathbf{pair}(M_1, \ldots, \mathbf{pair}(M_n, \mathbf{ok}) \ldots)$. The pattern-matching notation **let** $\langle x_1, \ldots, x_n \rangle = M$ **in** P is short for a process that uses the destructors **fst**, **snd**, and **exercise** to unpack the components of the tuple M into the variables $x_1, \ldots, x_n$.

Evaluation Contexts: $K_{\widetilde{a}:\widetilde{T}}[-]$

$K_{\widetilde{a}:\widetilde{T}}[P] \triangleq$ **new** $\widetilde{a}{:}\widetilde{T}; (P \mid P')$ for some P'.

We make the bound names $\widetilde{a}$ and their types $\widetilde{T}$ explicit, in order to constrain the process we put in the hole.

Semantics Next, we present the operational semantics of our calculus via standard structural equivalence ($P \equiv Q$) and reduction ($P \longrightarrow Q$) relations. Structural equivalence states what processes should be considered equivalent up to syntactic re-arrangement. It is the smallest equivalence relation closed under evaluation contexts such that the set of processes with parallel composition and $\mathbf{0}$ constitutes a commutative monoid, satisfying the axioms given below.

Axioms for Structural Equivalence: $P \equiv Q$

$$\mathbf{new}\ a{:}T; (P \mid Q) \equiv P \mid \mathbf{new}\ a{:}T; Q \quad \text{if } a \notin fn(P)$$
$$\mathbf{new}\ a_1{:}T_1; \mathbf{new}\ a_2{:}T_2; P \equiv \mathbf{new}\ a_2{:}T_2; \mathbf{new}\ a_1{:}T_1; P \quad \text{if } a_1 \neq a_2, a_1 \notin fn(T_2), a_2 \notin fn(T_1)$$

Reduction is the smallest relation on closed processes which is closed under structural equivalence and evaluation contexts. Statements and expectations are inert processes; they have no particular rules for reduction or equivalence (although they are affected by other rules).

Axioms for Reduction: $P \longrightarrow P'$

$$\mathbf{out}\ a(M); P \mid \mathbf{in}\ a(x); Q \longrightarrow P \mid Q\{M/x\}$$
$$\mathbf{out}\ a(M); P \mid \mathbf{!in}\ a(x); Q \longrightarrow P \mid Q\{M/x\} \mid \mathbf{!in}\ a(x); Q$$
$$\mathbf{let}\ x = g(\widetilde{M})\ \mathbf{in}\ P\ \mathbf{else}\ Q \longrightarrow P\{M/x\} \quad (\text{if } g(\widetilde{M}) \Downarrow M)$$
$$\mathbf{let}\ x = g(\widetilde{M})\ \mathbf{in}\ P\ \mathbf{else}\ Q \longrightarrow Q \quad (\text{if } g(\widetilde{M}) \not\Downarrow M)$$

Notation: $P \longrightarrow_{\equiv}^{*} P'$ is $P \equiv P'$ or $P \longrightarrow^{*} P'$.

4 Safety Properties Despite Compromise

Safety We recall standard notions of safety and robust safety for processes. Informally, a process is safe when any expectation that may occur in evaluation context follows from statements also in evaluation context; a process is robustly safe when its composition with any process (representing any active adversary) is safe.

Definition 1 (Safety) *A closed process P is* safe *if and only if whenever*

$$P \longrightarrow_{\equiv}^{*} \mathbf{new}\ \tilde{a}{:}\tilde{T}; (\mathbf{expect}\ C \mid P')$$

we have $P' \equiv K_{\tilde{b}:\tilde{U}}[S]$ with $S \vdash C$ and $fn(C) \cap \{\widetilde{b}\} = \varnothing$.

Definition 2 (Opponent) *A process O is* **Un**-typed *if and only if every type occurring in O is* **Un**. *A process O is* **expect**-free *if and only if there is no* **expect** *in O. A process O is an* opponent *if and only if it is* **Un**-*typed and* **expect**-*free.*

In the definition, **Un**-typability is a technicality, explained in the next section, and the absence of **expect**s reflects that we are only concerned about the safety of our process, not of the opponent. For any process, there is an equivalent **Un**-typed process obtained by rewriting the type annotations for bound names and erasing all **expect**s.

Definition 3 (Robust Safety) *A process P is* robustly safe *if and only if $P \mid O$ is safe for all opponents O.*

As usual in the pi calculus, in a process $P \mid O$, the opponent O gets access to every free name of P, and can thus interact with P using communications and cryptography based on these names. In addition, we may want to give the opponent access to some messages that include restricted names, for instance to the public key associated with a private signing key. To this end, we may export those messages as outputs on free names. In case P is of the form $K_{\widetilde{c}:\widetilde{T}}[Q]$ and the opponent initially has access to the messages $M_1,\ldots,M_n$ within the scope of the restricted names $\widetilde{c}$, one thus considers the robust safety of $K_{\widetilde{c}:\widetilde{T}}[Q \mid \prod_{i=1}^{n} \mathbf{out}\ e_i(M_i)]$ for some fresh names $e_1,\ldots,e_n$.

Modelling Partial Compromise We first give a general definition of compromise in systems, then we specialize it to a calculus with explicit trust boundaries for principals.

Definition 4 (Safety Despite Compromise) *A process of the form $P = K_{\widetilde{c}:\widetilde{T}}[Q]$ is* safe despite compromise of Q *if and only if there exist Q' and σ such that:*

(1) *$Q = Q'\sigma$ with $fn(Q') \cap \{\widetilde{c}\} = \varnothing$ and*

(2) *$K_{\widetilde{c}:\widetilde{T}}[Q'\sigma] \mid O$ is safe for all opponents O and Q' such that $fn(Q') \cap \{\widetilde{c}\} = \varnothing$.*

Formally, safety despite compromise is a property of the pair $(K_{\widetilde{c}:\widetilde{T}}[-], Q)$. The definition requires that one identifies a subprocess, Q, representing the part of the system that may be compromised. The free names of Q may include those of P, as well as those bound in the evaluation context $K_{\widetilde{c}:\widetilde{T}}[-]$. The messages of Q that include the names $\widetilde{c}$ represent the additional capabilities released to the opponent as the result of the compromise. Thus, any substitution σ that meets Condition (1) safely collects all such messages, and Condition (2) captures the intuition that Q may then be replaced by arbitrary code with access to the same capabilities.

In simple cases, we can let σ export all the names in $\widetilde{c}$, but usually this is too demanding. Hence, we request the existence of any σ that still enables Q to be expressed under the scope restriction, to account for any partial knowledge Q may have of the restricted names, and we give access to the range of σ. For instance, if $\widetilde{c}$ is a single name s and Q includes only the verification key $\mathbf{vk}(s)$ associated with s, we may apply the definition either for $\sigma = \{s/x\}$ or $\sigma = \{\mathbf{vk}(s)/x\}$, but the latter choice makes it easier to satisfy Condition (2).

