A Type Discipline for Authorization Policies

CÉDRIC FOURNET and ANDREW D. GORDON
Microsoft Research
and
SERGIO MAFFEIS
Imperial College London

Distributed systems and applications are often expected to enforce high-level authorization policies. To this end, the code for these systems relies on lower-level security mechanisms such as, for instance, digital signatures, local ACLs, and encrypted communications. In principle, authorization specifications can be separated from code and carefully audited. Logic programs, in particular, can express policies in a simple, abstract manner.

We consider the problem of checking whether a distributed implementation based on communication channels and cryptography complies with a logical authorization policy. We formalize authorization policies and their connection to code by embedding logical predicates and claims within a process calculus. We formulate policy compliance operationally by composing a process model of the distributed system with an arbitrary opponent process. Moreover, we propose a dependent type system for verifying policy compliance of implementation code. Using Datalog as an authorization logic, we show how to type several examples using policies and present a general schema for compiling policies.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs; K.6.5 [**Management of Computer and Information Systems**]: Security and Protection

General Terms: Languages, Security, Theory, Verification

Additional Key Words and Phrases: Authorization, type systems, process calculus, spi calculus

1. TYPING IMPLEMENTATIONS OF AUTHORIZATION POLICIES

1.1 Background

Given a request to access a sensitive resource in a computer system, an *authorization policy* determines whether the request is allowed. The conditions in authorization policies typically involve the action (for example, writing a file), objects (the file being accessed, its directory), and subjects (the requester, the owner of the file). A system complies with the policy if these conditions hold whenever the action is performed. Authorization and access control issues can be complex, even at an abstract level. Some policies address security concerns for multiple actors and may involve numerous concepts such as roles, groups, partial trust, and controlled delegation. Their study has a long history [Lampson 1971; Samarati and de Capitani di Vimercati 2001].

Often, authorization policies are expressed precisely only in code, intermingled with other functions and with low-level enforcement mechanisms such as cryptography or system calls. The result can be hard to analyze and audit. Hence, a reasonable guiding princi-

Fournet and Gordon's address: Microsoft Research, 7 J J Thomson Avenue, Cambridge CB3 0FB, United Kingdom. Maffeis' address: Department of Computing, 180 Queen's Gate, Imperial College London, SW7 2AZ, United Kingdom.

ple is to express authorization policies in a high-level language, separate from imperative code and independent of particular enforcement mechanisms. Specifically, logic programming seems well suited for expressing policies precisely and concisely: each authorization request is formulated as a logical request against a database of facts and rules. Often, the policy itself carefully controls changes to the database. In particular, variants of Datalog have been usefully applied to design trust management systems (for instance, PolicyMaker [Blaze et al. 1996], SD3 [Jim 2001], Binder [DeTreville 2002]), to express complex policies (for instance, Cassandra [Becker and Sewell 2004]), and to study authorization languages (for instance, SDSI/SPKI [Abadi 1998; Li and Mitchell 2003], XrML [ContentGuard 2002]).

1.2 Our Approach

Given a target authorization policy, we consider the problem of verifying whether a particular system correctly implements the policy. In a distributed setting, this refinement typically involves security protocols and cryptography. For instance, when receiving a request, one may first verify an identity certificate, then authenticate the message, and finally consider the privileges associated with the sender.

Since the whole system can be seen as a complex cryptographic protocol, we adopt two ideas from work on specifying security protocols:

—First, annotations on the code of a system mark security-related events such as access rights being granted and checked. In previous work, the relation between imperative code and declarative policies is usually informal: theoretical studies rarely connect an authorization logic to an operational semantics. Our work makes the connection explicit; we aim to show that every successful access control decision made by code actually conforms to the authorization policy.

—Second, we adapt the standard “network is the opponent” threat model, a conservative model first formalized by Dolev and Yao [1983]. Hence, we aim to show that active attacks on the underlying cryptographic protocols cannot bypass our authorization policy; in particular, we want to prove the absence of the man-in-the-middle or impersonation attacks that often afflict cryptographic protocols.

Our formal development is within a typed version of the spi calculus [Abadi and Gordon 1999], a pi calculus with abstract cryptographic operations. We use inert processes—called statements and expectations—as code annotations to state the global authorization policy, to mark successful authorization checks, and to mark the pre-conditions for access to sensitive resources.

—A *statement* records an arbitrary logical clause. For example, a statement

$$\text{employee(alice)}$$

records that alice belongs to the group of employees. Such an annotation would follow code checking for alice in a suitable database, for example. A statement of a logical clause

$$\text{canRead(X,handbook)} :- \text{employee(X)}$$

records that any employee can read a particular file handbook. Such a statement might be a top-level annotation on the whole system, stating a global policy.

—An *expectation* is a falsifiable claim that a particular fact or clause is logically entailed by the set of active statements. For example, the expectation

expect canRead(alice,handbook)

records that canRead(alice,handbook) should be entailed in the current context. Such an annotation would precede the code providing alice access to the sensitive resource handbook, for example. This expectation is justified if the two previous statements are active. On the other hand, if those are the only active statements, the expectation

expect canRead(bob,handbook)

is unjustified. The presence of this expectation at runtime may reveal a coding error that allows bob access to handbook without a preceding check for bob in the employee database.

Our methodology is to insert statements after code performing dynamic checks, and to insert expectations before code accessing sensitive resources, so that access control errors result in unjustified expectations. The role of our type system is to check statically that in all executions, all expectations are justified by previously executed statements.

Statements and expectations generalize the begin- and end-events of a previous embedding [Gordon and Jeffrey 2003b] of the correspondences of Woo and Lam [1993] in a process calculus. Correspondences are a common basis for specifying correctness of authentication protocols. (Authentication should not be confused with authorization, although the former is often a prerequisite for the latter; authorization answers questions such as “is this request allowed?” while authentication answers subsidiary questions such as “who sent this request?”)

In contrast to several previous works, we use the authorization language as a statically enforced specification, instead of a language for programming dynamic authorization decisions. The two approaches are complementary, and indeed may be combined. The static approach is less flexible in terms of policies, as we need to anticipate the usage of the facts and rules involved at runtime. In contrast, a logic-based implementation may dynamically accept (authenticated) facts and rules, as long as they lead to a successful policy evaluation. The static approach is more flexible in terms of implementations, as we can assemble imperative and cryptographic mechanisms (for example, communications to collect remote certificates), irrespective of the logic-based evaluation strategy suggested by the policy. Hence, the static approach may be more efficient and pragmatically simpler to adapt to existing systems. Non-executable policies may also be simpler to write and to maintain, as they can safely ignore functional issues.

1.3 Summary of Contributions

To our knowledge, our work is the first to relate authorization logics to their cryptographic implementation in a process calculus.

—We show how to embed a range of authorization logics within a pi calculus. (We use Datalog as a simple, concrete example of an authorization logic.)

—We develop a new type system that checks conformance to a logic policy by keeping track of logical facts and rules in the typing environment, and using logical deduction to type authorization expectations. Our main result, Theorem 3, states that all expectations activated in a well-typed program follow from the enclosing policy.

—As a sample application, we present two distributed implementations of a simple Datalog policy for conference management that features rules for filing reports and delegating reviews. One implementation requests each delegation to be registered online, whereas the other enables offline, signature-based delegation, and checks the whole delegation chain later, when a report is filed.

—As another application, we present a generic implementation of Datalog via a translation into the pi calculus. The translated processes are well-typed in our system. They can serve as a default centralized implementation for any part of a policy.

We built a typechecker and a symbolic interpreter for our language, and used them to validate these applications. Our initial experience confirms the utility of such tools for writing code that composes several protocols, even if its overall size remains modest so far (a few hundred lines).

1.4 Related Work

There is a substantial literature on type systems for checking security properties. To the best of our knowledge, the earliest work on types for access control is by Jones and Liskov [1978]. They propose extending strongly typed languages with constraint expressions to specify an upper bound on legal access to objects of a given type. If well-typed, a program never tries to access an object whose type does not have the correct access permission.

In the context of process calculi there are, for example, type systems to check various information flow [Abadi 1999; Gordon and Jeffrey 2005; Pottier 2002] and authenticity [Duggan 2002; Gordon and Jeffrey 2003a] properties in the pi calculus and the spi calculus, access control properties of mobile code in the boxed ambient calculus [Bugliesi, Castagna, and Crafa 2004], and discretionary access control [Bugliesi, Colazzo, and Crafa 2004], and role-based access control [Braghin et al. 2004] in the pi calculus.

Various experimental systems, such as JIF [Myers and Liskov 2000] and KLAIM [De Nicola et al. 2000], for example, include types for access control. Still, there appears to be no prior work on typing implementations of a general authorization logic.

In the context of strand spaces and nonce-based protocols, Guttman et al. [2004] annotate send actions in a protocol with trust logic formulas which must hold when a message is sent, and receive actions with formulas which can be assumed to hold when a message is received. Their approach also relies on logically-defined correspondence properties, but it assumes the dynamic invocation of an external authorization engine, thereby cleanly removing the dependency on a particular authorization policy when reasoning about protocols. A more technical difference between our approaches is that we attach static authorization effects to any operation (input, decryption, matching) rather than just to message inputs.

ProVerif [Blanchet 2002] checks correspondence assertions in the applied pi calculus by reduction to a logic programming problem. ProVerif can check complex disjunctive correspondences, but has not been applied to check general clausally-defined authorization policies.

Guelev et al. [2004] also adopt a conference programme committee as a running example, in the context of model checking the consequences of access control policies.

1.5 Contents

The paper is organized as follows. Section 2 reviews Datalog, illustrates its usage to express authorization policies, and states a general definition of authorization logics. Section 3 defines a spi calculus with embedded authorization assertions. Section 4 presents our type system and states our main safety results. Section 5 develops well-typed distributed implementations for our sample delegation policy. Section 6 describes our pi calculus implementation of Datalog and states its correctness and completeness. Section 7 concludes and sketches future work.

Appendixes contain the proofs of the theorems stated in the body of the paper. Appendix A contains the proofs for Datalog, and a generic substitutivity property of authorization logics useful for our main results. Appendix B contains the proofs of our robust safety result for the spi calculus. Appendix C contains the formal definition of syntactic sugar and the proofs for the encoding of Datalog in spi.

Some details of proofs omitted from this paper plus the full listing of the example in Section 5 appear in a technical report [Fournet et al. 2005a]. A preliminary, abridged version of this work appears as a conference paper [Fournet et al. 2005b].

2. A SIMPLE LOGIC FOR AUTHORIZATION

We briefly present a syntax and semantics for Datalog, and discuss its use in formulating authorization policies. (For a comprehensive survey of Datalog, see Ceri et al. [1989].) The results in subsequent sections are independent of many of the details of Datalog; we formulate a notion of *authorization logic* to capture the properties on which we rely.

2.1 Syntax of Datalog

A Datalog program consists of *facts*, which are statements about the universe of discourse, and *clauses*, which are rules that can be used to infer facts. In the following, we interpret Datalog programs as authorization policies.

SYNTAX FOR DATALOG:

X, Y, Z	logic variable
$u ::=$	term
$\quad X$	$\quad$ logic variable
$\quad M$	$\quad$ spi calculus message (see Section 3)
$L ::=$	literal
$\quad p(u_1, \ldots, u_n)$	$\quad$ predicate p holds for terms $u_1, \ldots, u_n$
$C ::=$	Horn clause
$\quad L :- L_1, \ldots, L_n$	$\quad$ clause, with $n \geq 0$ and $fv(L) \subseteq \bigcup_i fv(L_i)$
$S ::=$	Datalog program (or policy)
$\quad \{C_1, \ldots, C_n\}$	$\quad$ set of clauses

Convention: a clause $L :-$ with an empty body (a *fact*) is denoted simply by L.
We let F range over facts.

Terms range over logic variables X, Y, Z and messages M; these messages are treated as Datalog atoms, but they have some structure in our spi calculus, defined in Section 3.

A clause $L :- L_1, \ldots, L_n$ has a *head*, L, and a *body*, $L_1, \ldots, L_n$; it is intuitively read as the universal closure of the propositional formula $L_1 \wedge \ldots \wedge L_n \rightarrow L$. In a clause, logic variables occurring in the body bind those occurring in the head. A phrase of syntax is *ground* if it has no free variables. We require that each clause be ground. A *fact* F is a clause with an empty body.

We use the following notations: for any phrase φ, we let $fn(\varphi)$ and $fv(\varphi)$ collect free spi calculus names and free variables, respectively. We write $\widetilde{\varphi}$ for the tuple $\varphi_1, \ldots, \varphi_t$, for some $t \geq 0$. We write $\{u/X\}$ for the capture-avoiding substitution of term u for variable X, and write $\{\widetilde{u}/\widetilde{X}\}$ instead of $\{u_1/X_1\} \ldots \{u_n/X_n\}$. We let σ range over these substitutions. Similarly, we write $\{M/x\}$ for the capture-avoiding substitution of message M for name x. We use postfix notations for applying substitutions.

2.2 Semantics of Datalog

We describe standard semantics for deriving facts and clauses from a Datalog program.

Facts can be derived using the rule below:

LOGICAL INFERENCE OF FACTS: $S \models F$

(Infer Fact)

$$\frac{L :- L_1, \ldots, L_n \in S \qquad S \models L_i\sigma \quad \forall i \in 1..n}{S \models L\sigma} \quad \text{for } n \geq 0$$

More generally, a clause C is *entailed* by a program S, written $S \models C$, when we have $\{F \mid S' \cup \{C\} \models F\} \subseteq \{F \mid S' \cup S \models F\}$ for all programs S'. Similarly, C is *uniformly contained in* S when the inclusion above holds for all programs S' containing only facts. Entailment is a contextual property for programs: if $S \models C$ and $S \subseteq S'$, then $S' \models C$. We rely on this property when we reason about partial programs. In Datalog, entailment and uniform containment coincide, hence entailment is decidable [Sagiv 1987] and can be checked operationally using the *chase* technique.

THEOREM 1. *[Sagiv 1987] For all C and sets of clauses S, (1) and (2) are equivalent:*

(1) *for all sets of facts S', $\{F \mid S' \cup \{C\} \models F\} \subseteq \{F \mid S' \cup S \models F\}$;*

(2) *$S \cup \{L_1\sigma, \ldots, L_n\sigma\} \models L\sigma$, where $C = L :- L_1, \ldots, L_n$ and $\sigma = \{\widetilde{x}/\widetilde{X}\}$ is an injective substitution such that $\{\widetilde{x}\} \cap (fn(S) \cup fn(C)) = \varnothing$ and $\widetilde{X} = fv(L_1, \ldots, L_n)$.*

In light of the previous theorem, we generalize inference to clauses, as follows.

LOGICAL INFERENCE FOR CLAUSES (ENTAILMENT): $S \models C$

(Infer Clause)

$$\frac{S \cup \{L_1\sigma, \ldots, L_n\sigma\} \models L\sigma \qquad \sigma \text{ maps } fv(L_1, \ldots, L_n) \text{ to fresh, distinct atoms}}{S \models L :- L_1, \ldots, L_n}$$

We rely on the following monotonicity and substitutivity properties of Datalog inference when developing our type system.

