A Type Discipline for Authorization Policies

Cédric Fournet[1], Andrew D. Gordon[1], and Sergio Maffeis[1,2]

[1] Microsoft Research
[2] Department of Computing, Imperial College London

Abstract. 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.
For a given authorization policy, we consider the problem of checking whether a cryptographic implementation complies with the policy. We formalize authorization policies by embedding logical predicates and queries within a spi calculus. This embedding is new, simple, and general; it allows us to treat logic programs as specifications of code using secure channels, cryptography, or a combination. Moreover, we propose a new dependent type system for verifying such implementations against their policies. Using Datalog as an authorization logic, we show how to type several examples using policies and present a general schema for compiling policies.

1 Typing Implementations of Authorization Policies

An *authorization policy* prescribes conditions that must be satisfied before performing any privileged action (for example, accessing a sensitive resource). A system complies with the policy if these conditions hold whenever the action is performed—however, the policy does not usually prescribe a particular choice of enforcement mechanisms.

Authorization issues can be complex, even at an abstract level. Some policies address security concerns for numerous actors, involving roles, groups, partial trust, and controlled delegation. Those policies are best expressed in high-level languages, with supporting tools. Specifically, logic programming seems well suited for expressing policies: each authorization request is formulated as a logical request against a database of facts and rules, while the policy itself carefully controls changes to the database. Hence, variants of Datalog have been usefully applied to design trust management systems (e.g., PolicyMaker [6], SD3 [20], Binder [12]), express complex policies (e.g., Cassandra [4]), and study authorization languages (e.g., SDSI/SPKI [1,21], XrML [11]).

Given a target policy, we consider the problem of verifying that a particular system correctly implements this 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. Authorization decisions are often intermingled with other imperative code, and are hard to analyze and audit. For instance, the request may rightfully appear in many places in the code, most of them

To appear in the proceedings of the 14th European Symposium on Programming.
S. Sagiv (Ed.): ESOP 2005, LNCS.

without a valid identity certificate at hand. The relation between imperative code and declarative policies is usually informal: theoretical studies rarely connect the logic to an operational semantics.

Our formal development is within a spi calculus [3], that is, a pi calculus with abstract cryptographic operations. We use a global policy, interpreted against processes in a way that generalizes a previous embedding [17] of correspondence assertions [24]. There are many techniques to verify standard correspondences with respect to the Dolev-Yao model [13], the standard "network is the opponent" threat model for cryptographic protocols. However, these correspondences are attached to low-level events (such as a successful decryption), and it can be quite hard to relate them to high-level access control decisions. Perhaps in consequence, more abstract correspondences have seldom been validated against the Dolev-Yao model, even though they rely on cryptography.

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. The static approach is less expressive 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 expressive 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.

Summary of Contributions To our knowledge, this is the first attempt to relate authorization logics to their cryptographic implementation in a pi calculus. Specifically:

- 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 queries. Our main theorem states that all queries 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 featuring 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, more theoretical application, we present a generic implementation of Datalog in the pi calculus—well-typed in our system—which can be used 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).

Related Work There is a substantial literature on type systems for checking security properties. In the context of process calculi, there are, for example type systems to check secrecy [2] and authenticity [16] properties in the spi calculus, access control properties of mobile code in the boxed ambient calculus [8], and discretionary access control [9] and role-based access control [7] in the pi calculus. Moreover, various experimental systems, such as JIF [22] and KLAIM [23], 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.* [19] 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. More technically, we attach static authorization effects to any operation (input, decryption, matching) rather than just message inputs.

Blanchet's ProVerif [5] 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 authorization policies.

Guelev *e*t al. [18] also adopt a conference programme committee as a running example, in the context of model checking the consequences of access control policies.

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 provides our pi calculus implementation of Datalog and states its correctness and completeness. Section 7 concludes and sketches future work. Due to space constraints, some standard definitions and all proofs are omitted; they appear in a technical report [14].