As an example, consider a process of the form $P = \mathbf{new}\ s; (P_S \mid P_{V1} \mid P_{V2})$ where P_S is a process that signs messages using the signing key s, and P_{V1} and P_{V2} are processes that verify messages with the verification key $\mathbf{vk}(s)$. Suppose we define our subprocesses as follows:

$$\begin{aligned} P_S &= !\mathbf{in}\ c'(x); (Good(x) \mid \mathbf{out}\ c(\mathbf{sign}(x,s))) \\ P_{V1} = P_{V2} &= !\mathbf{in}\ c(sig); \mathbf{let}\ x = \mathbf{verify}(sig, \mathbf{vk}(s))\ \mathbf{in} \\ &\qquad \mathbf{expect}\ Good(x) \end{aligned}$$

The process P is robustly safe, and is also safe despite compromise of P_{V1} (taking $\sigma = \{\mathbf{vk}(s)/x\}$). Intuitively, this confirms that safety based on signature verification in P_{V2} is not affected by compromise of P_{V1}. Further, we may retain that P is safe despite compromise of P'_{V1} obtained from P_{V1} after receiving a few well-signed signatures. For instance, after two reductions $P \mid \mathbf{out}\ c'(u) \longrightarrow\!\!\!\!\rightarrow P'$ where

$$\begin{aligned} P' &= \mathbf{new}\ s; (P_S \mid Good(u) \mid P'_{V1} \mid P_{V2}) \\ P'_{V1} &= P_{V1} \mid \mathbf{let}\ x = \mathbf{verify}(\mathbf{sign}(u,s), \mathbf{vk}(s))\ \mathbf{in} \\ &\qquad \mathbf{expect}\ Good(x) \end{aligned}$$

the resulting process P' is safe despite compromise of P'_{V1} (using now the substitution $\sigma = \{\mathbf{vk}(s)/x\}\{\mathbf{sign}(u,s)/y\}$). Conversely, P is not safe despite compromise of an active signer such as P_S (it can only be that $\sigma = \{s/x\}$), and P' without $Good(u)$ is not even safe.

The next proposition is an alternative characterization of safety despite compromise in terms of robust safety. This characterization is more specific to the pi calculus. It establishes a proof technique for showing safety despite compromise once σ has been identified. Intuitively, it uses a particular inner opponent $O' = \prod_{x \in dom(\sigma)} \mathbf{out}\ e_x(x)$ that publishes all its knowledge on public channels $\{e_x \mid x \in dom(\sigma)\}$.

Proposition 1 *A process of the form $K_{\widetilde{c}:\widetilde{T}}[Q]$ is* safe despite compromise of Q *if and only if there exist Q', σ, and fresh names $\{e_x \mid x \in dom(\sigma)\}$ such that*

(1) *$Q = Q'\sigma$ with $fn(Q') \cap \{\widetilde{c}\} = \varnothing$ and*

(2) *$K_{\widetilde{c}:\widetilde{T}}[\prod_{x \in dom(\sigma)} \mathbf{out}\ e_x(x\sigma)]$ is robustly safe.*

The proof is by a direct argument based on traces.

Distributed Configurations of Principals We now set a particular syntax for distributed process configurations, or *worlds*. Our configurations consist of named locations that may share bindings and policies. Each location has a name a, interpreted as a principal.

Syntax for Worlds:

$W, V ::=$		world
	$a[P]$	process P controlled by principal a
	new $c{:}T; W$	fresh generation of name c in W
	export $x = M; W$	global binding of variable x to M
	$W \mid V$	parallel composition of W and V
	S	global policy
	$\mathbf{0}$	inactivity

By convention, we assume that principals are not restricted. We let $W[-]$ range over world contexts.

We treat worlds as syntactic sugar for pi processes, as follows. When translating a world, we assume that the variables $\widetilde{x}$ and the names $\widetilde{c}$ are pairwise distinct, and that the names e_x are fresh and pairwise distinct.

- We define a process $[\![P]\!]_a$ to represent the process P running on behalf of principal a, with the translation rules $[\![C]\!]_a \triangleq a$ **says** C and $[\![\textbf{expect } C]\!]_a \triangleq$ **expect** a **says** C plus trivial structural rules for the other cases.
- We define a process $[\![W]\!]$ to represent a world W, with the translation rules $[\![a[P]]\!] \triangleq [\![P]\!]_a$ and $[\![\textbf{export } x = M; W]\!] \triangleq ([\![W]\!] \mid \textbf{out } e_x(x))\{M/x\}$ where $x \notin fv(M)$ plus trivial structural rules for the other cases.

Logically, the translation prefixes all formulas within $a[P]$ with a **says**. Intuitively, this restricts the statements made by a, irrespective of P. We have, for instance, that a **says false** $\mid [\![P]\!]_a$ is trivially robustly safe.

We intend to treat principals as simple units of compromise: following the additional syntax of worlds, we can determine compromised code (Q in Definition 4) from the subset of principals that are deemed compromised:

Definition 5 (Safety Despite Compromised Principals) *Consider a world of the form* $W[Q]$ *where* $Q = \prod_{b \in \widetilde{b}} b[P_b]$. *Its translation is of the form* $[\![W[Q]]\!] = K_{\widetilde{c}:\widetilde{T}}[[\![Q]\!]\sigma]$. *Let* $S_{\widetilde{b}} = \bigwedge_{b \in \widetilde{b}} b$ **says false**.

The world $W[Q]$ *is* safe despite $\widetilde{b}$ *when* $S_{\widetilde{b}} \mid K_{\widetilde{c}:\widetilde{T}}[[\![Q]\!]\sigma]$ *is safe despite compromise of* $[\![Q]\!]\sigma$.

A world is safe despite compromised principals *when it is safe despite* $\widetilde{b}$, *for all subsets* $\{\widetilde{b}\}$ *of its principals.*

In the definition, σ accounts for the substitution of messages for variables bound in **export**s. Each additional formula b **says false** in $S_{\widetilde{b}}$ safely approximates that b is compromised, so that b may say anything. (This formula is equivalent to $\forall X.b$ **says** X.) Without such an assumption, we would not expect W to be safe despite bad principals, because expectations that follow from b **says** ... in the remaining principals may break safety.

Definition 4 leaves open the choice of the parameters σ and Q'. Taking advantage of the structure of worlds, Definition 5 provides guidance for selecting Q'. One can then easily select σ from Q', for instance as the most specific substitution that introduces subterms containing the names $\widetilde{c}$, or simply a substitution from variables to $\{\widetilde{c}\} \cap fn(Q)$.