PROPOSITION 1. *If $S \models C$ then $S \cup \{C'\} \models C$.*

PROPOSITION 2. *If $S \models C$ and σ sends names to messages, $S\sigma \models C\sigma$.*

2.3 Some Predicates for Authorization

Our main example application is a simplified conference management system, in charge of assigning papers to referees and collecting their reports. For simplicity, we focus on the fragment of the policy that controls the right to file a paper report in the system, from the conference manager's viewpoint. This right, represented by the predicate Report(U,ID,R), is parameterized by the principal who files the report, a paper identifier, and the report content. It means that principal U can submit report R on paper ID. For instance, the fact Report(alice,42,report42) authorizes a particular report to be filed. Ideally, such facts should be deducible from the policy, rather than added to the policy one at a time. To this end, we introduce a few other predicates.

Some predicates represent the content of some *extensional* database of explicitly given facts. In our example, for instance, PCMember(U) means that principal U is a member of the programme committee for the conference; Referee(U,ID) means that principal U has been asked to review ID; and Opinion(U,ID,R) means that principal U has written report R on paper ID. Other predicates are *intensional*; they represent views computed from this authorization database. For instance, one may decide to specify Report(U,ID,R) using two clauses:

$$\begin{array}{ll} \text{Report(U,ID,R)}{:-}\text{Referee(U,ID),Opinion(U,ID,R)} & \text{(clause A)} \\ \text{Report(U,ID,R)}{:-}\text{PCMember(U),Opinion(U,ID,R)} & \text{(clause B)} \end{array}$$

These clauses state that U can report R on ID if she has this opinion and, moreover, either U has been assigned this paper (clause A), or U is in the programme committee (clause B)—thereby enabling PC members to file reports on any paper even if it has not been assigned to them. Variants of this policy are easily expressible; for instance, we may instead state that PC members can file only subsequent reports, not initial ones, by using a recursive variant of clause B:

$$\text{Report(U,ID,R)}{:-}\text{PCMember(U),Opinion(U,ID,R),Report(V,ID,S)}$$

Continuing with our example, we extend the policy to enable any designated referees to delegate their task to a subreferee. To this end, we add an extensional predicate, Delegate(U,V,ID), meaning that principal U intends to delegate paper ID to principal V, and we add a clause to derive new facts Referee(V,ID) accordingly:

$$\text{Referee(V,ID)} :- \text{Referee(U,ID),Delegate(U,V,ID)} \qquad \text{(clause C)}$$

Conversely, the policy { A,B,C } does not enable a PC member to delegate a paper, unless the paper has been assigned to her.

As can be seen from these clauses, our logical formalization adopts the subjective viewpoint of the conference system, which implicitly owns all predicates used to control reports. In contrast, more sophisticated authorization languages [Abadi et al. 1993] associate facts with the principals that "say" them. Even if Opinion(U,_) and Delegate(U,...) are implicitly owned by U, these predicates represent the fact that the conference system believes these facts, rather than U's intents. Also, the distinction between intensional and extensional predicates is useful to interpret policies but is not essential. As we illustrate in Section 5, this distinction in the specification does not prescribe any implementation strategy.

2.4 A General Notion of Authorization Logic

Although Datalog suffices as an authorization logic for the examples and applications developed in this paper, its syntax and semantics are largely irrelevant to our technical developments. More abstractly, our main results hold for any logic that meets the following requirements.

AUTHORIZATION LOGIC: $(\mathscr{C}, fn, \models)$

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

By Propositions 1 and 2, Datalog is an authorization logic.

3. A SPI CALCULUS WITH AUTHORIZATION ASSERTIONS

The spi calculus [Abadi and Gordon 1999] extends the pi calculus with abstract cryptographic operations in the style of Dolev and Yao [1983]. Names represent both cryptographic keys and communication channels. The version of spi given here has a small but expressive range of primitives: encryption and decryption using shared keys, input and output on shared channel names, and operations on pairs. We conjecture that our results, including our type system, would smoothly extend to deal with more complex features such as asymmetric cryptography and communications, and a richer set of data types.

The main new features of our calculus are authorization assertions represented by inert processes called statements and expectations. These processes generalize the begin- and end-assertions in previous embeddings of correspondences in process calculi [Gordon and Jeffrey 2003b]. Similarly, statements and expectations track security properties but, (in contrast to assertions in typical programming languages) do not in themselves affect the behaviour of processes.

A *statement* is simply a clause C (either a fact or a rule). For example, the following process is a composition of clause A of Section 2.3 with two facts:

A | Referee(alice,42) | Opinion(alice,42,report42) (process P)

An *expectation* **expect** C represents the expectation on the part of the programmer that the rule or fact C can be inferred from clauses in parallel. Expectations typically record authorization conditions. For example, the following process represents the (justified) expectation that a certain fact follows from the clauses of P.

P | **expect** Report(alice,42,report42) (process Q)

Expectations most usefully concern messages instantiated at runtime. In the following, the content x of the report is received from the channel c:

P | **out** c (report42,**ok**) | **in** c(x,y); **expect** Report(alice,42,x) (process R)

In this parallel composition, the second subprocess outputs a message whose payload is a pair that contains the report plus the distinguished token **ok** (an annotation to help typing, with no effect at runtime). The third subprocess inputs a message, binds its content to x and y, and expects x to be the report.

All the statements arising in our case studies fall into two distinct classes. One class consists of unguarded, top-level statements of authorization rules, such as those in the previous section, that define the global authorization policy. The other class consists of input-guarded statements, triggered at runtime, that declare facts—not rules—about data arising at runtime, such as the identities of particular reviewers or the contents of reports. Moreover, we have not found a use for expectations of proper rules; all the expectations in our case studies are simply of facts.

The syntax and informal semantics of our full calculus is as follows. Binding occurrences of names have type annotations, T or U; the syntax of our system of dependent types is in Section 4.

SYNTAX FOR MESSAGES AND PROCESSES:

a,b,c,k,x,y,z	name
$M,N ::=$	message
x	name: a key or a channel
$\{M\}N$	authenticated encryption of M with key N
(M,N)	message pair
ok	distinguished message
$P,Q,R ::=$	process
out $M(N)$	asynchronous output of N to channel M
in $M(x{:}T);P$	input of x from channel M (x has scope P)
new $x{:}T;P$	fresh generation of name x (x has scope P)
$P \mid Q$	parallel composition of P and Q
$!P$	unbounded parallel composition of replicas of P
0	inactivity
decrypt M **as** $\{y{:}T\}N;P$	bind y to decryption of M with key N (y has scope P)
split M **as** $(x{:}T,y{:}U);P$	solve $(x,y) = M$ (x has scope U and P; y has scope P)
match M **as** $(N,y{:}U);P$	solve $(N,y) = M$ (y has scope P)
C	statement of clause C
expect C	expectation that clause C is derivable

Notations: $(\widetilde{x}{:}\widetilde{T}) \triangleq (x_1{:}T_1,\ldots,x_n{:}T_n)$ and **new** $\widetilde{x}{:}\widetilde{T};P \triangleq$ **new** $x_1{:}T_1;\ldots$ **new** $x_n{:}T_n;P$
Let $S = \{C_1,\ldots,C_n\}$. We write $S \mid P$ for $C_1 \mid \ldots \mid C_n \mid P$.

The **split** and **match** processes for destructing pairs are worth comparing. A **split** binds names to the two parts of a pair, while a **match** is effectively a **split** followed by a conditional; think of **match** M **as** $(N,y);P$ as **split** M **as** $(x,y);$**if** $x = N$ **then** P. Taking **match** as primitive is a device to avoid using unification in a dependent type system [Gordon and Jeffrey 2003a].

Next, we present the operational semantics of our calculus via standard structural equivalence ($P \equiv Q$) and reduction ($P \rightarrow Q$) relations. The following rules are standard. Statements and expectations are inert processes; they do not have particular rules for reduction or equivalence (although they are affected by other rules). The conditional operations **decrypt**, **split**, and **match** simply get stuck if decryption or matching fails; we could allow alternative branches for error handling, but they are not needed for the examples in the paper.

RULES FOR STRUCTURAL EQUIVALENCE: $P \equiv Q$

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv P' \Rightarrow \textbf{new}\ x{:}T; P \equiv \textbf{new}\ x{:}T; P'$	(Struct Res)
$P \equiv P' \Rightarrow P \mid R \equiv P' \mid R$	(Struct Par)
$P \equiv P' \Rightarrow !P \equiv !P'$	(Struct Repl)
$P \mid \mathbf{0} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Unfold)
$!!P \equiv !P$	(Struct Repl Repl)
$!(P \mid Q) \equiv !P \mid !Q$	(Struct Repl Par)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct Repl Zero)
$\textbf{new}\ x{:}T; (P \mid Q) \equiv P \mid \textbf{new}\ x{:}T; Q$	(Struct Res Par) (for $x \notin fn(P)$)
$\textbf{new}\ x_1{:}T_1; \textbf{new}\ x_2{:}T_2; P \equiv$	(Struct Res Res)
$\quad \textbf{new}\ x_2{:}T_2; \textbf{new}\ x_1{:}T_1; P$	(for $x_1 \neq x_2, x_1 \notin fn(T_2), x_2 \notin fn(T_1)$)

RULES FOR REDUCTION: $P \rightarrow P'$

$P \rightarrow P' \Rightarrow P \mid Q \rightarrow P' \mid Q$	(Red Par)
$P \rightarrow P' \Rightarrow \textbf{new}\ x{:}T; P \rightarrow \textbf{new}\ x{:}T; P'$	(Red Res)
$P \equiv Q, Q \rightarrow Q', Q' \equiv P' \Rightarrow P \rightarrow P'$	(Red Struct)
$\textbf{out}\ a(M) \mid \textbf{in}\ a(x{:}T); P \rightarrow P\{M/x\}$	(Red Comm)
$\textbf{decrypt}\ \{M\}k\ \textbf{as}\ \{y{:}T\}k; P \rightarrow P\{M/y\}$	(Red Decrypt)
$\textbf{split}\ (M,N)\ \textbf{as}\ (x{:}T, y{:}U); P \rightarrow P\{M/x\}\{N/y\}$	(Red Split)
$\textbf{match}\ (M,N)\ \textbf{as}\ (M, y{:}U); P \rightarrow P\{N/y\}$	(Red Match)

Notation: $P \rightarrow^*_{\equiv} P'$ is $P \equiv P'$ or $P \rightarrow^* P'$.

In examples, we rely on derived notations for n-ary tuples and pattern-matching via sequences of match and split operations. For $n > 2$, $(M_1, M_2, \ldots, M_n)$ abbreviates $(M_1, (M_2, \ldots, M_n))$. For pattern matching, we write **tuple** M **as** $(\underline{N}_1, \ldots, \underline{N}_n); P$, where $n > 0$, M is a message (expected to be a tuple), and each $\underline{N}_i$ is an atomic pattern. Let an atomic pattern be either a variable pattern x, or a constant pattern, written $=M$, where M is a message to be matched. Each variable pattern translates to a **split**, and each constant pattern translates to a **match**. For example, **tuple** (a,b,c) **as** $(x, =b, y); P$ translates to the process **split** $(a,(b,c))$ **as** (x,z); **match** z **as** (b,z); **split** (z,z) **as** $(y,z); P$, where z is fresh. The translation introduces a fresh temporary name z not occurring free in P, and at the last step it duplicates z in order to allow a match or split operation. When using the **tuple** notation, we omit the types from variable patterns because they can be inferred during typechecking. Appendix C includes the formal definition of this tuple notation.

We enrich the syntax of inputs and decryption with the tuple notation as follows, where in both translations the name y is chosen to not occur in $\underline{N}_1,\ldots,\underline{N}_n,P$.

$$\begin{aligned}\textbf{in } M(\underline{N}_1,\ldots,\underline{N}_n);P &\stackrel{\triangle}{=} \textbf{in } M(y);\textbf{tuple } y \textbf{ as } (\underline{N}_1,\ldots,\underline{N}_n);P\\ \textbf{decrypt } M \textbf{ as } \{\underline{N}_1,\ldots,\underline{N}_n\}N;P &\stackrel{\triangle}{=} \textbf{decrypt } M \textbf{ as } \{y\}N;\textbf{tuple } y \textbf{ as } (\underline{N}_1,\ldots,\underline{N}_n);P\end{aligned}$$

The notation does not translate to an atomic primitive; hence, in the case of input, a message may be received, then silently discarded because it does not match the pattern. This does not matter in our case because we are mostly interested in safety properties.

The presence of statements and expectations in a process induces the following safety properties. Intuitively, an expectation **expect** C is *justified* when there are sufficient active statements to derive C. (A statement is active when it appears in the context, in parallel composition with **expect** C.) Then a process is safe if every expectation in every reachable process is justified.

SAFETY:

A process P is *safe* if and only if whenever

$$P \rightarrow_{\equiv}^{*} \textbf{new } \tilde{x}{:}\tilde{T};(\textbf{expect } C \mid P')$$

we have $P' \equiv \textbf{new } \tilde{y}{:}\tilde{U};(C_1 \mid \ldots \mid C_n \mid P'')$ and $\{C_1,\ldots,C_n\} \models C$ with $\{\tilde{y}\} \cap fn(C) = \varnothing$.

The definition mentions $\tilde{x}$ to allow fresh names in C, while it mentions $\tilde{y}$ to ensure that the clauses $C, C_1, \ldots, C_n$ all use the same names; the scopes of these names are otherwise irrelevant in the logic. Were the definition to omit the outer restricted names $\tilde{x}$, the process

$$\textbf{new } \mathrm{x};\ \textbf{expect } \mathrm{Foo(x)}$$

would be judged safe (because this process does not match the pattern for P in the definition). Conversely, were the definition to omit the intermediate restricted names $\tilde{y}$, the process

$$\textbf{expect } \mathrm{Bar()} \mid \mathrm{Bar()}{:}{-}\mathrm{Foo(X)} \mid \textbf{new } \mathrm{y};\ \mathrm{Foo(y)}$$

would be judged unsafe (because this process matches the pattern for P in the definition whereas its subprocess Bar():$-$Foo(X) $\mid$ **new** y; Foo(y) does not match the pattern for P').

Given a process P representing the legitimate participants making up a system, we want to show that no opponent process O can induce P into an unsafe state, where some expectation is unjustified. An opponent is any process within our spi calculus, except it is not allowed to include any expectations itself. (The opponent goal is to confuse the legitimate participants about who is doing what.) As a technical convenience, we require every type annotation in an opponent to be a certain type **Un**; type annotations do not affect the operational semantics, so the use of **Un** does not limit opponent behaviour.

OPPONENTS AND ROBUST SAFETY:

A process O is an *opponent* if and only if it contains no expectations, and every type annotation is **Un**.
A process P is *robustly safe* if and only if $P \mid O$ is safe for all opponents O.

As a consequence of this definition, in every run of a robustly safe process P in parallel with some opponent, every expectation can be justified by statements activated in P.

For example, the process Q given earlier is robustly safe, because the statements in P suffice to infer Report(alice,42,report42), and they persist in any interaction with an opponent. On the other hand, the process R is safe on its own, but is not robustly safe. Consider the opponent **out** c (bogus,**ok**). We have:

$$\mathrm{R} \mid \textbf{out}\ \mathrm{c}\ (\mathrm{bogus},\textbf{ok}) \rightarrow \mathrm{P} \mid \textbf{out}\ \mathrm{c}\ (\mathrm{report42},\textbf{ok}) \mid \textbf{expect}\ \mathrm{Report(alice,42,bogus)}$$

This is unsafe because Report(alice,42,bogus) is not derivable from the statements in process P. We can secure the channel c by using the **new** operator to make it private. The process **new** c; R is robustly safe; no opponent can inject a message on c.