2 A Simple Logic for Authorization

Datalog We briefly present a syntax and semantics for Datalog. (For a comprehensive survey of Datalog, see for instance [10].) 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 programs as authorization policies.

Syntax for Datalog:

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

A literal L is a predicate $p(u_1,\ldots,u_n)$, of fixed arity $n \geq 0$, on terms $u_1,\ldots,u_n$. Terms range over logical 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, 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, $L :- $. We often write the (ground) literal L as an abbreviation of the fact $L :- $.

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 u for 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 $\{M/n\}$ for capture-avoiding substitution of message M for name n.

A fact can be derived from a Datalog program using the rule below:

Logical Inference: $S \models F$

(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, also 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. We generalize inference to clauses accordingly:

Logical Inference for Clauses (Entailment): $S \models C$

(Infer Clause)

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

Example 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 wrote 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 single report to be filed. Preferably, 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 committee; 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:

Report(U,ID,R):−Referee(U,ID),Opinion(U,ID,R) (clause A)
Report(U,ID,R):−PCMember(U),Opinion(U,ID,R) (clause B)

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:

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

Delegation 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:

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

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

Discussion In contrast to more sophisticated authorization languages, which associates facts with principals "saying" them, we adopt the subjective viewpoint of the conference system, which implicitly owns all predicates used to control reports. 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.

From Datalog to Arbitrary Authorization Logics 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 requirements listed below:

Definition 1. *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) $S \models C \Rightarrow S\sigma \models C\sigma$.

3 A Spi Calculus with Authorization Assertions

The spi calculus [3] extends the pi calculus with abstract cryptographic operations in the style of Dolev and Yao [13]. 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 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: statements and expectations. These processes generalize the begin- and end-assertions in previous embeddings of correspondences in process calculi [17]. Similarly, they track security properties, but 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 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 variables instantiated at runtime. In the following, the contents 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)

(The distinguished name **ok** is an annotation to help typing, with no effect at runtime.)

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, all the expectations in our case studies are of facts, not rules.

The syntax and the operational semantics of our full calculus appear on the next page. 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 [16]. Binding occurrences of names have type annotations, T or U; the syntax of our system of dependent types is in Section 4.

The operational semantics is defined as a reduction relation, with standard rules. Statements and expectations are inert processes; they do not have particular rules for reduction or congruence (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.

In examples, we rely on derived notations for n-ary tuples, and for pattern-matching tuples via sequences of match and split operations. For $n > 2$, $(M_1, M_2, \ldots, M_n)$ abbreviates $(M_1, (M_2, \ldots, M_n))$. We write our process notation for pattern-matching tuples in the form **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

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 name
$P,Q,R ::=$	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)
$!P$	unbounded parallel composition of replicas of P
$P \mid Q$	parallel composition of P and Q
$\mathbf{0}$	inactivity
decrypt L **as** $\{y{:}T\}N;P$	bind y to decryption of L 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 $C_1 \mid \ldots \mid C_n \mid P$.

Rules for Reduction: $P \to P'$

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

Structural equivalence $P \equiv Q$ is defined as usual, and $P \to^*_{\equiv} P'$ is $P \equiv P'$ or $P \to^* P'$.

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. We allow pattern-matching in conjunction with input and decryption processes, and omit type annotations. The technical report has the formal details of these notations.

The presence of statements and expectations in a process induces the following safety properties. Informally, to say an expectation **expect** C is *justified* means there are sufficient statements in parallel to derive C. Then a process is safe if every expectation in every reachable process is justified.

Definition 2 (Safety). *A process P is* safe *iff whenever $P \to^*_{\equiv}$* **new** $\tilde{x}{:}\tilde{T};($**expect** $C \mid P')$ *then $P' \equiv$* **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, …, C_n all use the same names; the scopes of these names are otherwise irrelevant in the logic.)

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.

Definition 3 (Opponent). *A process O is an* opponent *iff it contains no expectations, and every type annotation is* **Un**.