Example: Ordering a Song via a Proxy We are now ready to analyze distributed implementations of the example policy discussed in Section 1. For brevity, we name our three principals by their initial letters: a user u, a proxy p, and a store s. We consider configurations with the following processes acting for these three principals.

$$
\begin{aligned}
Q_u &= \mathit{Order}(\mathit{georgiaOnMyMind}) \mid \\
&\quad \textbf{out } \mathit{net}(\textbf{senc}(\langle \mathit{georgiaOnMyMind} \rangle, k_{up})) \\
Q_p &= \textbf{in } \mathit{net}(\mathit{cipher}); \\
&\quad \textbf{let } \langle \mathit{song} \rangle = \textbf{sdec}(\mathit{cipher}, k_{up}) \textbf{ in} \\
&\quad \textbf{out } \mathit{request}(\textbf{sign}(\langle u, \mathit{song} \rangle, k_p)) \\
Q_s &= !\textbf{in } \mathit{request}(\mathit{sig}); \\
&\quad \textbf{let } \langle \mathit{user}, \mathit{song} \rangle = \textbf{verify}(\mathit{sig}, v_p) \textbf{ in} \\
&\quad \textbf{expect } \mathit{CanDownload}(\mathit{user}, \mathit{song})
\end{aligned}
$$

The user code Q_u asserts its intent to order a song (event (2) in Section 1) then uses public channel *net* to contact the proxy and shared key k_{up} to authenticate the song order by encryption. The proxy code Q_p receives this ciphertext, decrypts (and thus authenticates) the requested song, then uses a public channel *request* to forward the request to the store, and endorses it by signing the pair u, song. The store code Q_s finally receives the request, verifies its signature, then would trigger the actual song download, guarded by the expectation (4) of Section 1.

Hence, Q_p knows in advance about u, but Q_s does not. Informally, the store is delegating user identification and authentication to proxies. (We may have additional copies of Q_p running in parallel for any number of other users.)

We use the following world context to represent our assumptions on the secrets shared by these principals:

$$
\begin{aligned}
K_{k_{up},k_p}[\,] = \; & \textbf{new } k_{up}{:}\textbf{Key } \langle \mathit{song}{:}\textbf{Un} \rangle \\
& \{u \textbf{ says } \mathit{Order}(\mathit{song})\}; \\
& \textbf{new } k_p{:}\textbf{SK } \langle u{:}\textbf{Un}, \mathit{song}{:}\textbf{Un} \rangle \\
& \{u \textbf{ says } \mathit{Order}(\mathit{song}), p \textbf{ says } \mathit{Registered}(u)\}; \\
& \textbf{export } v_p = \textbf{vk}(k_p); \\
& [\,]
\end{aligned}
$$

(The type annotations are explained in the next section; for now they can be read as the expected events that should hold when using these names.)

Finally, consider the world

$$W = K_{k_{up},k_p}\,[u[Q_u] \mid p[Q_p]] \mid s[Q_s]]$$

and the policy $S = (1) \wedge (3)$, as defined in Section 1. We have the following security properties:

- The system $S \mid W$ is robustly safe.

  Intuitively, this confirms that an outsider, given access to the channels *net* and *request* and to the proxy's signature-verification key $\mathbf{vk}(k_p)$ but not to the keys used by the user and proxy, cannot cause the store to enable any download not authorized by the user and the proxy, as prescribed in S. (Robust safety would clearly collapse without the name restrictions of K_{k_{up},k_p}.)

- The system $S \mid W$ is safe despite u.

  Intuitively, if the user gets compromised, then then any fact u **says** C may hold.

- The system $S \mid W$ is not safe despite p.

  Intuitively, the store delegates to the proxy any user authentication, so a rogue proxy may as well send requests not initiated by u, or requests on behalf of any fake principal, using for instance the code $Q'_p = \mathbf{out}\ request(\mathbf{sign}(\langle user, song\rangle, k_p))$ for any pair $user, song$.

- The system $(S \wedge (5)) \mid W$ is safe despite the compromise of any or all of the principals.

  Instead of (5), we may as well include other variants as long as they imply the delegation from store to proxy embedded in our implementation.

- Alternatively, we may keep the policy and strengthen the protocol, for instance by having the user authenticate its order using a signature verified by the store rather than encryption. Although this is more secure, this may be deemed less efficient, as it forces the store to manage user credentials.

The proofs of the positive statements above are by typing, as detailed in Section 6.

5 A Type System for Authorization

We present a dependent type system for statically checking implementations of authorization policies. Our immediate starting point is our type system for centralized authorization policies [23], and earlier work [30] on subtyping, and public and tainted types, for handling public-key cryptography. The present system features two major improvements. First, it is parametric in the set of constructors and destructors present in the language, and can therefore easily be generalized to new cryptographic operations. In the future, we plan to make this parametricity explicit, by stating sufficient conditions for the acceptability of arbitrary typed evaluation rules for new destructors. Second, we use subtyping to inject in the type system information about the strength of the logical policy present in a typing environment, and hence available as an assumption during the verification pass. Thus, when certain formulas are derivable from the environment, we can relax the logical demands on the effects of cryptographic objects or secure channels.

5.1 Types

We begin with the syntax of types.

Syntax for Types:

$T, U ::=$	type
Ch T	channel for messages of type T
Ok S	ok to assume the formulas S
Pair$(x{:}T, U)$	dependent pair (scope of x is U)
SK T	signing key for T message
Signed T	signature on T message
VK T	verification key for **Signed** T message
Key T	symmetric key for T plaintext
Enc T	ciphertext obtained from T plaintext

U is *generative* iff U is **Ch** T, **Key** T, or **SK** T.

Notation: **Un** $\triangleq$ **Ch Ok** $\varnothing$

The first three types **Ch** T, **Ok** S, and **Pair**$(x{:}T, U)$ form a core type system, independent of any cryptographic operations; the subsequent types are for typing cryptography. The type **Ok** S [28] is populated only if all the formulas in S hold; we can regard S as being a logical representation of a computational effect; hence, our type system is a logical generalization of type and effect systems [26].

The restriction operator **new** $a{:}T; P$, generates fresh names of type T, and is well-typed only if T is one of the generative types. The derived type **Un** represents messages that can flow to or from the attacker; it is a type of public, untrusted data.

Derived Tuple Notations Recall our use of $\langle M_1, \ldots M_n\rangle$ as a shorthand for $\mathbf{pair}(M_1, \ldots, \mathbf{pair}(M_n, \mathbf{ok}) \ldots)$. In our type system, this message is of the type $\langle x_1 : T_1, \ldots, x_n : T_n\rangle S$, a shorthand for $\mathbf{Pair}(x_1{:}T_1, \ldots \mathbf{Pair}(x_n{:}T_n, \mathbf{Ok}\ S) \ldots)$, where each M_i has type T_i and the **ok** carries the formula $S\{M_1/x_1\} \ldots \{M_n/x_n\}$, which must hold for the tuple to be well-typed. The pattern-matching notation

let $\langle x_1,\ldots,x_n\rangle = M$ **in** P is short for a process that uses the destructors **fst**, **snd**, and **exercise** to unpack the components of the tuple M of type $\langle x_1{:}T_1,\ldots,x_n{:}T_n\rangle S$ into the variables $x_1,\ldots,x_n$. At the type level, the formula S can be assumed when typing the continuation P.