4. A TYPE SYSTEM FOR VERIFYING AUTHORIZATION ASSERTIONS

We present a dependent type system for statically checking implementations of authorization policies. Although we develop a type system, other styles of static analysis are likely applicable to the problem of proving robust safety. Moreover, instead of or in combination with a static analysis, an implementation may record the statements issued by a running program within some distributed database so as to check expectations dynamically; our implementation of Datalog in Section 6 is a step in this direction.

Our starting point is a type and effect system [Gordon and Jeffrey 2002b] for verifying one-to-many correspondences, which itself builds on prior work on types for channel-based communication in the pi calculus [Milner 1999; Pierce and Sangiorgi 1996] and types for cryptographic primitives in the spi calculus [Abadi 1999]. Apart from the new support for logical assertions, the current system features two improvements. First, a new rule for parallel composition allows us to typecheck a safe process such as $L \mid \textbf{expect}\ L$; the analogous parallel composition cannot be typed in the original system. Second, effects are merged into typing environments, leading to a much cleaner presentation, and to the elimination of typing rules for effect subsumption.

4.1 Syntax of Types and Environments

We begin by defining the syntax and informal semantics of message types.

SYNTAX FOR TYPES:

$T,U ::=$	type
$\textbf{Un}$	public data
$\textbf{Ch}(T)$	channel for messages of type T
$\textbf{Key}(T)$	secret key for plaintexts of type T
$(x{:}T,U)$	dependent pair (scope of x is U)
$\textbf{Ok}(S)$	ok to assume the clauses S

T is *generative* (may be freshly created) if and only if T is $\textbf{Un}$, $\textbf{Ch}(U)$, or $\textbf{Key}(U)$.

Notation: $(x_1{:}T_1,\ldots,x_n{:}T_n,T_{n+1}) \stackrel{\triangle}{=} (x_1{:}T_1,\ldots,(x_n{:}T_n,T_{n+1}))$

A message of type **Un** is public data that may flow to or from the opponent; for example, all ciphertexts are of type **Un**. A message of type $\textbf{Ch}(T)$ is a name used as a secure channel for messages of type T. Similarly, a message of type $\textbf{Key}(T)$ is a name used as a secret key for encrypting and decrypting plaintexts of type T. A message of the dependent type $(x{:}T,U)$ is a pair (M,N) where M is of type T, and N is of type $U\{M/x\}$. The type $(x{:}T,U)$ is a generalization of an ordinary product type. (Such dependent types are standard in intuitionistic type theory [Martin-Löf 1984], where they are known as Σ-types.) Finally, the token **ok** is the unique message of type $\mathbf{Ok}(S)$, proving S may currently be inferred.

For example, the type $\mathbf{Ch}((x{:}\mathbf{Un}, \mathbf{Ok}(\text{Report}(\text{alice}, 42, x))))$ can be assigned to c in process R, stating that c is a channel for communicating pairs $(M, \mathbf{ok})$ where $M : \mathbf{Un}$ and $\mathbf{ok} : \mathbf{Ok}(\text{Report}(\text{alice}, 42, M))$. This example illustrates a common idiom, in which the final component of a communicated tuple is an **ok** that conveys facts about the previous components. In this idiom, the tuple takes the form $(M_1, \ldots, (M_n, \mathbf{ok}) \ldots)$ and has the type $(x_1{:}T_1, \ldots, (x_n{:}T_n, \mathbf{Ok}(S)) \ldots)$. The names $x_1, \ldots, x_n$ can occur free in the clauses S, so that the **ok** has type $\mathbf{Ok}(S\{M_1/x_1\} \ldots \{M_n/x_n\})$, and hence conveys the facts $S\{M_1/x_1\}$ $\ldots \{M_n/x_n\}$, which may refer to the components $M_1, \ldots, M_n$.

The *generative types*, **Un**, $\mathbf{Ch}(U)$, or $\mathbf{Key}(U)$, are the types of freshly generated names. A restriction **new** $x{:}T; P$ generates the name x and is only well-typed if the type T is generative.

Next, we define typing environments—lists of name bindings and clauses—plus some auxiliary functions. The function $dom(-)$ sends an environment to the set of names to which it assigns a type. The function $env(-)$ sends a process to an environment that collects its top-level statements, with suitable name bindings for any top-level restrictions. The function $clauses(-)$ sends an environment to the program consisting of all the clauses listed in the environment plus the clauses in top-level $\mathbf{Ok}(-)$ types.

SYNTAX FOR ENVIRONMENTS, AND FUNCTIONS: $dom(E)$, $env(P)$, $clauses(E)$

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

Notation: $E(x) = T$ if $E = E', x{:}T, E''$
E is *generative* if and only if $E = x_1{:}T_1, \ldots, x_n{:}T_n$ and each T_i is generative.

$$dom(E,C) = dom(E) \quad dom(E,x{:}T) = dom(E) \cup \{x\} \quad dom(\varnothing) = \varnothing$$

$$env(P \mid Q)^{\widetilde{x},\widetilde{y}} = env(P)^{\widetilde{x}}, env(Q)^{\widetilde{y}} \quad (\text{where } \{\widetilde{x},\widetilde{y}\} \cap fn(P \mid Q) = \varnothing)$$
$$env(\mathbf{new}\ x{:}T; P)^{x,\widetilde{x}} = x{:}T, env(P)^{\widetilde{x}} \quad (\text{where } \{\widetilde{x}\} \cap fn(P) = \varnothing)$$
$$env(!P)^{\widetilde{x}} = env(P)^{\widetilde{x}} \qquad env(C)^{\varnothing} = C \qquad env(P)^{\varnothing} = \varnothing \text{ otherwise}$$

Convention: $env(P) \triangleq env(P)^{\widetilde{x}}$ for some distinct $\widetilde{x}$ such that $env(P)^{\widetilde{x}}$ is defined.

$$clauses(E,C) = clauses(E) \cup \{C\} \quad clauses(E,x{:}\mathbf{Ok}(S)) = clauses(E) \cup S$$
$$clauses(E,x{:}T) = clauses(E) \text{ if } T \neq \mathbf{Ok}(S) \quad clauses(\varnothing) = \varnothing$$

4.2 Judgments and Typing Rules

Our system consists of three judgments, defined by the following tables.

The judgment $E \vdash \diamond$ means that the environment E is well-formed.

RULES FOR ENVIRONMENTS: $E \vdash \diamond$

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env x)
$$\frac{E \vdash \diamond \quad fn(T) \subseteq dom(E) \quad x \notin dom(E)}{E, x{:}T \vdash \diamond}$$

(Env C)
$$\frac{E \vdash \diamond \quad fn(C) \subseteq dom(E)}{E, C \vdash \diamond}$$

The rules (Env $\varnothing$), (Env x), (Env C) ensure that each name occurring in a type or a clause in an environment is itself assigned a type by the environment, and that each name is assigned a type at most once.

The judgment $E \vdash M : T$ means that in environment E, the message M has type T.

RULES FOR MESSAGES: $E \vdash M : T$

(Msg x)
$$\frac{E \vdash \diamond \quad x \in \mathit{dom}(E)}{E \vdash x : E(x)}$$

(Msg Encrypt)
$$\frac{E \vdash M : T \quad E \vdash N : \mathbf{Key}(T)}{E \vdash \{M\}N : \mathbf{Un}}$$

(Msg Encrypt Un)
$$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un}}{E \vdash \{M\}N : \mathbf{Un}}$$

(Msg Pair)
$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M,N) : (x{:}T,U)}$$

(Msg Pair Un)
$$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un}}{E \vdash (M,N) : \mathbf{Un}}$$

(Msg Ok)
$$\frac{E \vdash \diamond \quad \mathit{fn}(S) \subseteq \mathit{dom}(E) \quad \mathit{clauses}(E) \models C \quad \forall C \in S}{E \vdash \mathbf{ok} : \mathbf{Ok}(S)}$$

(Msg Ok Un)
$$\frac{E \vdash \diamond}{E \vdash \mathbf{ok} : \mathbf{Un}}$$

The rule (Msg x) assigns a name the type given it by the environment. The rule (Msg Encrypt) assigns a ciphertext the type **Un**, provided that the encryption key has a type $\mathbf{Key}(T)$ and the plaintext has type T. The rule (Msg Pair) assigns a pair (M,N) the type $(x{:}T,U)$, provided that M has type T and N has the type $U\{M/x\}$ dependent on M. The rule (Msg Ok) populates an $\mathbf{Ok}(S)$ type only if we can infer each clause in the Datalog program S from the clauses in the environment E. For example, using clause A of Section 2.3, if

$$\begin{aligned} E = {} & \text{alice}{:}\mathbf{Un}, 42{:}\mathbf{Un}, \text{report42}{:}\mathbf{Un}, \\ & A, \text{Referee(alice,42)}, \text{Opinion(alice,42,report42)} \end{aligned}$$

then $E \vdash \mathbf{ok} : \mathbf{Ok}(\text{Report}(\text{alice}, 42, \text{report42}))$.

As in previous systems [Gordon and Jeffrey 2003a; 2002b], we need the rules (Msg Encrypt Un), (Msg Pair Un), and (Msg Ok Un) to assign **Un** to arbitrary messages known to the opponent.

The judgment $E \vdash P$ means that in environment E, the process P is well-typed.

RULES FOR PROCESSES: $E \vdash P$

(Proc Nil)
$$\frac{E \vdash \diamond}{E \vdash \mathbf{0}}$$

(Proc Rep)
$$\frac{E \vdash P}{E \vdash !P}$$

(Proc Res)
$$\frac{E, x{:}T \vdash P \quad T \text{ generative}}{E \vdash \mathbf{new}\ x{:}T; P}$$

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

(Proc Expect)
$$\frac{E, C \vdash \diamond \quad \mathit{clauses}(E) \models C}{E \vdash \mathbf{expect}\ C}$$

(Proc Fact)
$$\frac{E, C \vdash \diamond}{E \vdash C}$$

(Proc Decrypt)
$$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Key}(T) \quad E, y{:}T \vdash P}{E \vdash \mathbf{decrypt}\ M\ \mathbf{as}\ \{y{:}T\}N; P}$$

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

(Proc Decrypt Un)
$$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un} \quad E, y{:}\mathbf{Un} \vdash P}{E \vdash \mathbf{decrypt}\ M\ \mathbf{as}\ \{y{:}\mathbf{Un}\}N; P}$$

(Proc Input Un)
$$\frac{E \vdash M : \mathbf{Un} \qquad E, x{:}\mathbf{Un} \vdash P}{E \vdash \mathbf{in}\ M(x{:}\mathbf{Un}); P}$$

(Proc Match)
$$\frac{E \vdash M : (x{:}T, U) \quad E \vdash N : T \quad E, y{:}U\{N/x\} \vdash P}{E \vdash \mathbf{match}\ M\ \mathbf{as}\ (N, y{:}U\{N/x\}); P}$$

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

(Proc Match Un)
$$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un} \quad E, y{:}\mathbf{Un} \vdash P}{E \vdash \mathbf{match}\ M\ \mathbf{as}\ (N, y{:}\mathbf{Un}); P}$$

(Proc Output Un)
$$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un}}{E \vdash \mathbf{out}\ M(N)}$$

(Proc Split)
$$\frac{E \vdash M : (x{:}T, U) \quad E, x{:}T, y{:}U \vdash P}{E \vdash \mathbf{split}\ M\ \mathbf{as}\ (x{:}T, y{:}U); P}$$

(Proc Split Un)
$$\frac{E \vdash M : \mathbf{Un} \quad E, x{:}\mathbf{Un}, y{:}\mathbf{Un} \vdash P}{E \vdash \mathbf{split}\ M\ \mathbf{as}\ (x{:}\mathbf{Un}, y{:}\mathbf{Un}); P}$$

There are three rules of particular interest. (Proc Expect) allows **expect** C provided that C is entailed in the current environment. (Proc Fact) allows any statement, provided its names are in scope. (Proc Par) is the rule for parallel composition; it allows $P \mid Q$, provided that P and Q are well-typed given the top-level statements of Q and P, respectively. For example, by (Proc Par), $\varnothing \vdash$ Foo() $\mid$ **expect** Foo() follows from $\varnothing \vdash$ Foo() and Foo() $\vdash$ **expect** Foo(), the two of which follow directly by (Proc Fact) and (Proc Expect).

The rules (Proc Nil), (Proc Rep), (Proc Res), (Proc Output), and (Proc Input) type the core processes of the pi calculus, other than composition. These rules are much as in early systems for the pi calculus [Pierce and Sangiorgi 1996]. The rule (Proc Input) relies on the invariant that any message sent on a channel of type $\mathbf{Ch}(T)$ has type T, ensured by the rule (Proc Output). Similarly, the rule (Proc Decrypt) for decryption relies on the invariant that any plaintext encrypted with a key of type $\mathbf{Key}(T)$ has type T, ensured by the rule (Msg Encrypt). The rules (Proc Split) and (Proc Match) are for destructing pairs.

The rules (Proc Output Un), (Proc Input Un), (Proc Decrypt Un), (Proc Match Un), and (Proc Split Un) allow arbitrary opponent processes to be typed, assuming that all messages occurring in such processes can be assigned the **Un** type; these rules are needed to establish Lemma 2 below.

We have implemented a typechecker for this type system, with Datalog as its authorization logic. It consists of procedures to check the judgments $E \vdash P$ and $E \vdash M : T$, given their parameters as input. To apply the rules (Msg Ok) and (Proc Expect) we invoke a decision procedure for Datalog entailment. For some rules, type checking depends on a procedure that, given an environment E and a message M, infers a type T such that $E \vdash M : T$. With a more verbose syntax, in which each **ok** term is annotated with the S from its type $\mathbf{Ok}(S)$, the set of all types assignable to a term would be computable. Indeed, the three judgments of the type system would be decidable, although the time complexity would be exponential, due to the presence of two alternative typing rules for many message and process constructs. Instead, our typechecker follows the syntax of the paper, without annotations on **ok** terms, and our type inference procedure is incomplete, as it does not apply (Msg Ok), so as to avoid guessing the set S. Still, this incompleteness seldom arises; the type of an **ok** term is usually determined by context, as in the common case when the **ok** occurs as the component of a tuple whose type is determined by an encryption key or a communication channel. Hence, although our typechecker is incomplete, it can check a wide range of programs, including all the examples in this paper. Like other typecheckers for the spi calculus such as Cryptyc [Gordon and Jeffrey 2002a], our implementation allows type annotations to be omitted from inputs or decryptions, as they can be inferred from context, but requires type annotations on the names bound by restrictions. We do not consider the general problem of inferring type annotations for all bound names, although this is likely a prerequisite for the practical application of our system.

To the best of our knowledge, the type inference problem for systems of dependent types for the spi calculus has not been addressed in the literature. Maffei [2006] develops type systems for authentication properties in a spi calculus, using certain tags instead of dependent types, and proposes a type and tag inference algorithm. Maffei's is the closest work to the type inference problem for our system, but he does not consider dependent types nor indeed authorization properties.