Definition 4 (Robust Safety). *A process P is* robustly safe *iff $P \mid O$ is safe for all opponents O.*

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:

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

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 new dependent type system for checking implementations of authorization policies. Our starting point for this development was a type and effect system by Gordon and Jeffrey [15] for verifying one-to-many correspondences. Apart from the new support for logical assertions, the current system features two improvements. First, a new rely-guarantee rule for parallel composition allows us to typecheck a safe process such as $L \mid \mathbf{expect}\ L$; 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. We begin by defining the syntax and informal semantics of message types.

Syntax for Types:

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

T is *generative* (may be freshly created) iff T is either **Un**, **Key**(U), or **Ch**(U).

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

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 $\mathbf{Ch}(T)$ is a name used as a secure channel for messages of type T. Similarly, a message of type $\mathbf{Key}(T)$ is a name used as a secret key for encrypting and decrypting plaintexts of type T. A message of type $(x{:}T, U)$ is a pair (M, N) where M is of type T, and N is of type $U\{M/x\}$. Finally, the token **ok** is the unique message of type $\mathbf{Ok}(S)$, proving S may currently be inferred.

For example, the type $\mathbf{Ch}((x{:}\mathbf{Un}, \mathbf{Ok}(\mathrm{Report}(\mathrm{alice}, 42, x))))$ 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 $\mathbf{ok} : \mathbf{Ok}(\mathrm{Report}(\mathrm{alice}, 42, M))$.

Next, we define typing environments—lists of variable bindings and clauses—plus two auxiliary functions. 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 $env(P)$ and $clauses(E)$:

$E ::= \varnothing \mid E, x{:}T \mid E, C$ Notation: $E(x) = T$ if $E = E', x{:}T, E''$

E is *generative* iff $E = x_1{:}T_1, \ldots, x_n{:}T_n$ and each T_i is generative.

$env(P \mid Q)^{\widetilde{x},\widetilde{y}} = env(P)^{\widetilde{x}}, env(Q)^{\widetilde{y}}$ (where $\{\widetilde{x},\widetilde{y}\} \cap fn(P \mid Q) = \varnothing$)
$env(\mathbf{new}\ x{:}T; P)^{x,\widetilde{x}} = x{:}T, env(P)^{\widetilde{x}}$ (where $\{\widetilde{x}\} \cap fn(P) = \varnothing$)
$env(!P)^{\widetilde{x}} = env(P)^{\widetilde{x}}$ $env(C)^{\varnothing} = C$ $env(P)^{\varnothing} = \varnothing$ otherwise
Convention: $env(P) \triangleq env(P)^{\widetilde{x}}$ for some distinct $\widetilde{x}$ such that $env(P)^{\widetilde{x}}$ is defined.

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

Our system consists of three judgments, defined by the following tables. The judgments define well-formed environments, types of messages, and well-formed processes.

Rules for Environments and Messages: $E \vdash \diamond$, $E \vdash M : T$

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

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

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

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

The rule (Msg Ok) populates an $\mathbf{Ok}(S)$ type only if we can infer each clause in the Datalog program S from the clauses in E. For example, if

$$E = \mathrm{alice}{:}\mathbf{Un}, 42{:}\mathbf{Un}, \mathrm{report42}{:}\mathbf{Un}, \mathrm{Referee}(\mathrm{alice},42), \mathrm{Opinion}(\mathrm{alice},42,\mathrm{report42})$$

then $E \vdash \mathbf{ok} : \mathbf{Ok}(\mathrm{Report}(\mathrm{alice}, 42, \mathrm{report42}))$. The other message typing rules are fairly standard. As in previous systems [16,15], we need the rules (Msg Encrypt Un), (Msg Pair Un), and (Msg Ok Un) to assign **Un** to arbitrary messages known to the opponent.

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

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

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

(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 C is entailed in the current environment. (Proc Fact) allows any statement, provided its names are in scope. (Proc Par) is a rely-guarantee 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), $\varnothing \vdash \mathrm{Foo}() \mid$ **expect** $\mathrm{Foo}()$ follows from $\varnothing \vdash \mathrm{Foo}()$ and $\mathrm{Foo}() \vdash$ **expect** $\mathrm{Foo}()$, the two of which follow directly by (Proc Fact) and (Proc Expect).