Derived Notations for Tuples with Effects:

$\langle M_1,\ldots,M_n\rangle \triangleq \mathbf{pair}(M_1,\ldots,\mathbf{pair}(M_n,\mathbf{ok})\ldots) \quad \langle\rangle \triangleq \mathbf{ok}$
$\langle x_1{:}T_1,\ldots,x_n{:}T_n\rangle S \triangleq \mathbf{Pair}(x_1{:}T_1,\ldots,\mathbf{Pair}(x_n{:}T_n,\mathbf{Ok}\ S))$
$\langle\rangle S \triangleq \mathbf{Ok}\ S$
let $\langle x_{n+1},\ldots,x_1\rangle = D$ **in** P **else** $Q \triangleq$
  let $x_{n+1} = \mathbf{fst}(D)$ **in**
  let $\langle x_n,\ldots,x_1\rangle = \mathbf{snd}(D)$ **in** P **else** Q
let $\langle\rangle = D$ **in** P **else** $Q \triangleq$
  let $z = \mathbf{exercise}(D)$ **in** P **else** Q $\quad z$ fresh

5.2 Core Type System

Our type system consists of a set of inductively defined judgments. Each judgment includes an *environment*, which defines the types of names and variables in scope, together with assumed formulas.

Syntax for Environments:

$E ::=$	environment
$\varnothing$	empty
$E,u{:}T$	u has type T
E,C	C is a valid formula

Notation: $E(u) = T$ if $E = E',u{:}T,E''$.
$E = a_1{:}T_1,\ldots,a_n{:}T_n$ is *generative* iff each T_i is generative.

We assume standard notions of $fn, fv, fnfv, dom$ for environments. We define a function $formulas(E)$ which returns the formulas appearing at top level in E.

Functions: $fn(E)$, $dom(E)$ **and** $formulas(E)$

$fnfv(E,C) = fn(E) \cup fnfv(C)$
$fnfv(E,u{:}T) = fnfv(E) \cup fnfv(T) \quad fnfv(\varnothing) = \varnothing$
$dom(E,C) = dom(E) \quad dom(E,u{:}T) = dom(E) \cup \{u\}$
$dom(\varnothing) = \varnothing$
$formulas(E,C) = formulas(E) \cup \{C\}$
$formulas(E,u{:}T) = formulas(E) \quad formulas(\varnothing) = \varnothing$

We define a function $env(P)$ on processes which returns an environment containing the restrictions and clauses appearing at the top level in P.

Environment of Processes: $env(P)$

$env(\mathbf{0})^{\varnothing} = \varnothing$
$env(\mathbf{new}\ a{:}T;P)^{a,\widetilde{a}} = a{:}T, env(P)^{\widetilde{a}} \quad (\{\widetilde{a}\} \cap fn(P) = \varnothing)$
$env(P \mid Q)^{\widetilde{a},\widetilde{b}} = env(P)^{\widetilde{a}}, env(Q)^{\widetilde{b}} \quad (\{\widetilde{a},\widetilde{b}\} \cap fn(P \mid Q) = \varnothing)$
$env(\mathbf{expect}\ C)^{\varnothing} = \varnothing$
$env(C)^{\varnothing} = C$
$env(!\mathbf{in}\ M(x);P)^{\varnothing} = \varnothing$
$env(\mathbf{in}\ M(x);P)^{\varnothing} = \varnothing$
$env(\mathbf{out}\ M(N);P)^{\varnothing} = \varnothing$
$env(\mathbf{let}\ x = g(\widetilde{M})\ \mathbf{in}\ P\ \mathbf{else}\ Q)^{\varnothing} = \varnothing$
Convention: $env(P) \triangleq env(P)^{\widetilde{a}}$ for some distinct $\widetilde{a}$ such that $env(P)^{\widetilde{a}}$ is defined.

Judgments of the Type System:

$E \vdash \diamond$	environment E is well-formed
$E \vdash T{::}Public$	in environment E, type T has kind public
$E \vdash T{::}Tainted$	in environment E, type T has kind tainted
$E \vdash T{<:}U$	in environment E, T is a subtype of U
$E \vdash M : T$	in environment E, message M has type T
$E \vdash P$	in environment E, process P is well-typed

An environment is well-formed when all the names and variables being declared are distinct, and when all its free names and free variables are declared.

Rule for Environments: $E \vdash \diamond$

$$\frac{fnfv(E) \subseteq dom(E) \quad \text{if } E = E',u{:}T,E'', \text{ then } u \notin (dom(E',E'') \cup fv(T))}{E \vdash \diamond}$$

The kinding judgments $T{::}Public$ and $T{::}Tainted$ describe whether values of type T can flow to and from the opponent, respectively. They contribute to generate the subtyping judgment.

Kinding Rules: $E \vdash T{::}Public, E \vdash T{::}Tainted$

$$\frac{E \vdash T{::}Public \quad E \vdash T{::}Tainted}{E \vdash \mathbf{Ch}\ T{::}Public}$$

$$\frac{E \vdash T{::}Public \quad E \vdash T{::}Tainted}{E \vdash \mathbf{Ch}\ T{::}Tainted}$$

$$\frac{E \vdash \diamond \quad fnfv(S) \subseteq dom(E)}{E \vdash \mathbf{Ok}\ S{::}Public}$$

$$\frac{E \vdash \diamond \quad fnfv(S) \subseteq dom(E) \quad formulas(E) \vdash C \quad \forall C \in S}{E \vdash \mathbf{Ok}\ S{::}Tainted}$$

$$\frac{E \vdash T{::}Public \quad E,x{:}T \vdash T'{::}Public}{E \vdash \mathbf{Pair}(x{:}T,T'){::}Public}$$

$$\frac{E \vdash T{::}Tainted \quad E,x{:}T \vdash T'{::}Tainted}{E \vdash \mathbf{Pair}(x{:}T,T'){::}Tainted}$$

The subtyping judgment $E \vdash T <: U$ is not generally needed to type well-behaved processes, but is useful to type partially compromised processes. The subtyping rules depend on the kinding judgments and on logical entailment.

Subtyping Rules: $E \vdash T <: U$

Notation: $E \vdash T <:> U$ means $E \vdash T <: U \wedge E \vdash U <: T$.

$$\frac{E \vdash T :: \mathit{Public} \quad E \vdash U :: \mathit{Tainted}}{E \vdash T <: U}$$

$$\frac{E \vdash T_1 <: U_1 \quad E, x{:}T_1 \vdash T_2 <: U_2}{E \vdash \mathbf{Pair}(x{:}T_1, T_2) <: \mathbf{Pair}(x{:}U_1, U_2)}$$

$$\frac{\mathit{fnfv}(S) \subseteq \mathit{dom}(E) \quad \mathit{formulas}(E) \cup S \vdash C \quad \forall C \in S'}{E \vdash \mathbf{Ok}\ S <: \mathbf{Ok}\ S'}$$

$$\frac{E \vdash T <:> U}{E \vdash \mathbf{Ch}\ T <: \mathbf{Ch}\ U}$$

The rules for the typing judgment for messages $E \vdash M : T$ are mostly standard. The rule for typing constructors relies on a type signature $f : (T_1, \ldots, T_n) \mapsto T$, relating the type of a term with the type of its argument, described in the next section. The rule for **ok** tokens, which are supposed to convey the logical effects annotated in the type, requires that the environment contains enough formulas to guarantee that those effects hold.