We have not considered implementing other authorization logics than Datalog. Decidability of logical entailment is not essential, of course; typecheckers for undecidable or potentially intractable type systems are widely used in practice.

4.3 Main 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. *If $E \vdash P$ and either $P \equiv P'$ or $P \to P'$ then $E \vdash P'$.*

THEOREM 2. *If $E \vdash P$ and E is generative, then P is safe.*

Our second theorem is that well-typed processes whose free names are public, that is, of type **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 of type **Un**.

LEMMA 2. *If $fn(O) \subseteq \{\widetilde{x}\}$ for opponent O then $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash O$.*

THEOREM 3. *If $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash P$ then P is 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 an expectation of an unstated fact, even if the expectation is unreachable and the process is in fact robustly safe.

We conclude this section by showing that our calculus can encode standard one-to-many correspondence assertions. The idea of correspondences is that processes are annotated with two kinds of labelled events: begin-events and end-events. The intent is that in each run, for every end-event, there is a preceding begin-event with the same label.

For example, consider the (trivial) authorization logic $(\mathscr{C}, fn, \models)$, where $L \in \mathscr{C}$ are the labels used for the correspondence assertions, $\models$ is defined as $\{L\} \models L$ for each $L \in \mathscr{C}$, and fn is standard. In this setting, we can encode begin-events and end-events as follows.

$$\mathbf{begin}\ L; P \triangleq L \mid P \qquad\qquad \mathbf{end}\ L; P \triangleq \mathbf{expect}\ L \mid P$$

Given this trivial authorization logic our type system essentially degenerates to previous systems for authentication properties. For example, given this encoding and a minor extension to the type system (tagged union types), we can express and typecheck all of the authentication protocols from one previous study [Gordon and Jeffrey 2002b], including WMF and BAN Kerberos.

The correspondences expressible by standard begin- and end-events are a special case of the class of correspondences expressible in our calculus where the predicates in expectations are *extensional*, that is, given explicitly by facts. Hence, we refer to our generalized correspondence assertions based on intensional predicates as *intensional correspondences*, to differentiate them from standard (extensional) correspondences.

Finally, neither our operational semantics nor our type system handles one-to-one correspondences, where each begin-event corresponds to at most one end-event.

5. APPLICATION: PROGRAMME COMMITTEE ACCESS CONTROL

We provide two spi calculus implementations for the Datalog policy with delegation introduced in Section 2 (defining clauses A, B, and C). In both implementations, the server enables those three clauses as part of its policy, and also maintains a local database of registered reviewers on a private channel pwdb:

```
A | B | C | new pwdb : Ch( u:Un, Key(v:Un,id:Un,Ok(Delegate(u,v,id))),
                                  Key(id:Un,report:Un,Ok(Opinion(u,id,report))));
```

Hence, each message on pwdb codes an entry in the reviewer database, and associates the name u of a reviewer with two keys used to authenticate her two potential actions: delegating a review, and filing a report. The usage of these keys is detailed below.

Although we present our code in several fragments, these fragments should be read as parts of a single process, whose typability and safety properties are summarized at the end of the section. Hence, for instance, our policy and the local channel pwdb are defined for all processes displayed in this section.

5.1 Online Delegation, with Local State

Our first implementation assumes that the conference system is contacted whenever a referee decides to delegate her task. Hence, the system keeps track of expected reports using another local database, each record noting a fact of the form Referee(U,ID). When a report is received, the authenticated sender of the report is correlated with the principal that appears in the corresponding record. When a delegation request is received, the corresponding record is checked, then updated.

The following code defines the (abstract) behaviour of reviewer v; it is triggered whenever a message is sent on createReviewer; it has public channels providing controlled access to all her privileged actions—essentially any action authenticated with one of her two keys. For simplicity, we proceed without checking the legitimacy of requests, and we assume that v is not a PC member—otherwise, we would implement a third action for filing PC member reports.

```
(!in createReviewer(v);
  new kdv: Key(z:Un,id:Un,Ok(Delegate(v,z,id)));
  new krv: Key(id:Un,report:Un,Ok(Opinion(v,id,report)));
  ( (!out pwdb(v,kdv,krv))
  | (!in sendreportonline(=v,id,report);
        Opinion(v,id,report) | out filereport(v,{id,report,ok}krv) )
  | (!in delegateonline(=v,w,id);
        Delegate(v,w,id) | out filedelegate(v,{w,id,ok}kdv) ))) |
```

In the code triggered by createReviewer messages, we first generate two new keys kdv and krv. The replicated output on pwdb associates these keys with v. The replicated input on sendreportonline guards a process that files v's reports; in this process, the authenticated encryption {id,report,**ok**}krv protects the report and also carries a fact Opinion(v,id,report) stating its authenticity. The replicated input on delegateonline similarly guards a process that files v's delegations.

Next, we give the corresponding code that receives these two kinds of requests at the server. (We omit the code that selects reviewers and sends messages on refereedb.) In the code guarded by **!in** filereport(v,e), the decryption "proves" Opinion(v,id,report), whereas the input on refereedb "proves" Referee(v,id): when both operations succeed, these facts and clause A jointly guarantee that Report(v,id,report) is derivable. Conversely, our type system would catch errors such as forgetting to correlate the paper or the reviewer name (for instance, writing =v,id instead of =v,=id in refereedb), leaking the decryption key, or using the wrong key.

The process guarded by **!in** filedelegate(v,sigd) is similar, except that it uses the fact Delegate(v,w,id) granted by decrypting under key kdv to transform Referee(v,id) into Referee(w,id), which is expected for typing **ok** in the output on refereedb.

```
new refereedb : Ch(u:Un,(id:Un,Ok(Referee(u,id)))); (!in
filereport(v,e);
    in pwdb(=v,kdv,krv); decrypt e as {id,report,_}krv;
    in refereedb(=v,=id,_); expect Report(v,id,report)) |
(!in filedelegate(v,sigd);
    in pwdb(=v,kdv,krv); decrypt sigd as {w,id,_}kdv;
    in refereedb(=v,=id,_); out refereedb(w,id,ok)) |
```

The code for processing PC member reports is similar but simpler:

```
new kp:Key(u:Un,Ok(PCMember(u)));
(!in createPCMember(u,pc);PCMember(u) | out pc({(u,ok)}kp) ) |
(!in filepcreport(v,e,pctoken);
  in pwdb(=v,kdv,krv); decrypt e as {id,report,_}krv;
  decrypt pctoken as {=v,_}kp; expect Report(v,id,report) ) |
```

Instead of maintaining a database of PC members, we (arbitrarily) use capabilities, consisting of the name of the PC member encrypted under a new private key kp. The code implements two services as replicated inputs, to register a new PC member and to process a PC member report. The fact Opinion(v,id,report) is obtained as above. Although the capability sent back on channel pc has type **Un**, its successful decryption yields the fact PCMember(v) and thus enables Report(v,id,report) by clause B.

5.2 Offline Delegation, with Certificate Chains

Our second implementation relies instead on explicit chains of delegation certificates. It does not require that the conference system be contacted when delegation occurs; on the other hand, the system may have to check a list of certificates before accepting an incoming report. Moreover, we rely on self-authenticated capabilities under key ka for representing initial refereeing requests, instead of messages on the private database channel refereedb.

The idea is that, when a referee v files a report for paper id, she also presents a delegation chain showing she is authorized to file the report. In the implementation, we let a *delegation chain proving* Referee(v,id) be a message in one of two forms:

—either an authenticated encryption {v,id,**ok**}ka where ka is the key used by the PC chair to appoint referees directly, implying Referee(v,id);

—or a tuple (t,{v,id,**ok**}kdt,ct), where t is a principal with delegation key kdt, so that {v,id,**ok**}kdt proves Delegate(t,v,id), and ct is a (shorter) delegation chain proving the fact Referee(t,id).

Given clause C governing delegation, an easy bottom-up argument establishes that the existence of such a delegation chain does indeed prove Referee(v,id). The following code for accepting and checking a delegation chain supports this inductive argument.

```
( Delegate(U,W,ID):−Delegate(U,V,ID),Delegate(V,W,ID) ) |
( Delegate(U,U,ID):−Opinion(U,ID,R) ) |
new ka:Key((u:Un,(id:Un,Ok(Referee(u,id)))));
(!in filedelegatereport(v,e,cv);
  in pwdb(=v,kdv,krv); decrypt e as {id,report,_}krv;
  new link:Ch(u:Un,c:Un,Ok(Delegate(u,v,id))); out link(v,cv,ok) |
  !in link(u,cu,_);
   ( decrypt cu as {=u,=id,_}ka; expect Report(v,id,report)) |
   ( tuple cu as (t,delegation,ct); in pwdb(=t,kdt,_);
     decrypt delegation as {=u,=id,_}kdt; out link(t,ct,ok)) |
```

The two auxiliary clauses make Delegate reflexive and transitive; these clauses give us more freedom but they do not affect the outcome of our policy—one can check that these two clauses are redundant in any derivation of Report.

The process guarded by the replicated input on channel filedelegatereport allocates a private channel link and uses that channel recursively to verify, one certificate at a time, that the message cv filed with the report is indeed a delegation chain proving Referee(v,id). The process guarded by link has two cases: the base case (**decrypt** cu) verifies an initial refereeing request and finally accepts the report as valid; the recursive case (**tuple** cu) verifies a delegation step then continues on the rest of the chain (ct). The type assigned to link precisely states our loop invariant: Delegate(u,v,id) proves that there is a valid delegation chain from u (the current delegator) down to v (the report writer) for paper id.

PROPOSITION 3. *Let E_{Un} assign type* **Un** *to createReviewer, createPCMember, sendreportonline, delegateonline, filereport, filedelegate, filepcreport, filedelegatereport, and any other name in its domain.*

Let E_P assign the types displayed above to pwdb, refereedb, kp, and ka.

Let P be a process such that $E_{Un}, E_P \vdash P$.

Let Q be the process comprising all process fragments in this section followed by P.

We have $E_{Un} \vdash Q$, and hence Q is robustly safe.

This proposition is proved by typing Q then applying Theorem 3. In its statement, the process P has access to the private keys and channels collected in E_P; this process accounts for any trusted parts of the server left undefined, including for instance code that assigns papers to reviewers by issuing facts on Referee and using them to populate refereedb and generate valid certificates under key ka. We may simply take $P = \mathbf{0}$, or let P introduce its own policy extensions, as long as it complies with the typing environments E_{Un} and E_P.

In addition, the context (implicitly) enclosing Q in our statement of robust safety accounts for any untrusted part of the system, including the opponent, but also additional code for the reviewers interacting with Q (and possibly P) using the names collected in E_{Un}, and in particular the free names of Q. Hence, the context may impersonate referees, intercept messages on free channels, then send on channel filedelegatereport any term computed from intercepted messages. The proposition confirms that minimal typing assumptions on P suffice to guarantee the robust safety of Q.

6. APPLICATION: A DEFAULT IMPLEMENTATION FOR DATALOG

In this section, we describe a translation from Datalog programs to the spi calculus. To each predicate p and arity n, we associate a fresh name p_n with a channel type $T_{p,n}$. Unless the predicate p occurs with different arities, we omit indexes and write just p and T_p for p_n and $T_{p,n}$. Relying on some preliminary renaming, we also reserve a set of names $\mathscr{V}$ for Datalog variables. The translation is given below.

TRANSLATION FROM DATALOG TO THE SPI CALCULUS: $[\![S]\!]$

$T_{p,n} = \mathbf{Ch}(x_1{:}\mathbf{Un},\ldots,x_n{:}\mathbf{Un},\mathbf{Ok}(p(x_1,\ldots,x_n)))$

$[\![S]\!] = \prod_{C\in S}[\![C]\!] \qquad [\![\varnothing]\!] = \mathbf{0}$
$[\![L:-L_1,\ldots,L_m]\!] = ![\![L_1,\ldots,L_m]\!]^{\varnothing}[[\![L]\!]^+] \quad$ for $m \geq 0$
$[\![p(u_1,\ldots,u_n)]\!]^+ = \mathbf{out}\ p_n(u_1,\ldots,u_n,\mathbf{ok})$
$[\![L_1,L_2,\ldots,L_m]\!]^{\Sigma}[\cdot] = [\![L_1]\!]^{\Sigma}\left[[\![L_2,\ldots,L_m]\!]^{\Sigma\cup fv(L_1)}[\cdot]\right] \qquad [\![\varepsilon]\!]^{\Sigma}[\cdot] = [\cdot]$
$[\![p(u_1,\ldots,u_n)]\!]^{\Sigma}[\cdot] = \mathbf{in}\ p_n(\underline{u}_1,\ldots,\underline{u}_n,{=}\mathbf{ok});[\cdot]$
where $\underline{u}_i$ is u_i when $u_i \notin (\mathscr{V}\setminus(\Sigma\cup fv(u_{j<i})))$ and $\underline{u}_i$ is $=u_i$ otherwise.

$P \Downarrow_L$ when $\exists P'.P \rightarrow^*_{\equiv} P' \mid [\![L]\!]^+$

The process $[\![S]\!]$ represents the whole program S. The process $[\![L:-L_1,\ldots,L_m]\!]$ is a replicated process representing the clause $L:-L_1,\ldots,L_m$. The process $[\![L]\!]^+$ is an output representing the conclusion L of a clause. The context $[\![L_1,L_2,\ldots,L_m]\!]^{\Sigma}[\cdot]$, where $[\cdot]$ is a hole to be filled with a process, represents the body of a clause. Finally, the predicate $P \Downarrow_L$ holds if the process P eventually produces an output representing the fact L.

For example, using the policy of Section 2, the translation of predicate Report uses a channel Report of type $T_{Report} = \mathbf{Ch}(U{:}\mathbf{Un},ID{:}\mathbf{Un},R{:}\mathbf{Un},\mathbf{Ok}(\text{Report(U,ID,R)}))$ and the translation of clause A yields the process

$$\begin{aligned}&[\![\text{Report(U,ID,R)}{:}-\text{Referee(U,ID),Opinion(U,ID,R)}]\!] \quad = \\ &\quad !\mathbf{in}\ \text{Referee(U,ID,}{=}\mathbf{ok});\ \mathbf{in}\ \text{Opinion(=U,=ID,R,}{=}\mathbf{ok});\ \mathbf{out}\ \text{Report(U,ID,R,}\mathbf{ok})\end{aligned}$$

The next lemma states that a Datalog program, considered as a policy, is well typed when placed in parallel with its own translation.

LEMMA 3. *Let S be a Datalog program using predicates $\widetilde{p_n}$ and names $\widetilde{y}$ with $fn(S) \subseteq \{\widetilde{y}\}$. Let $E = \widetilde{y{:}\mathbf{Un}}, \widetilde{p_n{:}T_{n,p}}$. We have $E \vdash S \mid [\![S]\!]$.*

More precisely, the lemma also shows that our translation is compositional: one can translate some part of a logical policy, develop some specific protocols that comply with some other part of the policy, then put the two implementations in parallel and rely on messages on channels p_n to safely exchange facts concerning shared predicates.

Lemma 3 establishes that our translation is correct by typing. The following theorem also states that the translation is complete: any fact that logically follows from the Datalog program can be observed in the pi calculus.

THEOREM 4. *Let S be a Datalog program and F a fact:* $S \models F$ *if and only if* $[\![S]\!] \Downarrow_F$.