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 (Type Preservation). *If $E \vdash P$ and either $P \equiv P'$ or $P \to P'$ then $E \vdash P'$.*

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

Our second theorem is that well-typed processes whose free names 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 (Opponent Typability). *If $fn(O) \subseteq \{\widetilde{x}\}$ for opponent O then $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash O$.*

Theorem 2 (Robust Safety). *If $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash P$ then P is robustly safe.*

We conclude this section by showing 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.

We can encode one particular syntax [15] as follows:

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

With this encoding and a minor extension to the type system (tagged union types), we can express and typecheck all of the authentication protocols from Gordon and Jeffrey's paper [15], including WMF and BAN Kerberos.

The correspondences expressible by standard begin- and end-assertions 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.

5 Application: Access Control for a Programme Committee

We provide two 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, 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.

Online Delegation, with Local State. Our first implementation assumes the conference system is contacted whenever a referee decides to delegate her task. Hence, the system keeps track of expected reports using a local database, each record showing a fact of the form Referee(U,ID). When a report is received, the authenticated sender of the report is correlated with her record. When a delegation request is received, the corresponding record is 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 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,{w,id,ok}kdv) ))) |
```

Two new keys are first generated. The replicated output on pwdb associates those keys with v. The replicated input on sendreportonline guards a process that files v's reports; in this process, the encryption {id,report,**ok**}krv protects the report and also carries the 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 message on refereedb.) In the process 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 (e.g., 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,w,id,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,w,id,sigd);
     in pwdb(=v,kdv,krv); decrypt sigd as {=w,=id,_}kdv;
     in refereedb(=v,=id,_); out refereedb(w,id,ok)) |
```

Reviews from PC members, using Capabilities. 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 also implement 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.

Offline Delegation, with Certificate Chains. The 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.

To this end, the process guarded by the replicated input on channel filedelegatereport allocates a private channel link and uses that channel recursively to verify each piece of evidence filed with the report, one certificate at a time. 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). Note that 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 report writer) up to v (the current delegator) for paper id.

A further, less important difference is that our second implementation relies on self-authenticated capabilities under key ka for representing initial refereeing requests, instead of messages on the private database channel refereedb. Finally, our second implementation relies on auxiliary clauses making 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.

```
( 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,skt,ct);
    in pwdb(=t,kdt,_); decrypt skt as {=u,=id,_}kdt; out link(t,ct,ok)) |
```

Proposition 1. *Let E_P assign the types displayed above to pwdb, refereedb, kp, and ka. Let E_{Un} assign type **Un** to createReviewer, createPCMember, sendreportonline, delegateonline, filereport, filedelegate, filepcreport, filedelegatereport, and any other variable in its domain.*

Let P be a process such that $E_{Un}, E_P \vdash P$. Let Q be the process consisting of 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 2. 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, notably the attacker, but also additional code for the reviewers interacting with Q (and possibly P) using the names collected in E_{Un}, and notably the free channels 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.

We finally 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 type $T_{p,n}$. Unless the predicate p occurs with different arities, we omit indices and write 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: $[\![S]\!]$