Typing Rules for Messages: $E \vdash M : T$

$$\frac{E \vdash M : U \quad E \vdash U <: T}{E \vdash M : T} \qquad \frac{E \vdash \diamond \quad u \in \mathit{dom}(E)}{E \vdash u : E(u)}$$

$$\frac{E \vdash M_i : T_i \quad \forall i \in 1..n \quad f : (T_1, \ldots, T_n) \mapsto T}{E \vdash f(M_1, \ldots, M_n) : T}$$

$$\frac{E \vdash M_1 : T_1 \quad E \vdash M_2 : T_2\{M_1/x\}}{E \vdash \mathbf{pair}(M_1, M_2) : \mathbf{Pair}(x{:}T_1, T_2)}$$

$$\frac{E \vdash \diamond \quad \mathit{fnfv}(S) \subseteq \mathit{dom}(E) \quad \mathit{formulas}(E) \vdash C \quad \forall C \in S}{E \vdash \mathbf{ok} : \mathbf{Ok}\ S}$$

In particular, we can check that $\mathbf{Un} \triangleq \mathbf{Ch}\ \mathbf{Ok}\ \varnothing$ is a type for values that can be freely used by the opponent: we have $E \vdash M : \mathbf{Un}$ if and only $E \vdash M :: \mathit{Public}$ and $E \vdash M :: \mathit{Tainted}$.

The rules for the typing judgment for processes $E \vdash P$ are inspired by those in our previous work [23]. From that work, there are three rules of particular interest. The rule for expectations **expect** C requires that C is entailed by the current environment. The rule for statements allows any statement C, provided its names are in scope. The rule for parallel composition allows $P \mid Q$, provided that P and Q are well-typed given the top-level statements of Q and P, respectively.

Typing Rules for Processes: $E \vdash P$

$$\frac{E \vdash \diamond}{E \vdash \mathbf{0}} \qquad \frac{E, a{:}T \vdash P \quad T \text{ generative}}{E \vdash \mathbf{new}\ a{:}T; P}$$

$$\frac{E, \mathit{env}(Q) \vdash P \quad E, \mathit{env}(P) \vdash Q \quad \mathit{fnfv}(P \mid Q) \subseteq \mathit{dom}(E)}{E \vdash P \mid Q}$$

$$\frac{E, C \vdash \diamond \quad \mathit{formulas}(E) \vdash C}{E \vdash \mathbf{expect}\ C} \qquad \frac{E, C \vdash \diamond}{E \vdash C}$$

$$\frac{E \vdash \mathbf{in}\ M(x); P}{E \vdash \mathbf{!in}\ M(x); P} \qquad \frac{E \vdash M : \mathbf{Ch}\ T \quad E, x{:}T \vdash P}{E \vdash \mathbf{in}\ M(x); P}$$

$$\frac{E \vdash M : \mathbf{Ch}\ T \quad E \vdash N : T \quad E \vdash P}{E \vdash \mathbf{out}\ M(N); P}$$

$$\frac{\begin{array}{l}\widetilde{y}{:}\widetilde{U} \vdash g(\widetilde{N}) \mapsto N : (\widetilde{T}) \mapsto T, S \qquad E \vdash Q \\ \text{if } \mathit{mgu}(\widetilde{M} = \widetilde{N}) \text{ exists, then} \\ E \vdash \widetilde{M} : \widetilde{T} \qquad (E, \widetilde{y}{:}\widetilde{U}, S \vdash P\{N/x\})(\mathit{mgu}(\widetilde{M} = \widetilde{N}))\end{array}}{E \vdash \mathbf{let}\ x = g(\widetilde{M})\ \mathbf{in}\ P\ \mathbf{else}\ Q}$$

The rule for destructor evaluations is a novelty of this type system. It relies on the typed evaluation for destructors $\widetilde{y}{:}\widetilde{U} \vdash g(\widetilde{N}) \mapsto N : (\widetilde{T}) \mapsto T, S$, relating input and output types $((\widetilde{T}) \mapsto T)$ with the types of free variables $\widetilde{U}$ and the effects S. These rules, defined below, extend the ones for evaluation given in Section 3. (The rules for cryptographic operations are given in Section 5.3.)

Typed Evaluation: $E \vdash g(\widetilde{M}) \mapsto M : (\widetilde{T}) \mapsto T, S; g(\widetilde{M}) \Downarrow M$

$$x_1{:}T_1, x_2{:}T_2 \vdash \mathbf{fst}(\mathbf{pair}(x_1, x_2)) \mapsto x_1 : (\mathbf{Pair}(x_1{:}T_1, T_2)) \mapsto T_1, \varnothing$$

$$x_1{:}T_1, x_2{:}T_2 \vdash \mathbf{snd}(\mathbf{pair}(x_1, x_2)) \mapsto x_2 : (\mathbf{Pair}(x_1{:}T_1, T_2)) \mapsto T_2, \varnothing$$

$$x{:}T \vdash \mathbf{eq}(x, x) \mapsto x : (T, U) \mapsto T, \varnothing$$

$$x{:}\mathbf{Ok}\ S \vdash \mathbf{exercise}(x) \mapsto x : (\mathbf{Ok}\ S) \mapsto \mathbf{Ok}\ S, S$$

$$\frac{\widetilde{x}{:}\widetilde{U} \vdash g(\widetilde{N}) \mapsto N : (\widetilde{T}) \mapsto T, S \quad \sigma = \mathit{mgu}(\widetilde{M} = \widetilde{N})}{g(\widetilde{M}) \Downarrow N\sigma}$$

Going back to the rule for destructor evaluation, if there is a most general unifier σ for the parameters of g, $\widetilde{M}$ and $\widetilde{N}$, the rule requires that the continuation process $P\{N/x\}$, where the symbolic result of destructing g is bound to x, is well-typed in an environment enriched with the definition of the fresh variables $\widetilde{y}$ and the effects S introduced by the typed evaluation rule, taking into account σ.

As an example, let us specialize the typing rule for the destructor $g = \mathbf{eq}$, by eliminating the typed evaluation $x{:}U_1 \vdash \mathbf{eq}(x, x) \mapsto x : (U_1, U_2) \mapsto U_1, \varnothing$. After simplification, we obtain the derived typing rule

$$\frac{\begin{array}{l}E \vdash Q \\ \text{if } mgu(M_1 = M_2) \text{ exists, then} \\ E \vdash M_1 : U_1 \quad E \vdash M_2 : U_2 \\ (E \vdash P\{M_1/x\})mgu(M_1 = M_2)\end{array}}{E \vdash \textbf{let } x = \textbf{eq}(M_1, M_2) \textbf{ in } P \textbf{ else } Q}$$

5.3 Typing Cryptographic Operations

We augment our core type system with additional rules for signatures and encryption. We could accommodate further cryptographic operations by including additional sets of rules, in this style. Each cryptographic operation requires new types, constructors, a destructor, rules for kinding, subtyping, typing, and typed evaluation.