To illustrate our translation, we sketch an alternative implementation of our conference management server. Instead of coding the recursive processing of messages sent by sub-referees, as in Section 5, we set up a replicated input for each kind of certificate, with code to check the certificate and send a message on a channel of the translation. Independently, when a fact is expected, we simply read it on a channel of the translation. For instance, to process incoming reports, we may use the code

```
!in trivial_filereport(v,id,report);
 in Report(=v,=id,=report,=ok); expect Report(v,id,report)
```

The translation of clause A sends a matching message on Report, provided the system sends matching messages on Opinion and Referee. This approach is correct and complete, but also non-deterministic and very inefficient. As a refinement, since any (well-typed) program can access the channels of the translation, one may use the translation as a default implementation for some clauses and provide optimized code for others.

7. CONCLUSIONS AND FUTURE WORK

We presented a spi calculus with embedded authorization policies, a type system that can statically check conformance to a policy (even in the presence of active attackers), and a series of applications coded using a prototype implementation.

In itself, our type system does not “solve” authorization: the security of a well-typed program still relies on a careful (manual) review of the policy, on the discriminating statement of trusted facts (or even rules) in the program, and on the presence of expectations marking sensitive actions—indeed, in our setting, every program is safe for a sufficiently permissive policy. Nonetheless, our type system statically enforces a discipline prescribed by the policy across the program, as it uses channels and cryptographic primitives to process messages, and can facilitate code reviews.

As it stands, our calculus and type system are simple and illustrative, but have many limitations that may be investigated. For example, we do not consider revocation or temporary activation of authorization statements. From a logical viewpoint, many authorization languages include notions of locality and partial trust, considering for example facts and clauses relative to each principal [Abadi et al. 1993]. A first step will be to consider a combination of the present system with ideas from a recent work [Gordon and Jeffrey 2005] on a type system for checking secrecy in a pi calculus despite the compromise of some principals. We are also exploring extensions of our type system to support, for instance, some subtyping, public-key cryptographic primitives, and linearity properties. More experimentally, we plan to extend our typechecker and symbolic interpreter, and to study their integration with other proof techniques.

ACKNOWLEDGMENTS

Karthikeyan Bhargavan contributed to several discussions at the start of this project, and commented on a draft of this paper. Martín Abadi, Nick Benton, and the anonymous conference reviewers made useful suggestions.

A. DATALOG PROOFS

This section develops proofs of Theorem 1 and Propositions 1 and 2.

LEMMA 4. *If $S \models F$ then $S \cup \{C\} \models F$.*

PROOF. By induction on the depth of the derivation tree for $S \models F$. $\square$

LEMMA 5. *If $S \models F$ and $S \cup \{F\} \models F'$ then $S \models F'$.*

PROOF. By induction on the derivation of $S \cup \{F\} \models F'$. $\square$

LEMMA 6. *If $S \models F$ and σ replaces names with messages, then $S\sigma \models F\sigma$.*

PROOF. By induction on the depth of the derivation tree for $S \models F$. $\square$

RESTATEMENT OF THEOREM 1. *For all clauses C and sets of clauses S, (1) and (2) are equivalent:*

(1) *For all sets of facts S', $\{F \mid S' \cup \{C\} \models F\} \subseteq \{F \mid S' \cup S \models F\}$;*
(2) *$S \cup \{L_1\sigma, \dots, L_n\sigma\} \models L\sigma$, where $C = L :- L_1, \dots, L_n$ and $\sigma = \{\widetilde{x}/\widetilde{X}\}$ is an injective substitution such that $\{\widetilde{x}\} \cap (fn(S) \cup fn(C)) = \varnothing$ and $\widetilde{X} = fv(L_1, \dots, L_n)$.*

PROOF. That (2) implies (1) follows by induction on the structure of S'. The inductive case uses a nested induction on the derivations of $\models$, and Lemmas 4, 5, and 6. That (1) implies (2) is by definition of (Infer Fact). $\square$

The next two lemmas prove monotonicity and closure under substitutions of Datalog, which are the properties (Mon) and (Subst) needed to show that it is an authorization logic.

RESTATEMENT OF PROPOSITION 1. *If $S \models C$ then $S \cup \{C'\} \models C$.*

PROOF. By cases on the last rule used in the derivation of $S \models C$, using Lemma 4. $\square$

RESTATEMENT OF PROPOSITION 2. *If $S \models C$ and σ sends names to messages, $S\sigma \models C\sigma$.*

PROOF. By cases on the last rule used in the derivation of $S \models C$, using Lemma 6 and standard properties of substitutions. $\square$

The following is a strengthening property of authorization logics with respect to sets of clauses equivalent up to fresh renamings. It will be used in the proofs of Appendix B.2.

LEMMA 7. *Let $(\mathscr{C}, fn, \models)$ be an authorization logic, and let $C \in \mathscr{C}$, $S, S' \subseteq \mathscr{C}$. If $S \cup S\{\widetilde{y}/\widetilde{x}\} \cup S' \models C$ where $\{\widetilde{y}\} \cap fn(S \cup S' \cup \{C\}) = \varnothing$ and the $\widetilde{y}$ are distinct, then $S \cup S' \models C$.*

PROOF. Follows from the property (Subst) of an authorization logic and from standard properties of injective substitutions of fresh names. $\square$

B. SPI CALCULUS PROOFS

This section has three parts. Appendix B.1 contains the definition of an alternative, more explicit, type system for the spi calculus and the proof that it is equivalent to the one given in the main body of the paper. Appendix B.2 shows the main properties of the type system—subject congruence and subject reduction, in particular. Appendix B.3 contains the proofs of opponent typability and of the main results of the paper concerning safety.

All the results in this section are independent of the choice of authorization logics.

B.1 An Alternative Type System

We define a type system for the spi calculus that uses *guarantees* to represent the top level, active statements from processes while maintaining invariance under renaming of bound names. It is informative to capture these guarantees explicitly with typing rules, rather than to capture them implicitly via the separate function $env(P)$ as used in the system in the main body of the paper. We show the equivalence of the two type systems, and induce soundness of the main system from proofs about the alternative system. Still, we expect that a direct proof of soundness for the main system would proceed similarly to the proof for the alternative system.

GUARANTEES:

$G,H ::=$	guarantee
$\mathbf{0}$	no guarantee
$G \mid H$	composition
$\mathbf{new}\ x{:}T; G$	restriction
C	clause C can be assumed

The function $env(-)$ defined below, which given a guarantee extracts the corresponding environment, is analogous to the one given in Section 4 for processes.

FROM GUARANTEES TO ENVIRONMENTS: $env(G)$

$env(\mathbf{0})^{\varnothing} = \varnothing \qquad env(C)^{\varnothing} = C$

$env(G \mid H)^{\widetilde{x},\widetilde{y}} = env(G)^{\widetilde{x}}, env(H)^{\widetilde{y}}$ (where $\{\widetilde{x},\widetilde{y}\} \cap fn(G \mid H) = \varnothing$)

$env(\mathbf{new}\ x{:}T; G)^{x,\widetilde{x}} = x{:}T, env(G)^{\widetilde{x}}$ (where $\{\widetilde{x}\} \cap fn(G) = \varnothing$)

Convention: $env(G) \triangleq env(G)^{\widetilde{x}}$ for some distinct $\widetilde{x}$ such that $env(G)^{\widetilde{x}}$ is defined.

Guarantee subsumption is a binary relation on guarantees characterized by the axioms (G Sub Idem) and (G Sub Order). If $G \sqsubseteq H$ then intuitively G contains fewer facts than H. Structural congruence for guarantees is defined in terms of subsumption.

GUARANTEE SUBSUMPTION: $G \sqsubseteq H$

$G \sqsubseteq G$	(G Sub Refl)
$G \sqsubseteq H, H \sqsubseteq G' \Rightarrow G \sqsubseteq G'$	(G Sub Trans)
$G \sqsubseteq H \Rightarrow \mathbf{new}\ x{:}T; G \sqsubseteq \mathbf{new}\ x{:}T; H$	(G Sub Res)
$G \sqsubseteq G' \Rightarrow G \mid H \sqsubseteq G' \mid H$	(G Sub Par)
$G \mid \mathbf{0} \sqsubseteq G$	(G Sub Par Zero)
$G \mid H \sqsubseteq H \mid G$	(G Sub Par Comm)
$(G \mid G') \mid H \sqsubseteq G \mid (G' \mid H)$	(G Sub Par Assoc)
$G \mid G \sqsubseteq G$	(G Sub Idem)
$G \sqsubseteq G \mid H$	(G Sub Order)
$\mathbf{new}\ x{:}T; (G \mid H) \sqsubseteq G \mid \mathbf{new}\ x{:}T; H$	(G Sub Res ParL) (for $x \notin fn(G)$)
$G \mid \mathbf{new}\ x{:}T; H \sqsubseteq \mathbf{new}\ x{:}T; (G \mid H)$	(G Sub Res ParR) (for $x \notin fn(G)$)
$\mathbf{new}\ x_1{:}T_1; \mathbf{new}\ x_2{:}T_2; G \sqsubseteq$	(G Sub Res Res)
$\quad \mathbf{new}\ x_2{:}T_2; \mathbf{new}\ x_1{:}T_1; G$	(for $x_1 \neq x_2, x_1 \notin fn(T_2), x_2 \notin fn(T_1)$)

STRUCTURAL CONGRUENCE FOR GUARANTEES: $G \equiv H$

$G \equiv H \stackrel{\triangle}{=} G \sqsubseteq H$ and $H \sqsubseteq G$ (G Struct)

Below we give the rules defining the type system with guarantees. The rules (ProcG Res), (ProcG Par), and (ProcG Fact) grow the guarantee of a process, (ProcG Rep) leaves it invariant, and all the other rules set it to **0**.

ADDITIONAL JUDGMENT:

$E \vdash P : G$ good process P guaranteeing G

GOOD PROCESSES: $E \vdash P : G$ (IN ENVIRONMENT E, PROCESS P GRANTS G).

(ProcG Nil) $$\frac{E \vdash \diamond}{E \vdash \mathbf{0} : \mathbf{0}}$$

(ProcG Rep) $$\frac{E \vdash P : G}{E \vdash !P : G}$$

(ProcG Res) $$\frac{E, x{:}T \vdash P : G \quad T \textit{ generative}}{E \vdash \mathbf{new}\ x{:}T; P : \mathbf{new}\ x{:}T; G}$$

(ProcG Par) $$\frac{E, \mathit{env}(G_2) \vdash P : G_1 \quad E, \mathit{env}(G_1) \vdash Q : G_2 \quad \mathit{fn}(P \mid Q) \subseteq \mathit{dom}(E)}{E \vdash P \mid Q : G_1 \mid G_2}$$

(ProcG Input) $$\frac{E \vdash M : \mathbf{Ch}(T) \quad E, x{:}T \vdash P : G}{E \vdash \mathbf{in}\ M(x{:}T); P : \mathbf{0}}$$

(ProcG Input Un) $$\frac{E \vdash M : \mathbf{Un} \quad E, x{:}\mathbf{Un} \vdash P : G}{E \vdash \mathbf{in}\ M(x{:}\mathbf{Un}); P : \mathbf{0}}$$

(ProcG Output) $$\frac{E \vdash M : \mathbf{Ch}(T) \quad E \vdash N : T}{E \vdash \mathbf{out}\ M(N) : \mathbf{0}}$$

(ProcG Output Un) $$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un}}{E \vdash \mathbf{out}\ M(N) : \mathbf{0}}$$

(ProcG Decrypt) $$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Key}(T) \quad E, y{:}T \vdash P : G}{E \vdash \mathbf{decrypt}\ M\ \mathbf{as}\ \{y{:}T\}N; P : \mathbf{0}}$$

(ProcG Decrypt Un) $$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un} \quad E, y{:}\mathbf{Un} \vdash P : G}{E \vdash \mathbf{decrypt}\ M\ \mathbf{as}\ \{y{:}\mathbf{Un}\}N; P : \mathbf{0}}$$

(ProcG Match) $$\frac{E \vdash M : (x{:}T, U) \quad E \vdash N : T \quad E, y{:}U\{N/x\} \vdash P : G}{E \vdash \mathbf{match}\ M\ \mathbf{as}\ (N, y{:}U\{N/x\}); P : \mathbf{0}}$$

(ProcG Match Un) $$\frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un} \quad E, y{:}\mathbf{Un} \vdash P : G}{E \vdash \mathbf{match}\ M\ \mathbf{as}\ (N, y{:}\mathbf{Un}); P : \mathbf{0}}$$

(ProcG Split) $$\frac{E \vdash M : (x{:}T, U) \quad E, x{:}T, y{:}U \vdash P : G}{E \vdash \mathbf{split}\ M\ \mathbf{as}\ (x{:}T, y{:}U); P : \mathbf{0}}$$

(ProcG Split Un) $$\frac{E \vdash M : \mathbf{Un} \quad E, x{:}\mathbf{Un}, y{:}\mathbf{Un} \vdash P : G}{E \vdash \mathbf{split}\ M\ \mathbf{as}\ (x{:}\mathbf{Un}, y{:}\mathbf{Un}); P : \mathbf{0}}$$

(ProcG Query)
$$\frac{E, C \vdash \diamond \quad clauses(E) \models C}{E \vdash \textbf{expect}\ C : \mathbf{0}}$$

(ProcG Fact)
$$\frac{E, C \vdash \diamond}{E \vdash C : C}$$

GENERIC JUDGMENT: $\mathscr{J}$

$\mathscr{J} ::= \diamond \mid M : T \mid P : G$ meta-syntax for the generic judgment

$fn(\diamond) = \varnothing \quad fn(M : T) = fn(M) \cup fn(T) \quad fn(P : G) = fn(P) \cup fn(G)$

$\diamond\sigma = \diamond \quad (M : T)\sigma = M\sigma : T\sigma \quad (P : G)\sigma = P\sigma : G\sigma$

We can show now that the two type systems are equivalent.

LEMMA 8. *$E \vdash P$ and $env(P)^{\widetilde{x}} = E'$ if and only if $E \vdash P : G$, for some G such that $E' = env(G)^{\widetilde{x}}$.*

PROOF. ($\Rightarrow$) By induction on the derivation of $E \vdash P$ and by definition of $env(G)$. ($\Leftarrow$) By induction on the derivation of $E \vdash P : G$ and by definition of $env(P)$. □

B.2 Properties of the Type System

We proceed to show the main properties of the type system, in particular subject congruence and subject reduction, which together give type preservation (Lemma 1).

Before proving subject congruence and subject reduction in detail, we state without proof a series of fairly standard technical properties of the type system. The companion technical report [Fournet et al. 2005a] describes the proofs of all of these properties.