$T_{p,n} = \mathbf{Ch}(x_1{:}\mathbf{Un}, \ldots, x_n{:}\mathbf{Un}, \mathbf{Ok}(p(x_1, \ldots, x_n)))$
$[\![S]\!] = \prod_{C \in S} [\![C]\!] \qquad [\![\varnothing]\!] = \mathbf{0} \qquad [\![L :- L_1, \ldots, L_m]\!] = \,!\,[\![L_1, \ldots, L_m]\!]^{\varnothing}[[\![L]\!]^+] \quad \text{for } m \geq 0$
$[\![p(u_1, \ldots, u_n)]\!]^+ = \mathbf{out}\ p_n(u_1, \ldots, u_n, \mathbf{ok})$
$[\![L_1, L_2, \ldots, L_m]\!]^{\Sigma}[\cdot] = [\![L_1]\!]^{\Sigma}\left[[\![L_2, \ldots, L_m]\!]^{\Sigma \cup fv(L_1)}[\cdot]\right] \qquad [\![\varepsilon]\!]^{\Sigma}[\cdot] = [\cdot]$
$[\![p(u_1, \ldots, u_n)]\!]^{\Sigma}[\cdot] = \mathbf{in}\ p_n(\underline{u}_1, \ldots, \underline{u}_n, = \mathbf{ok}); [\cdot]$
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 $\exists P'.P \to_{\equiv}^* P' \mid [\![L]\!]^+$

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

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

The next lemma states that the translation of a Datalog program is well typed when placed in parallel with itself as a policy.

Lemma 3 (Typability of the encoding). *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 3 (Correctness and completeness). *Let S be a Datalog program and F a fact. We have $S \models F$ if and only if $[\![S]\!] \Downarrow_F$.*

Example To illustrate our translation, we sketch an alternative implementation of our conference management server. Instead of coding the recursive processing of messages sent by subreferees, 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 **expect** processes 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.

Future Work From a logical viewpoint, many authorization languages extend Datalog with notions of locality and partial trust, considering for examples facts and clauses relative to each principal. Similarly, many variants of the pi calculus feature explicit localities and principals and could, in principle, provide a more realistic distributed semantics for these logics. 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 and the anonymous conference reviewers made useful suggestions.

References

1. M. Abadi. On SDSI's linked local name spaces. *J. Computer Security*, 6(1–2):3–21, 1998.
2. M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, Sept. 1999.
3. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.
4. M. Y. Becker and P. Sewell. Cassandra: flexible trust management, applied to electronic health records. In *17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 139–154, June 2004.
5. B. Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 342–359. Springer, 2002.
6. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE 17th Symposium on Research in Security and Privacy*, pages 164–173, 1996.
7. C. Braghin, D. Gorla, and V. Sassone. A distributed calculus for role-based access control. In *17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 48–60, June 2004.
8. M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: the calculus of boxed ambients. *ACM TOPLAS*, 26(1):57–124, Jan. 2004.
9. M. Bugliesi, D. Colazzo, and S. Crafa. Type based discretionary access control. In *CONCUR'04 - Concurrency Theory*, volume 3170 of *LNCS*, pages 225–239. Springer, Sept. 2004.
10. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
11. ContentGuard. XrML 2.0 Technical Overview. `http://www.xrml.org/`, Mar. 2002.
12. J. DeTreville. Binder, a logic-based security language. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 105–113, 2002.
13. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.
14. C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. Technical Report MSR–TR–2005–01, Microsoft Research, 2005.
15. A. D. Gordon and A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Software Security—Theories and Systems*, volume 2609 of *LNCS*, pages 270–282. Springer, 2002.
16. A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *J. Computer Security*, 11(4):451–521, 2003.
17. A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Comput. Sci.*, 300:379–409, 2003.
18. D. P. Guelev, M. D. Ryan, and P.-Y. Schobbens. Model-checking access control policies. In *Seventh Information Security Conference (ISC04)*, volume 3225 of *LNCS*. Springer, 2004.
19. J. D. Guttman, F. J. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Trust management in strand spaces: a rely-guarantee method. In *13th European Symposium on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 340–354. Springer, 2004.
20. T. Jim. SD3: a trust management system with certified evaluation. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 106–115, 2001.
21. N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *Proceedings of the 16th IEEE Computer Security Foundation Workshop (CSFW'03)*, pages 89–103, 2003.
22. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
23. R. D. Nicola, G. Ferrari, and R. Pugliese. Programming access control: The KLAIM experience. In *CONCUR 2000*, volume 1877 of *LNCS*, pages 48–65. Springer, 2000.
24. T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.