Signatures A private signature key k for plaintexts of type T is a name of the generative type $\textbf{SK}\ T$. A signature for a plaintext t of type T under key k is the term $\textbf{sign}(t,k)$ of type $\textbf{Signed}\ T$. Its type signature is $\textbf{sign} : (T, \textbf{SK}\ T) \mapsto \textbf{Signed}\ T$. The public verification key for a signature key k for plaintexts of type T is the term $\textbf{vk}(k)$ of type $\textbf{VK}\ T$. Its type signature is $\textbf{vk} : (\textbf{SK}\ T) \mapsto \textbf{VK}\ T$. The typed evaluation rule for signatures, which brings this information together, is:

$$x_1{:}T, x_2{:}\textbf{SK}\ T \vdash \textbf{verify}(\textbf{sign}(x_1,x_2), \textbf{vk}(x_2)) \mapsto x_1 : (\textbf{Signed}\ T, \textbf{VK}\ T) \mapsto T, \varnothing$$

Kinding and Subtyping Rules for Signing:

$$\frac{E \vdash T :: Public \quad E \vdash T :: Tainted}{E \vdash \textbf{SK}\ T :: Public} \qquad \frac{E \vdash T :: Public \quad E \vdash T :: Tainted}{E \vdash \textbf{SK}\ T :: Tainted}$$

$$\frac{E \vdash T :: Public}{E \vdash \textbf{VK}\ T :: Public} \qquad \frac{E \vdash T :: Tainted}{E \vdash \textbf{VK}\ T :: Tainted}$$

$$\frac{E \vdash T :: Public}{E \vdash \textbf{Signed}\ T :: Public} \qquad \frac{E \vdash \diamond \quad fnfv(T) \subseteq dom(E)}{E \vdash \textbf{Signed}\ T :: Tainted}$$

$$\frac{E \vdash T <: > U}{E \vdash \textbf{SK}\ T <: \textbf{SK}\ U} \qquad \frac{E \vdash T <: U}{E \vdash \textbf{VK}\ T <: \textbf{VK}\ U}$$

$$\frac{E \vdash T <: > U}{E \vdash \textbf{Signed}\ T <: \textbf{Signed}\ U}$$

Symmetric Encryption A symmetric encryption key k for plaintexts of type T is a name of the generative type $\textbf{Key}\ T$. A symmetric encryption for a plaintext t of type T under key k is the term $\textbf{senc}(t,k)$ of type $\textbf{Enc}\ T$. Its type signature is $\textbf{senc} : (T, \textbf{Key}\ T) \mapsto \textbf{Enc}\ T$. The typed evaluation rule for symmetric decryption is:

$$x_1{:}T, x_2{:}\textbf{Key}\ T \vdash \textbf{sdec}(\textbf{senc}(x_1,x_2),x_2) \mapsto x_1 : (\textbf{Enc}\ T, \textbf{Key}\ T) \mapsto T, \varnothing$$

Kinding and Subtyping Rules for Encryption:

$$\frac{E \vdash T :: Public \quad E \vdash T :: Tainted}{E \vdash \textbf{Key}\ T :: Public} \qquad \frac{E \vdash T :: Public \quad E \vdash T :: Tainted}{E \vdash \textbf{Key}\ T :: Tainted}$$

$$\frac{E \vdash \diamond \quad fnfv(T) \subseteq dom(E)}{E \vdash \textbf{Enc}\ T :: Public} \qquad \frac{E \vdash \diamond \quad fnfv(T) \subseteq dom(E)}{E \vdash \textbf{Enc}\ T :: Tainted}$$

$$\frac{E \vdash T <: > U}{E \vdash \textbf{Key}\ T <: \textbf{Key}\ U} \qquad \frac{E \vdash T <: > U}{E \vdash \textbf{Enc}\ T <: \textbf{Enc}\ U}$$

5.4 Results

Our first theorem is that well-typed processes are safe; to prove it, we rely on a lemma that both structural congruence and reduction preserve the process typing judgment.

Lemma 1 (Type Preservation) *If $E \vdash P$, P is closed and either $P \equiv P'$ or $P \rightarrow^* P'$ then $E \vdash P'$.*

Theorem 1 (Safety) *If $E \vdash P$ and E is generative, then P is safe.*

Our second theorem is that well-typed processes whose free names can be considered both public and tainted, that is, of type $\textbf{Un}$, are robustly safe. It follows from the first via an auxiliary lemma that any opponent process can be typed by assuming its free names are typable $\textbf{Un}$.

Lemma 2 (Opponent Typability) *If $E \vdash \diamond$ and $fnfv(O) = \widetilde{u}$ and $E \vdash \widetilde{u}{:}\textbf{Un}$ then $E \vdash O$, for all opponents O.*

Theorem 2 (Robust Safety) *If $E \vdash P$ and E is generative and $E, env(P) \vdash a : \textbf{Un}$ for all $a \in dom(E)$, then P robustly safe.*

For generic reasons, the converse of this theorem is false, that is, the type system is incomplete. For example, we cannot type a process that contains expectation of an unstated fact, even if the expectation is unreachable.

6 Applying the Type System

We now provide sufficient typed-based conditions for establishing safety despite compromise. Using typing, as well as the conditions for robust safety, we essentially have to show that the variables in the domain of σ can be $\textbf{Un}$-typed.

Theorem 3 (Safety Despite Compromise, by Typing) *A process of the form $K_{\widetilde{c}:\widetilde{T}}[Q]$ is* safe despite compromise of *Q if there exist Q', E, σ such that*

(1) $Q = Q'\sigma$ *with* $fn(Q') \cap \{\widetilde{c}\} = \varnothing$;

(2) $E \vdash K_{\widetilde{c}:\widetilde{T}}[0]$ *with* $E, env(K_{\widetilde{c}:\widetilde{T}}[0]) \vdash a : \mathbf{Un}$ *for every* $a \in dom(E)$;

(3) $E, env(K_{\widetilde{c}:\widetilde{T}}[0]) \vdash x\sigma : \mathbf{Un}$ *for every* $x \in dom(\sigma)$; *and*

(4) E *is generative.*

The proof is by appeal to Proposition 1 and Theorem 2. The environment E only binds free names a; condition (2) requires that they are **Un**-typed, possibly by "logical" subtyping.

Similarly, we can rely on typing for establishing safety properties for configurations of partly-compromised principals, as follows:

Theorem 4 (Safety Despite Compromised Principals, by Typing) *Consider a world of the form*

$$W = S \mid \mathbf{new}\ \widetilde{c}{:}\widetilde{T_c}; \widetilde{\mathbf{export}\ z = M_z}; \prod_{a \in \widetilde{a}} Q_a$$

where, for simplicity, S collects all the logical statements of W in evaluation context. Let E be the environment that maps $fn(W)$ to **Un**.