LEMMA 9. *If $E \vdash x : T$ and $E \vdash x : U$ then $T = U$.*

LEMMA 10. *If $E \vdash P : G$ and $E' \vdash P : G'$ then $G = G'$.*

LEMMA 11. *Let $\mathscr{J}$ range over $\{\diamond, M : T, P : G\}$. (i) If $E, x{:}U, E' \vdash \mathscr{J}$ and U is generative and $x \notin fn(\mathscr{J}) \cup fn(E')$ then $E, E' \vdash \mathscr{J}$. (ii) If $E, C, E' \vdash \diamond$ then $E, E' \vdash \diamond$.*

LEMMA 12. *If $E_1, E_2, E_3, E_4 \vdash \mathscr{J}$ and $dom(E_2) \cap fn(E_3) = \varnothing$ and $fn(E_2) \cap dom(E_3) = \varnothing$ then $E_1, E_3, E_2, E_4 \vdash \mathscr{J}$.*

LEMMA 13. *(i) If $G \sqsubseteq G'$ then $fn(G) \subseteq fn(G')$. (ii) If $G \equiv G'$ then $fn(G) = fn(G')$.*

LEMMA 14. *If $E, env(G)^{\widetilde{x}}, E' \vdash \mathscr{J}$ and $G \sqsubseteq G'$, $fn(G) = fn(G')$ and $\{\widetilde{x}\} \cap (fn(E') \cup fn(\mathscr{J})) = \varnothing$, then $E, env(G')^{\widetilde{z}}, E' \vdash \mathscr{J}$ and $\{\widetilde{z}\} \cap (fn(E') \cup fn(\mathscr{J})) = \varnothing$.*

LEMMA 15. *If $E, env(G)^{\widetilde{x}}, E' \vdash P : G$ and $\{\widetilde{x}\} \cap (fn(P) \cup fn(E')) = \varnothing$ then $E, E' \vdash P : G$.*

LEMMA 16. *(i) If $E, E' \vdash \mathscr{J}$ and $fn(C) \subseteq dom(E)$ then $E, C, E' \vdash \mathscr{J}$. (ii) If $E, E' \vdash \mathscr{J}$, $fn(T) \subseteq dom(E)$ and $x \notin dom(E, E')$, then $E, x{:}T, E' \vdash \mathscr{J}$.*

LEMMA 17. *If $E_1, x{:}T, E_2 \vdash \mathscr{J}$ and $E_1 \vdash M : T$ then $E_1, E_2\{M/x\} \vdash \mathscr{J}\{M/x\}$.*

LEMMA 18. *If $E \vdash P : G$ and $P \equiv P'$ then there exists a G' such that $E \vdash P' : G'$ and $G \equiv G'$.*

PROOF. By induction on the derivation of $P \equiv P'$ we show:

(1) If $E \vdash P : G$ then $E \vdash P' : G'$;

(2) If $E \vdash P' : G'$ then $E \vdash P : G$.

We show only the more interesting cases.

(Struct Par). Suppose $P \mid Q \equiv P' \mid Q$.
By hypothesis, $P \equiv P'$.
By hypothesis of (1), $E \vdash P \mid Q : G$.
By (ProcG Par), $E, env(G_2)^{\tilde{y}} \vdash P : G_1$ and $E, env(G_1)^{\tilde{x}} \vdash Q : G_2$ and $fn(P \mid Q) \subseteq dom(E)$ where $G = G_1 \mid G_2$.
By inductive hypothesis, $E, env(G_2)^{\tilde{y}} \vdash P' : G'_1 \equiv G_1$.
By Lemma 13, $fn(G_1) = fn(G'_1)$.
By Lemma 14, $E, env(G'_1)^{\tilde{x}} \vdash Q : G_2$.
By (ProcG Par), $E \vdash P' \mid Q : G'_1 \mid G_2$.
By definition of $\equiv$ by (G Sub Par), $G \equiv G'_1 \mid G_2$.
The proof for (2) is symmetric.

(Struct Par Assoc). Suppose $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$.
By hypothesis of (1), $E \vdash (P \mid Q) \mid R : G$.
By (ProcG Par), $E, env(G_2)^{\tilde{z}} \vdash P \mid Q : G_1$ and $E, env(G_1) \vdash R : G_2$, $fn((P \mid Q) \mid R) \subseteq dom(E)$, and $G = G_1 \mid G_2$.
By (ProcG Par), $E, env(G_2)^{\tilde{z}}, env(G_4)^{\tilde{y}} \vdash P : G_3$ and $E, env(G_2)^{\tilde{z}}, env(G_3)^{\tilde{x}} \vdash Q : G_4$, $fn(P \mid Q) \subseteq dom(E, env(G_2))$, and $G_1 = G_3 \mid G_4$.
By Lemma 12, $E, env(G_4)^{\tilde{y}}, env(G_2)^{\tilde{z}} \vdash P : G_3$ and $E, env(G_3)^{\tilde{x}}, env(G_2)^{\tilde{z}} \vdash Q : G_4$.
By (ProcG Par), $E, env(G_3)^{\tilde{x}} \vdash Q \mid R : G_4 \mid G_2$.
By (ProcG Par), $E \vdash P \mid (Q \mid R) : G_3 \mid (G_4 \mid G_2)$.
By (G Sub Par Assoc), $G = (G_3 \mid G_4) \mid G_2 \equiv G_3 \mid (G_4 \mid G_2)$.
The proof for (2) is similar.

(Struct Repl Unfold). Suppose $!P \equiv P \mid !P$.
By hypothesis of (1), $E \vdash !P : G$.
By (ProcG Rep), $E \vdash P : G$.
By Lemma 16, $E, env(G) \vdash !P : G$ and $E, env(G) \vdash P : G$.
By (ProcG Par), $E \vdash P \mid !P : G \mid G$.
By (G Sub Idem), $G \mid G \equiv G$.
By hypothesis of (2), $E \vdash P \mid !P : G$.
By (ProcG Par), $E, env(G_2) \vdash P : G_1$ and $E, env(G_1) \vdash !P : G_2$, where $G = G_1 \mid G_2$.
By (ProcG Rep), $E, env(G_1) \vdash P : G_2$.
By Lemma 10, $G_1 = G_2$.
By Lemma 15, $E \vdash !P : G_2$.
By (G Sub Idem), $G \equiv G_2$.

(Struct Res Par). Suppose $\textbf{new}\ x{:}T; (P \mid Q) \equiv P \mid \textbf{new}\ x{:}T; Q$.
By hypothesis, $x \notin fn(P)$.
By hypothesis of (1), $E \vdash \textbf{new}\ x{:}T; (P \mid Q) : G$.
By (ProcG Res), $E, x{:}T \vdash P \mid Q : G'$ where $G = \textbf{new}\ x{:}T; G'$.
By (ProcG Par), $E, x{:}T, env(G_2)^{\tilde{y}} \vdash P : G_1$ and $E, x{:}T, env(G_1)^{\tilde{x}} \vdash Q : G_2$ where $G' = G_1 \mid G_2$.
By (ProcG Res), $E, env(G_1)^{\tilde{x}} \vdash \textbf{new}\ x{:}T; Q : G_2$.
Since $x \notin fn(P)$, by Lemma 11, $E, env(G_2)^{\tilde{y}} \vdash P : G_1$.
By (ProcG Par), $E \vdash P \mid \textbf{new}\ x{:}T; Q$.
The proof for (2) is similar, using Lemma 16 instead of Lemma 11.

(Struct Res Res). Suppose $\mathbf{new}\ x_1{:}T_1; \mathbf{new}\ x_2{:}T_2; P \equiv \mathbf{new}\ x_2{:}T_2; \mathbf{new}\ x_1{:}T_1; P$.
By hypothesis, $x_1 \neq x_2, x_1 \notin fn(T_2), x_2 \notin fn(T_1)$.
By (ProcG Res), $E, x_1{:}T_1 \vdash \mathbf{new}\ x_2{:}T_2; P : G$.
By (ProcG Res), $E, x_1{:}T_1, x_2{:}T_2 \vdash P : G$.
Since $x_1 \neq x_2, x_1 \notin fn(T_2), x_2 \notin fn(T_1)$, by Lemma 12, $E, x_2{:}T_2, x_1{:}T_1 \vdash P : G$.
By two applications of (ProcG Res), $E \vdash \mathbf{new}\ x_2{:}T_2;, \mathbf{new}\ x_1{:}T_1; P : G$.
The proof for (2) is symmetric. □

LEMMA 19. *If $E \vdash P : G$ and $P \to P'$ then there exists a G' such that $E \vdash P' : G'$ and $G \sqsubseteq G'$.*

PROOF. The proof is by induction on the derivation of $P \to P'$.

(Red Comm). Suppose $\mathbf{out}\ a(M) \mid \mathbf{in}\ a(x{:}T); P \to P\{M/x\}$.
By hypothesis of the lemma, $E \vdash \mathbf{out}\ a(M) \mid \mathbf{in}\ a(x{:}T); P : G$.
By (ProcG Par), $E \vdash \mathbf{out}\ a(M) : \mathbf{0}$ and $E \vdash \mathbf{in}\ a(x{:}T); P : \mathbf{0}$, and $G = \mathbf{0} \mid \mathbf{0}$ because the only rules applicable to the premises are (ProcG Output) or (ProcG Output Un) for the first sub-term, and (ProcG Input) or (ProcG Input Un) for the second.
We distinguish two cases.

—If $E \vdash \mathbf{out}\ a(M) : \mathbf{0}$ is derived by (ProcG Output) then $E \vdash a : \mathbf{Ch}(U)$ and $E \vdash M : U$, and by Lemma 9, $E \vdash \mathbf{in}\ a(x{:}T); P : \mathbf{0}$ is derived by (ProcG Input), and $T = U$ and $E, x{:}U \vdash P : G'$ for some G'.
By Lemma 17, $E \vdash P\{M/x\} : G'\{M/x\}$.

—If $E \vdash \mathbf{out}\ a(M) : \mathbf{0}$ is derived by (ProcG Output Un) then $E \vdash a : \mathbf{Un}$ and $E \vdash M : \mathbf{Un}$, and by Lemma 9, $E \vdash \mathbf{in}\ a(x{:}T); P : \mathbf{0}$ is derived by (ProcG Input Un), and $T = \mathbf{Un}$ and $E, x{:}\mathbf{Un} \vdash P : G'$ for some G'.
By Lemma 17, $E, x{:}\mathbf{Un} \vdash P\{M/x\} : G'\{M/x\}$.

(Red Decrypt). Suppose $\mathbf{decrypt}\ \{M\}k\ \mathbf{as}\ \{y{:}T\}k; P \to P\{M/y\}$.
If $E \vdash \mathbf{decrypt}\ \{M\}k\ \mathbf{as}\ \{y{:}T\}k; P : G$ is derived by (ProcG Decrypt) then $G = \mathbf{0}$, $E \vdash M : T$, $E \vdash k : \mathbf{Key}(T)$, and $E, y{:}T \vdash P : G'$.
By Lemma 17, $E \vdash P\{M/y\} : G'\{M/y\}$.
The case for rule (ProcG Decrypt Un) is similar.

(Red Split). Suppose $\mathbf{split}\ (M,N)\ \mathbf{as}\ (x{:}T, y{:}U); P \to P\{M/x\}\{N/y\}$.
If $E \vdash \mathbf{split}\ (M,N)\ \mathbf{as}\ (x{:}T, y{:}U); P : G$ is derived by (ProcG Split) then $G = \mathbf{0}$, $E \vdash (M,N) : (x{:}T, U)$ and $E, x{:}T, y{:}U \vdash P : G'$.
By (Msg Pair), $E \vdash M : T$ and $E \vdash N : U\{M/x\}$.
By Lemma 17, $E, y{:}U\{M/x\} \vdash P\{M/x\} : G'\{M/x\}$.
By Lemma 17, $E \vdash P\{M/x\}\{N/y\} : G'\{M/x\}\{N/y\}$.
The case for rule (ProcG Split Un) is similar.

(Red Match). Suppose $\mathbf{match}\ (M,N)\ \mathbf{as}\ (M, y{:}U); P \to P\{N/y\}$.
If $E \vdash \mathbf{match}\ (M,N)\ \mathbf{as}\ (M, y{:}U); P : G$ is derived by (ProcG Match) then $G = \mathbf{0}$, $E \vdash (M,N) : (x{:}T, U)$, $E \vdash M : T$ and $E, y{:}U\{M/x\} \vdash P : G'$.
By (Msg Pair), $E \vdash N : U\{M/x\}$.
By Lemma 17, $E \vdash P\{N/y\} : G'\{N/y\}$.
The case for rule (ProcG Match Un) is similar.

(Red Par). Suppose $P \mid Q \to P' \mid Q$.
By hypothesis, $P \to P'$.

By hypothesis of the lemma, $E \vdash P \mid Q : G$. By (ProcG Par), $E, env(G_2) \vdash P : G_1$, $E, env(G_1) \vdash Q : G_2$, $fn(P \mid Q) \subseteq dom(E)$, and $G = G_1 \mid G_2$.
By inductive hypothesis, $E, env(G_2) \vdash P' : G'$, for some G' such that $G_1 \sqsubseteq G'$.
By Lemma 16, $E, env(G') \vdash Q : G_2$.
By (ProcG Par), $E \vdash P \mid Q : G' \mid G_2$.
By definition of $(GSubPar)$, $G_1 \mid G_2 \sqsubseteq G' \mid G_2$.

(Red Res). Suppose $\mathbf{new}\ x{:}T; P \to \mathbf{new}\ x{:}T; P'$.
By hypothesis, $P \to P'$.
By hypothesis of the lemma, $E \vdash \mathbf{new}\ x{:}T; P : G$.
By (ProcG Res), $E, x{:}T \vdash P : G'$ and $G = \mathbf{new}\ x{:}T; G'$.
By inductive hypothesis, $E, x{:}T \vdash P' : G''$ and $G' \sqsubseteq G''$.
By (ProcG Res), $E \vdash \mathbf{new}\ x{:}T; P' : \mathbf{new}\ x{:}T; G''$.
By (G Sub Res), $(x{:}T)G' \sqsubseteq (x{:}T)G''$.

(Red Struct). Suppose $P \to P'$.
By hypothesis, $P \equiv Q, Q \to Q', Q' \equiv P'$.
By Lemma 18 on $E \vdash P : G$, $E \vdash Q : G_1$ where $G_1 \equiv G$.
By inductive hypothesis on $E \vdash Q : G_1$, $E \vdash Q' : G_2$ and $G_1 \sqsubseteq G_2$.
By Lemma 18, $E \vdash P' : G_3 \equiv G_2$.
By definition of $\equiv$ and by (G Sub Trans), $G \sqsubseteq G_3$. □

RESTATEMENT OF LEMMA 1. *If $E \vdash P$ and either $P \equiv P'$ or $P \to P'$ then $E \vdash P'$.*

PROOF. By definition of $E \vdash P$ and Lemmas 18 and 19. □

B.3 Type Safety

We describe the proofs of opponent typability and of the main results of the paper concerning safety.

B.3.1 *Properties of the Opponent.* The following two lemmas are proved by easy inductions.

LEMMA 20. *For any M, if $fn(M) = \{\widetilde{x}\}$ then $\widetilde{x}{:}\mathbf{Un} \vdash M : \mathbf{Un}$.*

LEMMA 21. *For any opponent P, $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash P : G$, where $fn(P) \subseteq \{\widetilde{x}\}$.*

RESTATEMENT OF LEMMA 2. *For any opponent P, $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash P$, where $fn(P) \subseteq \{\widetilde{x}\}$.*

PROOF. Follows directly from Lemma 21 and Lemma 8. □

B.3.2 *Safety and Robust Safety*

LEMMA 22. *If $E \vdash P : G$ and $clauses(env(G)^{\widetilde{x}}) = \{C_1, \dots, C_n\}$ then there exists a P' such that $P \equiv \mathbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \dots \mid C_n \mid P')$.*

PROOF. By induction on the derivation of $E \vdash P : G$.