- *W is robustly safe if*

  (1) $E, S, \widetilde{c{:}T_c} \vdash M_z : \mathbf{Un}$ *for each exported variable z;*

  (2) $E, S, \widetilde{c{:}T_c}, \widetilde{z{:}\mathbf{Un}} \vdash [\![Q_a]\!]_a$ *for each principal a of W.*

- *For some principals $\widetilde{b} \subseteq \widetilde{a}$, let $Q = \prod_{b \in \widetilde{b}} Q_b$ and $S_{\widetilde{b}} = \bigwedge_{b \in \widetilde{b}} b$* **says false**. *Suppose there are Q' and σ such that (i) $Q = Q'\sigma$ and (ii) $fn(Q') \cap \{\widetilde{c}\} = \varnothing$.*

  W is safe despite $\widetilde{b}$ if, moreover,

  (3) $E, S, \widetilde{c{:}T_c}, S_{\widetilde{b}}, \widetilde{z{:}\mathbf{Un}} \vdash x\sigma : \mathbf{Un}$ *for each $x \in dom(\sigma)$*

Although the proof technique still requires identifying σ, its choice is usually routine. For instance, we may simply let σ substitute the names $c \in \{\widetilde{c}\} \cap fn(Q)$ for a distinct fresh variable. Then, condition (3) simply becomes $E, S, \widetilde{c{:}T_c}, S_{\widetilde{b}} \vdash c : \mathbf{Un}$ for each name $c \in \{\widetilde{c}\} \cap fn(Q)$.

Also, note that we only need to establish conditions (1) and (2) once for all $\widetilde{b}$. In particular, each local code Q_a is typed only once. The additional requirements focus on (a safe approximation of) the terms leaked by the compromised principals $\widetilde{b}$.

Ordering a Song via a Proxy: the Typed Story We are now ready to verify by typing the properties claimed for our example implementation of Section 4. Using the notations and numbering of Theorem 4, the partial environment $\widetilde{c{:}T_c}$ binds two names:

$$k_{up} : \mathbf{Key}\ \langle song{:}\mathbf{Un}\rangle\{u\ \mathbf{says}\ Order(song)\}$$

$$k_p : \mathbf{SK}\ \langle u{:}\mathbf{Un}, song{:}\mathbf{Un}\rangle \{u\ \mathbf{says}\ Order(song), p\ \mathbf{says}\ Registered(u)\}$$

Let E map the free names of W to **Un**. Let $S = (1) \wedge (3)$. Let $E' = E, S, \widetilde{c{:}T_c}$.

(1) For our single exported term, we have $E' \vdash \mathbf{vk}(k_p) : \mathbf{Un}$.

In more detail, let $S' = \{u\ \mathbf{says}\ Order(song), p\ \mathbf{says}\ Registered(u)\}$. By (Sub Ok) and (Sub Pair), we have $E' \vdash \langle u{:}\mathbf{Un}, song{:}\mathbf{Un}\rangle S' <: \langle u{:}\mathbf{Un}, song{:}\mathbf{Un}\rangle \varnothing <: \mathbf{Un}$.

By (Msg Msg) for **vk**, we have $E' \vdash \mathbf{vk}(k_p) : \mathbf{VK}\ \langle u{:}\mathbf{Un}, song{:}\mathbf{Un}\rangle S'$.

By (Sub VK), $E' \vdash \mathbf{vk}(k_p) : \mathbf{VK}\ \mathbf{Un}$ and, by kinding of **VK** , $E' \vdash \mathbf{vk}(k_p) : \mathbf{Un}$.

(2) We type our local processes, as follows:

- For the user, we have $E', v_p{:}\mathbf{Un} \vdash [\![Q_u]\!]_u$.

  In more detail, $[\![Q_u]\!]_u = S' \mid \mathbf{out}\ net(\mathbf{senc}(\langle georgiaOnMyMind\rangle, k_{up}))$ where $S' = \{u\ \mathbf{says}\ Order(georgiaOnMyMind)\}$ and we have:

  $E', S' \vdash \langle georgiaOnMyMind\rangle : T$ where $T = \langle song\rangle\{u\ \mathbf{says}\ Order(song)\}$, by (Msg Ok) and (Msg Pair).

  $E', S' \vdash \mathbf{senc}(\langle georgiaOnMyMind\rangle, k_{up}) : \mathbf{Enc}\ T$ by (Msg Msg) for **senc**.

  $E', S' \vdash \mathbf{senc}(\langle georgiaOnMyMind\rangle, k_{up}) : \mathbf{Un}$ by kinding of **senc**.

  $E', S' \vdash net : \mathbf{Un}$ hence $E', S' \vdash net : \mathbf{Ch}\ \mathbf{Un}$ by (Sub Ext) and (Sub Ch).

  $E', S' \vdash \mathbf{out}\ net(\mathbf{senc}(\langle georgiaOnMyMind\rangle, k_{up}))$ by (Proc Output).

  $E' \vdash S' \mid \mathbf{out}\ net(\mathbf{senc}(\langle georgiaOnMyMind\rangle, k_{up}))$ by (Proc Par).

- For the proxy, we have $E', v_p{:}\mathbf{Un} \vdash [\![Q_p]\!]_p$. In particular, we use both policy (3) and the effect u **says** $Order(song)$ from successful decryption with k_{up} in order to type the message signing.

- For the store, we have $E', v_p{:}\mathbf{Un} \vdash [\![Q_s]\!]_s$. In particular, we use policy (1) to type **expect** s **says** (4) in the process translation.

This establishes that $S \mid W$ of Section 4 is robustly safe.

(3) Moreover, we obtain safety despite subsets of $\{u, p, s\}$ as follows:

- Safety despite $\{s\}$: we simply use an empty σ.

- Safety despite $\{u\}$: we have E', u **says false** $\vdash k_{up} :$ **Un**, using u **says false** to show that u **says** $Order(song)$. Thus, we obtain Property (3) for $\sigma = \{k_{up}/x\}$ and $Q'_u = Q_u\{k_{up}/x\}$. We similarly obtain safety despite $\{u,s\}$.
- Conversely, we cannot derive $k_p :$ **Un**, even when p **says false**, as indeed $S \mid W$ is not safe despite p.

With the addition of the explicit delegation policy (5), we can complete our proof of safety despite compromised principals by showing that:

$$E', (5), p \textbf{ says false} \vdash k_p : \textbf{Un}$$
$$E', (5), p \textbf{ says false} \vdash k_{up} : \textbf{Un}$$

and verifying safety despite all subsets of $\{u,p,s\}$, including those that contain p.

7 Related Work

There is an extensive literature [6, 16, 42, 22, 7, 34, 33, 11, 19, 45, 35, 8, 20, 9, 10, 32] on logics for access control and authorization. Often, as in the earliest logic-based implementations of access control [47, 16], the intention is that a logical policy be evaluated directly at run time. In contrast, we view a logical policy as a specification for authorization decisions, and establish type-based techniques for showing that the code of a system conforms to this specification. As we have shown in a previous paper [23], we can represent and typecheck a range of implementation strategies including, but not limited to, direct evaluation of policy clauses at run time.