(ProcG Fact). Suppose $E \vdash C : C$.
Process $P = P' = C$ is in the required form.

(ProcG Res). Suppose $E \vdash \mathbf{new}\ x{:}T; P : \mathbf{new}\ x{:}T; G$.
By hypothesis, $E, x{:}T \vdash P : G$.
By inductive hypothesis, $P \equiv \mathbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \dots \mid C_n \mid P')$ where $clauses(env(G)^{\widetilde{x}}) = \{C_1, \dots, C_n\}$.

By (Struct Res), $\textbf{new}\ x{:}T; P \equiv \textbf{new}\ x{:}T, \widetilde{x}{:}\widetilde{T}; (C_1 \mid \ldots \mid C_n \mid P')$.
By definition, $\mathit{clauses}(\mathit{env}(\textbf{new}\ x{:}T; G)^{x,\widetilde{x}}) = \{C_1, \ldots, C_n\}$.

(ProcG Rep). Suppose $E \vdash {!P} : G$.
By hypothesis, $E \vdash P : G$.
By inductive hypothesis, $P \equiv \textbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \ldots \mid C_n \mid P')$ where $\mathit{clauses}(\mathit{env}(G)^{\widetilde{x}}) = \{C_1, \ldots, C_n\}$.
By (Struct Repl) and (Struct Res Par),
$!P \equiv \textbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \ldots \mid C_n \mid P') \mid !P \equiv \textbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \ldots \mid C_n \mid (P' \mid !P))$.

(ProcG Par). Suppose $E \vdash P \mid Q : G_1 \mid G_2$.
By hypothesis, $E, \mathit{env}(G_2) \vdash P : G_1$, $E, \mathit{env}(G_1) \vdash Q : G_2$.
By inductive hypotheses, $P \equiv \textbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \ldots \mid C_n \mid P')$ where
$\mathit{clauses}(\mathit{env}(G_1)^{\widetilde{x}}) = \{C_1, \ldots, C_n\}$ and $Q \equiv \textbf{new}\ \widetilde{y}{:}\widetilde{U}; (C'_1 \mid \ldots \mid C'_m \mid Q')$ where
$\mathit{clauses}(\mathit{env}(G_2)^{\widetilde{y}}) = \{C'_1, \ldots, C'_m\}$.
By α-conversion and commutativity,
$P \mid Q \equiv \textbf{new}\ \widetilde{x}{:}\widetilde{T}, \widetilde{y}{:}\widetilde{U}; (C_1 \mid \ldots \mid C_n \mid C'_1 \mid \ldots \mid C'_m \mid (P' \mid Q'))$.
By definition, $\mathit{clauses}(\mathit{env}(G_1 \mid G_2)^{\widetilde{x},\widetilde{y}}) = \{C_1, \ldots, C_n, C'_1, \ldots, C'_m\}$

All the other cases are trivial, as $G = \mathbf{0}$, $\mathit{clauses}(\mathit{env}(\mathbf{0})^{\varnothing}) = \varnothing$, and $P = P'$. □

RESTATEMENT OF THEOREM 2. *If $E \vdash P$ and E is generative then P is safe.*

PROOF. We need to show that whenever $P \rightarrow^*_{\equiv} \textbf{new}\ \tilde{x}{:}\tilde{T}; (\textbf{expect}\ C \mid P')$, we can refactor P' so that $P' \equiv \textbf{new}\ \tilde{y}{:}\tilde{U}; (C_1 \mid \ldots \mid C_n \mid P'')$, and $\{C_1, \ldots, C_n\} \models C$, with $\{\widetilde{y}\} \cap \mathit{fn}(C) = \varnothing$.
By hypothesis, $E \vdash P$.
By Lemma 19 and Lemma 18, if $P \rightarrow^*_{\equiv} \textbf{new}\ \tilde{x}{:}\tilde{T}; (\textbf{expect}\ C \mid P')$ then
$E \vdash \textbf{new}\ \tilde{x}{:}\tilde{T}; (\textbf{expect}\ C \mid P') : G$, for some G.
This must follow from repeatedly applying (ProcG Res) from the premise
$E, \widetilde{x}{:}\tilde{T} \vdash \textbf{expect}\ C \mid P' : G_1$, where $G = \textbf{new}\ \widetilde{x}{:}\widetilde{T}; G_1$.
This must follow from (ProcG Par) and (ProcG Query), from the premises
(i) $E, \widetilde{x}{:}\tilde{T}, \mathit{env}(G_1)^{\widetilde{y}} \vdash \textbf{expect}\ C : \varnothing$ and
(ii) $E, \widetilde{x}{:}\tilde{T} \vdash P' : G_1$, where $\mathit{fn}(\textbf{expect}\ C) = \mathit{fn}(C) \subseteq \mathit{dom}(E)$ and
$\mathit{clauses}(E, \widetilde{x}{:}\tilde{T}, \mathit{env}(G_1)^{\widetilde{y}}) \models C$, and $\{\widetilde{y}\} \cap \mathit{fn}(C) = \varnothing$.
Assume, without loss of generality, that $\mathit{clauses}(\mathit{env}(G_1)^{\widetilde{y}}) = \{C_1, \ldots, C_n\}$.
By generativity of E and by definition, $\{C_1, \ldots, C_n\} \models C$.
By Lemma 22 on (ii), $P' \equiv \textbf{new}\ \widetilde{y}{:}\widetilde{U}; (C_1 \mid \ldots \mid C_n \mid P'')$. □

RESTATEMENT OF THEOREM 3. *If $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash P$ then P is robustly safe.*

PROOF. Consider an arbitrary opponent O, and let $\{\widetilde{z}\} = \mathit{fn}(O) \cup \{\widetilde{x}\}$.
By hypothesis $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash P : G$, for some G.
By Lemma 21, $\widetilde{z}{:}\widetilde{\mathbf{Un}} \vdash O : G'$, for some G'.
By Lemma 16, $\widetilde{z}{:}\widetilde{\mathbf{Un}}, \mathit{env}(G) \vdash O : G'$ and $\widetilde{z}{:}\widetilde{\mathbf{Un}}, \mathit{env}(G') \vdash P : G$.
By (ProcG Par), $\widetilde{z}{:}\widetilde{\mathbf{Un}} \vdash P \mid O : G \mid G'$.
By Theorem 2, $P \mid O$ is safe. □

C. ENCODINGS FOR PATTERNS AND DATALOG

In this section we introduce the formal definition of syntactic sugar. We show that a derived typing rule is admissible. We then prove correctness and completeness of the implementation of Datalog. The results of this section assume that we are using Datalog as the underlying authorization logic.

C.1 Syntactic Sugar

The syntactic sugar for input and decryption consists in a straightforward translation into the syntactic sugar for tuple matching. The definition of the latter is given by induction on the length of the tuple, by cases depending on whether the first parameter is used for binding or for matching.

SYNTACTIC SUGAR: INPUT, DECRYPTION AND PATTERN-MATCHING

$\textbf{in}\ M(\widetilde{\underline{M}});P = \textbf{in}\ M(y{:}\mathrm{Ty}_C(M));\textbf{tuple}\ y\ \textbf{as}\ (\widetilde{\underline{M}});P$ (S Input)
(where $y \notin fn(\widetilde{\underline{M}}) \cup fn(P)$)

$\textbf{decrypt}\ M\ \textbf{as}\ \{\widetilde{\underline{N}}\}N;P = \textbf{decrypt}\ M\ \textbf{as}\ \{y{:}\mathrm{Ty}_K(N)\}N;\textbf{tuple}\ y\ \textbf{as}\ (\widetilde{\underline{N}});P$ (S Decrypt)
(where $y \notin fn(\widetilde{\underline{M}}) \cup fn(P)$)

$\textbf{tuple}\ M\ \textbf{as}\ (z,\widetilde{\underline{M}});P = \textbf{split}\ M\ \textbf{as}\ (z{:}\mathrm{Ty}_L(M),y{:}\mathrm{Ty}_R(M));\textbf{tuple}\ y\ \textbf{as}\ (\widetilde{\underline{M}});P$ (S Split)
(where $y \notin fn(\widetilde{\underline{M}}) \cup fn(P) \cup \{z\}$)

$\textbf{tuple}\ M\ \textbf{as}\ (z);P = \textbf{split}\ (M,M)\ \textbf{as}\ (z{:}\mathrm{Ty}(M),y{:}\mathrm{Ty}(M));P$ (S Split 0)
(where $y \notin fn(P) \cup \{z\}$)

$\textbf{tuple}\ M\ \textbf{as}\ (=N,\widetilde{\underline{N}});P = \textbf{match}\ M\ \textbf{as}\ (N,y{:}\mathrm{Ty}_R(M));\textbf{tuple}\ y\ \textbf{as}\ (\widetilde{\underline{N}});P$ (S Match)
(where $y \notin fn(\widetilde{\underline{M}}) \cup fn(P)$)

$\textbf{tuple}\ M\ \textbf{as}\ (=N);P = \textbf{match}\ (M,M)\ \textbf{as}\ (N,y{:}\mathrm{Ty}(M));P$ (S Match 0)
(where $y \notin fn(P)$)

When an environment E is fixed, the macro $\mathrm{Ty}_{[C/K/L/R]}(M)$ can be translated to T

if $E \vdash M : T'$ where T' is respectively $T, \textbf{Ch}(T), \textbf{Key}(T), (x : T,U)$ or $(x : U,T)$.

In the encoding of Datalog each predicate of arity n corresponds to a channel of arity $n+1$ carrying a tuple of names of type **Un**, together with an **ok** token guaranteeing that the predicate holds for all the communication parameters. To simplify the typing of the encoding, we derive a dedicated typing rule for this very common case.

DERIVED TYPING RULE:

(ProcG Input Der)

$$\frac{E \vdash p : T_{n,p} \qquad E,\widetilde{u}{:}\widetilde{\textbf{Un}},y : \textbf{Ok}(p(u_1,\ldots,u_n)) \vdash P : G}{E \vdash \textbf{in}\ p(\underline{u}_1,\ldots,\underline{u}_n,=\textbf{ok});P : \textbf{0}}$$

where $\widetilde{u}$ are the u_i occurring as input patterns; $y \notin fn(P)$.

LEMMA 23. *Rule (ProcG Input Der) is admissible.*

PROOF. We show that if $E \vdash p : T_{n,p}$ and $E,\widetilde{u}{:}\widetilde{\textbf{Un}},y : \textbf{Ok}(p(u_1,\ldots,u_n)) \vdash P : G$, then $E \vdash \textbf{in}\ p(\underline{u}_1,\ldots,\underline{u}_n,=\textbf{ok});P : \textbf{0}$.
By (S Input), $\textbf{in}\ p(\underline{u}_1,\ldots,\underline{u}_n,=\textbf{ok});P$ is translated as
$\textbf{in}\ p(y{:}\mathrm{Ty}(p));\textbf{tuple}\ y\ \textbf{as}\ (\underline{u}_1,\ldots,\underline{u}_n,=\textbf{ok});P$.
By definition of encoding, $T_{n,p} = \textbf{Ch}(u_1{:}\textbf{Un},\ldots,u_n{:}\textbf{Un},\textbf{Ok}(p(u_1,\ldots,u_n)))$.
We can conclude by (ProcG Input) if we can show that
$E,y{:}(u_1{:}\textbf{Un},\ldots,u_n{:}\textbf{Un},\textbf{Ok}(p(u_1,\ldots,u_n))) \vdash \textbf{tuple}\ y\ \textbf{as}\ (\underline{u}_1,\ldots,\underline{u}_n,=\textbf{ok});P : \textbf{0}$.
We prove it by induction on the number of parameters left to parse i.

—($i = 0$): We need to show that $E, y{:}\mathbf{Ok}(p(u_1,\dots,u_n)) \vdash \mathbf{tuple}\ y\ \mathbf{as}\ (=\mathbf{ok}); P : \mathbf{0}$.
By (S Match 0), $\mathbf{tuple}\ y\ \mathbf{as}\ (=\mathbf{ok}); P = \mathbf{match}\ (y,y)\ \mathbf{as}\ (\mathbf{ok}, y : \mathrm{Ty}(y)); P$.
By hypothesis, $E, \widetilde{u}{:}\widetilde{\mathbf{Un}}, y : \mathbf{Ok}(p(u_1,\dots,u_n)) \vdash P : G$.
By (ProcG Match) and Lemma 16 we conclude.

—($i = j+1$): We need to show that $E, y{:}(u_{n-i+1}{:}\mathbf{Un},\dots,u_n{:}\mathbf{Un}, \mathbf{Ok}(p(u_1,\dots,u_n))) \vdash \mathbf{tuple}\ y\ \mathbf{as}\ (\underline{u}_{n-i+1},\dots,\underline{u}_n, =\mathbf{ok}); P : \mathbf{0}$.
We split the proof in two cases, depending on $\underline{u}_{n-i+1}$.

—($\underline{u}_{n-i+1} = u_{n-i+1}$): By (S Split), $\mathbf{tuple}\ y\ \mathbf{as}\ (\underline{u}_{n-i+1},\dots,\underline{u}_n, =\mathbf{ok}); P = \mathbf{split}\ y\ \mathbf{as}\ (u_{n-i+1}{:}\mathrm{Ty}_L(y), y{:}\mathrm{Ty}_R(y)); \mathbf{tuple}\ y\ \mathbf{as}\ (u_{n-j+1}\dots,\underline{u}_n, =\mathbf{ok}); P$.
By definition, $\mathrm{Ty}_R(y) = (u_{n-j+1}{:}\mathbf{Un},\dots,u_n{:}\mathbf{Un}, \mathbf{Ok}(p(u_1,\dots,u_n)))$ and $\mathrm{Ty}_L(y) = \mathbf{Un}$.
By (ProcG Split) and by the inductive hypothesis, we conclude.

—($\underline{u}_{n-i+1} = =u_{n-i+1}$): similar to the previous case, using (S Match) and (ProcG Match) instead of (S Split) and (ProcG Split). □

C.2 Correctness and Completeness

In this section we show that the encoding of Datalog is both correct and complete. It is correct in the sense that if we can derive a fact F in the encoding of a Datalog program S ($[\![S]\!] \Downarrow_F$), then the we can also derive it in the original program ($S \models F$). It is complete in the sense that if we can derive a fact in Datalog ($S \models F$) then we can also derive it in the encoding ($[\![S]\!] \Downarrow_F$).

PREDICATES OF A DATALOG PROGRAM: $\mathrm{pred}(S)$

$\mathrm{pred}(\varnothing) = \varnothing \quad \mathrm{pred}(\{C\} \cup S) = \mathrm{pred}(C) \cup \mathrm{pred}(S) \quad \mathrm{pred}(p(u_1,\dots,u_n)) = \{p_n\}$

$\mathrm{pred}(L_1,\dots,L_n) = \bigcup_{i \in 1..n} \mathrm{pred}(L_i) \quad \mathrm{pred}(L_0 :- \widetilde{L}) = \mathrm{pred}(L_0) \cup \mathrm{pred}(\widetilde{L})$

Notation: $\widetilde{L} = L_1,\dots,L_n$

EXTRACTING BINDINGS FROM LITERALS: $env^{\Sigma}(L_1,\dots,L_n)$

$env^{\Sigma \cup fv(L_{n-1})}(L_1,\dots,L_n) = env^{\Sigma}(L_1,\dots,L_{n-1}), env^{\Sigma \cup fv(L_{n-1})}(L_n)$
$env^{\Sigma}(p(u_1,\dots,u_n)) = env^{\Sigma}(u_1,\dots,u_n), y{:}\mathbf{Ok}(p(u_1,\dots,u_n))$ (where y is fresh)
$env^{\Sigma}(u_1,\dots,u_n) = env^{\Sigma}(u_1,\dots,u_{n-1}), env^{\Sigma \cup fv(u_1,\dots,u_{n-1})}(u_n)$
$env^{\Sigma}(X) = X{:}\mathbf{Un}$ if $X \notin \Sigma \quad env^{\Sigma}(X) = \varepsilon$ if $X \in \Sigma \quad env^{\Sigma}(M) = \varepsilon$

The next two lemmas show that any process obtained by encoding a Datalog program, in parallel with the clauses of the program itself is typable in an environment formed according to the rules of the encoding.

LEMMA 24. *Consider a clause $C = L :- L_m,\dots,L_1$, and let $\widetilde{p_n} = \mathrm{pred}(C)$ and $fn(C) \subseteq \{\widetilde{y}\}$. Let $E = \widetilde{y}{:}\widetilde{\mathbf{Un}}, \widetilde{p_n}{:}\widetilde{T_{n,p}}$. We have $E, C \vdash [\![C]\!] : \mathbf{0}$.*

PROOF. Let $\Sigma_m = \varnothing$ and $\Sigma_i = \Sigma_{i+1} \cup fv(L_{i+1})$. By induction on the number of literals i that remain to be considered, we show that

$$E, L :- L_m,\dots,L_1, env^{\Sigma_{i+1}}(L_m,\dots,L_{i+1}) \vdash [\![L_i,\dots,L_1]\!]^{\Sigma_i}[[\![L]\!]^+] : \mathbf{0}$$

—$i = 0$: $E, C, env^{\Sigma_1}(L_m,\dots,L_1) \vdash [\![L]\!]^+ : \mathbf{0}$ easily follows from (ProcG Output) and (Infer Fact).

—$i = j+1$: We are to show $E,C,env^{\Sigma_{i+1}}(L_m,\dots,L_{i+1}) \vdash [\![L_i,L_j,\dots,L_1]\!]^{\Sigma_i}[[\![L]\!]^+] : \mathbf{0}$.
Suppose, without loss of generality, that $L_i = p(u_1,\dots,u_h)$.
By definition of encoding,
$[\![L_i,L_j,\dots,L_1]\!]^{\Sigma_i}[[\![L]\!]^+] = \mathbf{in}\ p(\underline{u}_1,\dots,\underline{u}_h,{=}\mathbf{ok});[\![L_j,\dots,L_1]\!]^{\Sigma_i \cup fv(L_i)}[[\![L]\!]^+]$.
By definition of Σ_j, $\Sigma_j = \Sigma_i \cup fv(L_i)$.
By inductive hypothesis, $E,C,env^{\Sigma_i}(L_m,\dots,L_i) \vdash [\![L_j,\dots,L_1]\!]^{\Sigma_j}[[\![L]\!]^+] : \mathbf{0}$.
By (ProcG Input Der),
$E,C,env^{\Sigma_{i+1}}(L_m,\dots,L_{i+1}) \vdash \mathbf{in}\ p(\underline{u}_1,\dots,\underline{u}_h,{=}\mathbf{ok});[\![L_j,\dots,L_1]\!]^{\Sigma_j}[[\![L]\!]^+] : \mathbf{0}$.

By definition of encoding and by (ProcG Rep) we conclude. □

RESTATEMENT OF LEMMA 3. *Let S be a Datalog program using predicates $\widetilde{p_n}$ and names $\widetilde{y}$ with $fn(S) \subseteq \{\widetilde{y}\}$. Let $E = \widetilde{y{:}\mathbf{Un}}, \widetilde{p_n{:}T_{n,p}}$. We have $E \vdash S \mid [\![S]\!]$.*

PROOF. By induction on the structure of S.

—($S = \varnothing$): We conclude with $\varnothing \vdash \mathbf{0}$.
—($S = S' \cup \{C\}$): By definition of encoding, we need to show that $E \vdash S' \mid [\![S']\!] \mid C \mid [\![C]\!]$.
By inductive hypothesis and weakening, we have $E,C \vdash S' \mid [\![S']\!]$.
By Lemma 24 and weakening, $E,S',C \vdash [\![C]\!] : \mathbf{0}$.
By (ProcG Fact) and weakening, $E,S' \vdash C : C$.
By (ProcG Par), $E,S' \vdash C \mid [\![C]\!] : C$.
By (ProcG Par) and weakening, we conclude. □

LEMMA 25. *Let $L = p(u_1,\dots,u_n)$ be a Datalog literal, let σ,ρ be substitutions (with disjoint domains) of messages for Datalog variables, and let Σ be a set of Datalog variables such that $dom(\sigma) = \Sigma$. Then, $([\![L]\!]^{\Sigma}[P])\sigma \mid [\![L\sigma\rho]\!]^+ \rightarrow^{n+1} P\sigma\rho$.*

PROOF. By induction on the arity n of the predicate p and by definition of syntactic sugar, following the structure of the proof of Lemma 23. □

LEMMA 26. *Let $C = L_0 :- L_1,\dots,L_n$ be a Datalog clause, and let σ be a substitution of messages for Datalog variables such that all the $L_i\sigma$ are ground facts. There exists a process P such that $[\![C]\!] \mid [\![L_1\sigma]\!]^+ \mid \dots \mid [\![L_n\sigma]\!]^+ \rightarrow^*_{\equiv} P \mid ([\![L_0]\!]^+)\sigma$.*

PROOF. By definition of encoding,
$[\![C]\!] \mid [\![L_1\sigma]\!]^+ \mid \dots \mid [\![L_n\sigma]\!]^+ \equiv [\![C]\!] \mid [\![L_1,\dots,L_n]\!]^{\varnothing}[[\![L_0]\!]^+] \mid [\![L_1\sigma]\!]^+ \mid \dots \mid [\![L_n\sigma]\!]^+$. We show, by induction on n, that $([\![L_1,\dots,L_n]\!]^{\Sigma}[[\![L_0]\!]^+])\sigma \mid [\![L_1\sigma\rho]\!]^+ \mid \dots \mid [\![L_n\sigma\rho]\!]^+ \rightarrow^*_{\equiv} [\![L_0]\!]^+\sigma\rho$ where $dom(\sigma) = \Sigma$, which implies the thesis.

—($n = 0$): By hypothesis, C is a ground fact.
By definition of encoding, $([\![\varepsilon]\!]^{\Sigma}[C])\sigma = ([\![C]\!]^+)\sigma$ and we conclude, with $\rho = \varnothing$.
—($n = m+1$): Suppose, without loss of generality, that $L_{m+1} = p(u_1,\dots,u_h)$.
By definition of encoding, $[\![L_{m+1},L_1,\dots,L_m]\!]^{\Sigma}[[\![L_0]\!]^+] = Q$ where
$Q = \mathbf{in}\ p(\underline{u}_1,\dots,\underline{u}_h,{=}\mathbf{ok});[\![L_1,\dots,L_m]\!]^{\Sigma \cup fv(L_{m+1})}[[\![L_0]\!]^+]$.
By Lemma 25, $Q\sigma \mid [\![L_1\sigma\rho]\!]^+ \mid \dots \mid [\![L_m\sigma\rho]\!]^+ \mid [\![L_{m+1}\sigma\rho]\!]^+ \rightarrow^{h+1}$
$([\![L_1,\dots,L_m]\!]^{\Sigma \cup fv(L_{m+1})}[[\![L_0]\!]^+])\sigma\rho \mid [\![L_1\sigma\rho]\!]^+ \mid \dots \mid [\![L_m\sigma\rho]\!]^+$,
where $dom(\rho) = fv(L_{m+1})$.
By inductive hypothesis,
$([\![L_1,\dots,L_m]\!]^{\Sigma \cup fv(L_{m+1})}[[\![L_0]\!]^+])\sigma\rho \mid [\![L_1\sigma\rho]\!]^+ \mid \dots \mid [\![L_m\sigma\rho]\!]^+ \rightarrow^*_{\equiv} [\![L_0]\!]^+\sigma\rho$. □

The lemma below shows that an encoded program is not consumed by reductions.

LEMMA 27. *If* $[\![S]\!] \rightarrow^*_{\equiv} P$ *then there exists* P' *such that* $P \equiv [\![S]\!] \mid P'$.

PROOF. By definition of encoding, structural congruence and reduction. □

Finally, we can show correctness and completeness for the encoding.

RESTATEMENT OF THEOREM 4. *Let* S *be a Datalog program and* F *a fact. We have* $S \models F$ *if and only if* $[\![S]\!] \Downarrow_F$.

PROOF. ($\Rightarrow$) By induction on the depth of the derivation tree for $S \models F$. The base case is by definition of encoding and by definition of $\Downarrow$. The inductive case follows by Lemma 27, Lemma 26 and by definition of $\Downarrow$.

($\Leftarrow$) By Lemma 3, there exists a generative environment E such that $E \vdash S \mid [\![S]\!]$.
By definition of $\Downarrow$, $\exists P. [\![S]\!] \rightarrow^*_{\equiv} P \mid [\![F]\!]^+$.
By Lemma 1, $E \vdash S \mid P \mid [\![F]\!]^+$. By reasoning on the typing rules, since $[\![S]\!]$ contains no statements and the subterm $[\![F]\!]^+$ is well-typed, it must be the case that $S \models F$. □

REFERENCES

ABADI, M. 1998. On SDSI's linked local name spaces. *Journal of Computer Security 6,* 1–2, 3–21.

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

ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. 1993. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems 15,* 4, 706–734.

ABADI, M. AND GORDON, A. D. 1999. A calculus for cryptographic protocols: The spi calculus. *Information and Computation 148*, 1–70.

BECKER, M. Y. AND SEWELL, P. 2004. Cassandra: flexible trust management, applied to electronic health records. In *17th IEEE Computer Security Foundations Workshop (CSFW'04)*. 139–154.

BLANCHET, B. 2002. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)*. Lecture Notes in Computer Science, vol. 2477. Springer, 342–359.

BLAZE, M., FEIGENBAUM, J., AND LACY, J. 1996. Decentralized trust management. In *IEEE 17th Symposium on Research in Security and Privacy*. 164–173.

BRAGHIN, C., GORLA, D., AND SASSONE, V. 2004. A distributed calculus for role-based access control. In *17th IEEE Computer Security Foundations Workshop (CSFW'04)*. 48–60.

BUGLIESI, M., CASTAGNA, G., AND CRAFA, S. 2004. Access control for mobile agents: the calculus of boxed ambients. *ACM Transactions on Programming Languages and Systems 26,* 1 (Jan.), 57–124.

BUGLIESI, M., COLAZZO, D., AND CRAFA, S. 2004. Type based discretionary access control. In *CONCUR'04—Concurrency Theory*. Lecture Notes in Computer Science, vol. 3170. Springer, 225–239.

CERI, S., GOTTLOB, G., AND TANCA, L. 1989. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering 1,* 1, 146–166.

CONTENTGUARD. 2002. XrML 2.0 Technical Overview. `http://www.xrml.org/`.

DE NICOLA, R., FERRARI, G., AND PUGLIESE, R. 2000. Programming access control: The KLAIM experience. In *CONCUR 2000—Concurrency Theory*. Lecture Notes in Computer Science, vol. 1877. Springer, 48–65.

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

DOLEV, D. AND YAO, A. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory IT–29,* 2, 198–208.

DUGGAN, D. 2002. Cryptographic types. In *15th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 238–252.

FOURNET, C., GORDON, A. D., AND MAFFEIS, S. 2005a. A type discipline for authorization policies. Tech. Rep. MSR–TR–2005–01, Microsoft Research.

FOURNET, C., GORDON, A. D., AND MAFFEIS, S. 2005b. A type discipline for authorization policies. In *14th European Symposium on Programming (ESOP'05)*. Lecture Notes in Computer Science, vol. 3444. Springer, 141–156.

GORDON, A. D. AND JEFFREY, A. 2002a. Cryptyc: Cryptographic protocol type checker. At *http://cryptyc.cs.depaul.edu/*.

GORDON, A. D. AND JEFFREY, A. 2002b. Typing one-to-one and one-to-many correspondences in security protocols. In *Software Security—Theories and Systems*. Lecture Notes in Computer Science, vol. 2609. Springer, 270–282.

GORDON, A. D. AND JEFFREY, A. 2003a. Authenticity by typing for security protocols. *Journal of Computer Security 11,* 4, 451–521.

GORDON, A. D. AND JEFFREY, A. 2003b. Typing correspondence assertions for communication protocols. *Theoretical Computer Science 300*, 379–409.

GORDON, A. D. AND JEFFREY, A. 2005. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *CONCUR 2005—Concurrency Theory*. Lecture Notes in Computer Science, vol. 3653. Springer, 186–201.

GUELEV, D. P., RYAN, M. D., AND SCHOBBENS, P.-Y. 2004. Model-checking access control policies. In *Seventh Information Security Conference (ISC04)*. Lecture Notes in Computer Science, vol. 3225. Springer.

GUTTMAN, J. D., THAYER, F. J., CARLSON, J. A., HERZOG, J. C., RAMSDELL, J. D., AND SNIFFEN, B. T. 2004. Trust management in strand spaces: a rely-guarantee method. In *13th European Symposium on Programming (ESOP'04)*. Lecture Notes in Computer Science, vol. 2986. Springer, 340–354.

JIM, T. 2001. SD3: a trust management system with certified evaluation. In *IEEE Computer Society Symposium on Research in Security and Privacy*. 106–115.

JONES, A. K. AND LISKOV, B. H. 1978. A language extension for expressing constraints on data access. *Commun. ACM 21,* 5, 358–367.

LAMPSON, B. 1971. Protection. In *Proc. 5th Princeton Conference on Information Sciences and Systems*. 437–443. Reprinted in ACM Operating Systems Review, 8(1)18–24, 1974.

LI, N. AND MITCHELL, J. C. 2003. Understanding SPKI/SDSI using first-order logic. In *16th IEEE Computer Security Foundation Workshop (CSFW'03)*. 89–103.

MAFFEI, M. 2006. Dynamic typing for security protocols. Ph.D. thesis, Università Ca' Foscari Venezia.

MARTIN-LÖF, P. 1984. *Intuitionistic Type Theory*. Bibliopolis.

MILNER, R. 1999. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press.

MYERS, A. C. AND LISKOV, B. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology 9,* 4, 410–442.

PIERCE, B. AND SANGIORGI, D. 1996. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science 6,* 5, 409–454.

POTTIER, F. 2002. A simple view of type-secure information flow in the π-calculus. In *15th IEEE Computer Security Foundations Workshop (CSFW'02)*. IEEE Computer Society Press, 320–330.

SAGIV, Y. 1987. Optimizing Datalog programs. In *Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, 349–362.

SAMARATI, P. AND DE CAPITANI DI VIMERCATI, S. 2001. Access control: Policies, models, and mechanisms. In *FOSAD 2000*. Lecture Notes in Computer Science, vol. 2171. Springer, 137–196.

WOO, T. AND LAM, S. 1993. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*. 178–194.