Another contrast with prior work on authorization logics is that our embedding of policies, principals, and authorization decisions within a process calculus allows us to consider formally the impact of partial compromise of implementation code. We are not aware of any prior formal studies of the impact of partial compromise on the validity of authorization decisions at uncompromised nodes.

On the other hand, partial compromise is certainly considered in most formal models of cryptographic protocols. For example, Lowe [36] assumes the attacker is itself a recognised principal of the system. In Paulson's inductive method [41], the attacker knows the long term keys corresponding to a set of "bad" principals. Often, invariants need to be individually adjusted to account for partial compromise; in our notation, we might weaken our expectation to something like

$$\textbf{expect } (store \textbf{ says } CanDownload(user, song)) \vee Bad(user) \vee Bad(proxy)$$

where $Bad(a)$ holds in case principal a is compromised. Instead, our notions of safety despite compromise and safety despite compromised principals are systematic generalizations of such representations of bad principals; invariants are left as intended, and any adjustment needed to account for implementation dependencies is an explicit weakening of the global policy. We leave to future work a formal comparison between our definition and those used for cryptographic protocols.

Our work builds on techniques from some of the prior type systems for proving secrecy [1, 4, 5] and authenticity properties [29, 28, 30] of cryptographic protocols expressed in process calculi. A precursor of our work [31] considers certain conditional secrecy properties within a pi calculus in the presence of partial compromise; as well as authorization properties, the additional features in the present paper include our systematic notion of safety despite compromise, and the generic treatment of multiple constructors and destructors in our type system. (For the sake of space, this paper only treats authentication and authorization properties, but we conjecture that conditional secrecy could easily be added.) Bugliesi, Focardi, and Maffei [17] check security properties in the presence of a fixed set of compromised hosts, but assume this set is known statically.

The decentralized label model (DLM) of Myers and Liskov [39] is the basis of the JFlow and Jif languages in which security types track the ownership of data. Secure program partitioning [49] is an implementation technique for Jif that takes partial compromise into account. Chothia, Duggan, and Vitek [18] propose a type system combining cryptography with DLM, in the setting of a process calculus, but do not consider partial compromise.

Declassification occurs when previously sensitive data is deliberately leaked by a system to its environment. Our formulation of partial compromise is a form of declassification in which all the data currently held by a principal is leaked. There is a large literature [44] on establishing confidentiality, formalized as various liberalizations of information flow properties [43], in the presence of declassification. It would be instructive to find formal connections between these liberalizations and our logic-based notions of safety despite compromise.

8 Conclusions

We have embedded a logic within a process calculus so as to annotate implementation code with events and invariants reflecting authorization decisions in the code. For each principal a, logical formulas a **says** C represent intended invariants in the process code $a[P]$ acting for a. We advocate logical policies as a specification for an implementation, and present a notion of conformance between an implementation and a policy that, unlike prior work, systematically accounts for the possibility that some principals are compromised. We can prove policy conformance via a type

system that supports a wide range of cryptographic primitives.

Two criticisms of our work are that we develop a theory and type system for a pi calculus rather than for a conventional programming language, and that we have typed only a small set of examples (those in this paper and its precursor [23], such as various recursive routines to check certificate chains). In fact, the pi calculus and its variants are an effective setting for developing security foundations, and various tools [15, 27, 37] exist to analyze security properties of processes. Moreover, although superficially dissimilar to typical code, pi calculi can represent the semantics of a range of programming languages [38, 46]. Via such representations, security tools for pi calculi have been applied to analyze executable code [12, 14]. In future, we intend to add a typechecker, based on the foundations in this paper, to an existing tool [14] for checking security properties of distributed code, and to assess the typechecker against an existing codebase [13].

Acknowledgment Martín Abadi made available an unpublished paper [2].

References

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

[2] M. Abadi. Access control in a core calculus of dependency. In *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin*, pages 5–31. Elsevier, 2007. Volume 172 of ENTCS.

[3] M. Abadi, A. Banarjee, N. Heintz, and J. G. Riecke. A core calculus of dependency. In *26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 147–160, 1999.

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

[5] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.

[6] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

[7] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, 1999.

[8] J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security*, 5(4):492–540, 2002.

[9] L. Bauer. *Access Control for the Web via Proof-carrying Authorization*. PhD thesis, Princeton University, 2003.

[10] M. Y. Becker and P. Sewell. Cassandra: flexible trust management, applied to electronic health records. In *17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 139–154, June 2004.

[11] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 41–52, New York, NY, USA, 2001. ACM Press.

[12] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In *11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 268–277, Oct. 2004.

[13] K. Bhargavan, C. Fournet, and A. D. Gordon. Verified reference implementations of WS-Security protocols. In *3rd International Workshop on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 88–106. Springer, 2006.

[14] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152, 2006.

[15] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96, 2001.

[16] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE 17th Symposium on Research in Security and Privacy*, pages 164–173, 1996.

[17] M. Bugliesi, R. Focardi, and M. Maffei. Authenticity by tagging and typing. In *Formal Methods in Security Engineering (FMSE'04)*, pages 1–12, 2004.

[18] T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *16th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2003.

[19] J. Crampton, G. Loizou, and G. O'Shea. A logic of access control. *The Computer Journal*, 44(2):137–149, 2001.

[20] J. DeTreville. Binder, a logic-based security language. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 105–113, 2002.

[21] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

[22] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory, RFC 2693, 1999. See `http://www.ietf.org/rfc/rfc2693.txt`.

[23] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 141–156. Springer, 2005.

[24] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies in distribued systems. Technical Report MSR–TR–2007–47, Microsoft Research, 2007.

[25] D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 283–296, 2006.

[26] D. Gifford and J. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.

[27] A. D. Gordon and A. Jeffrey. Cryptyc: Cryptographic protocol type checker. At *http://cryptyc.cs.depaul.edu/*, 2002.

[28] A. D. Gordon and A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Software Security—Theories and Systems*, volume 2609 of *Lecture Notes in Computer Science*, pages 270–282. Springer, 2002.

[29] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.

[30] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3/4):435–484, 2003.

[31] A. D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *CONCUR 2005—Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 186–201. Springer, 2005.

[32] J. D. Guttman, F. J. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Trust management in strand spaces: a rely-guarantee method. In *13th European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 340–354. Springer, 2004.

[33] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

[34] N. Li, B. Grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *IEEE Symposium on Security and Privacy*, pages 27–42, 2000.

[35] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

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

[37] M. Maffei. *Dynamic typing for security protocols*. PhD thesis, Università Ca' Foscari Venezia, 2006.

[38] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[39] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

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

[41] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[42] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure, 1996. See `http://theory.lcs.mit.edu/~rivest/sdsi10.ps`.

[43] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[44] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2005.

[45] P. Samarati and S. de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD 2000*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer, 2001.

[46] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. CUP, 2001.

[47] T. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[48] T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.

[49] S. Zdancevic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *Transactions on Computer Systems*, 20(3):283–328, 2002.