A Type Discipline for Authorization Policies

Cédric Fournet Andrew D. Gordon
Sergio Maffeis

September 22, 2005

Technical Report
MSR–TR–2005–01

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

Publication History

A portion of this work appears in the proceedings of *ESOP 2005: European Symposium on Programming (ESOP 2005)*, Edinburgh, April 6–8, 2005.

Affiliation

Sergio Maffeis is with Imperial College London. Most of this work was completed by the three authors while at Microsoft Research in Cambridge.

A Type Discipline for Authorization Policies

Cédric Fournet Andrew D. Gordon Sergio Maffeis

September 22, 2005

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.

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 new 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.

Contents

1 Typing Implementations of Authorization Policies

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 [22, 28].

Authorization policies are often only expressed precisely in code, intermingled with other functions and low-level enforcement mechanisms such as cryptography or system calls. The result can be hard to analyze and audit. Hence, a reasonable guiding principle 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 (e.g., PolicyMaker [6], SD3 [21], Binder [12]), to express complex policies (e.g., Cassandra [4]), and to study authorization languages (e.g., SDSI/SPKI [1, 23], XrML [11]).

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 used to specify 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 [13]. 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 man-in-the-middle or impersonation attacks that often afflict cryptographic protocols.

Our formal development is within a typed version of the spi calculus [3], 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

  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

  canRead(X,handbook) :– 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 following expectation records that canRead(alice,handbook) should be entailed in the current context.

  expect canRead(alice,handbook)

  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 [17] of Woo and Lam's correspondences [29] 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. 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.

Summary of Contributions To our knowledge, our work is the first to relate authorization logics to their cryptographic implementation in a process 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 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 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 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 various information flow [2, 18, 26] and authenticity [14, 16] properties in the pi calculus and 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 [24] and KLAIM [25], 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.* [20] 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.

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 clausally-defined authorization policies.

Guelev *et al.* [19] 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.

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. Appendix D lists the code of the example from Section 5 in the form accepted by our typechecker.

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 [10].) 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 we rely on.

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

M	spi calculus message (see Section 3)
$L ::=$	literal
$\quad p(u_1,\ldots,u_n)$	predicate p holds for terms $u_1,\ldots,u_n$
$C ::=$	Horn clause
$\quad L :- L_1,\ldots,L_n$	clause, with $n \geq 0$ and $fv(L) \subseteq \bigcup_i fv(L_i)$
$S ::=$	Datalog program (or policy)
$\quad \{C_1,\ldots,C_n\}$	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, 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 capture-avoiding substitution of message M for name x.

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 [27] and can be checked operationally using the *chase* technique.

Theorem 1 (Chase [27]) *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 (Monotonicity) *If* $S \models C$ *then* $S \cup \{C'\} \models C$.

Proposition 2 (Substitutivity) *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. 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 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:

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:

Report(U,ID,R):−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:

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

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

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 associate facts with principals "saying" 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 requirements listed below:

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 [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 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 [17]. Similarly, statements and expectations 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 2.3 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)

(The distinguished message **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 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 [16].

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)
$P \equiv P' \Rightarrow$ **new** $x{:}T;P \equiv$ **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)
new $x{:}T;(P \mid Q) \equiv P \mid$ **new** $x{:}T;Q$	(Struct Res Par) (for $x \notin fn(P)$)
new $x_1{:}T_1;$**new** $x_2{:}T_2;P \equiv$ **new** $x_2{:}T_2;$**new** $x_1{:}T_1;P$	(Struct Res Res) (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)
$P \rightarrow P' \Rightarrow$ **new** $x{:}T;P \rightarrow$ **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)

Notation: $P \to_{\equiv}^{*} P'$ is $P \equiv P'$ or $P \to^{*} P'$.

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 $\underline{N}_1, \ldots, \underline{N}_n, P$.

$$\begin{array}{rcl} \textbf{in}\ M(\underline{N}_1, \ldots, \underline{N}_n); P & \triangleq & \textbf{in}\ M(y); \textbf{tuple}\ y\ \textbf{as}\ (\underline{N}_1, \ldots, \underline{N}_n); P \\ \textbf{decrypt}\ M\ \textbf{as}\ \{\underline{N}_1, \ldots, \underline{N}_n\}N; P & \triangleq & \textbf{decrypt}\ M\ \textbf{as}\ \{y\}N; \textbf{tuple}\ y\ \textbf{as}\ (\underline{N}_1, \ldots, \underline{N}_n); P \end{array}$$

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 statements in parallel to derive 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

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

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, …, 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 following process would be judged safe:

new x; **expect** Foo(x)

Conversely, were the definition to omit the intermediate restricted names $\tilde{y}$, the following process would be judged unsafe:

Bar():−Foo(X) | **expect** Bar() | **new** y; Foo(y)

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:

R | **out** c (bogus,**ok**) → P | **out** c (report42,**ok**) | **expect** 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 new dependent type system for checking implementations of authorization policies. Our starting point for this development is 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 $L \mid$ **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
Un	public data
Ch(T)	channel for messages of type T
Key(T)	secret key for plaintexts of type T
$(x{:}T,U)$	dependent pair (scope of x is U)
Ok(S)	ok to assume the clauses S

T is *generative* (may be freshly created) if and only if T is **Un**, **Ch**(U), or **Key**(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 **Ch**(T) is a name used as a secure channel for messages of type T. Similarly, a message of type **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 **Ok**(S), proving S may currently be inferred.

For example, the type **Ch**$((x{:}$**Un**$,$**Ok**$(\mathrm{Report}(\mathrm{alice},42,x))))$ can be assigned to c in process R, stating that c is a channel for communicating pairs $(M,$**ok**$)$ where M : **Un** and **ok** : **Ok**$(\mathrm{Report}(\mathrm{alice},42,M))$.

Next, we define typing environments—lists of name 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 **Ok**$(-)$ types.

Syntax for Environments, and Functions: $dom(E)$**,** $env(P)$**, and** $clauses(E)$

$E ::=$	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.

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

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

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

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

4.2 Judgments and Typing Rules

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

Judgments of the Type System:

$E \vdash \diamond$	environment E is well-formed
$E \vdash M : T$	in environment E, message M has type T
$E \vdash P$	in environment E, process P is well-typed

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

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

(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 the environment E. For example, if

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

then $E \vdash \mathbf{ok} : \mathbf{Ok}(\text{Report}(\text{alice}, 42, \text{report42}))$. 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) $\dfrac{E \vdash \diamond}{E \vdash \mathbf{0}}$ (Proc Rep) $\dfrac{E \vdash P}{E \vdash !P}$ (Proc Res) $\dfrac{E, x{:}T \vdash P \quad T \text{ generative}}{E \vdash \mathbf{new}\ x{:}T; P}$

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

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

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

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

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

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

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

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

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

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

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

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

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), $\varnothing \vdash$ 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).

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

Theorem 2 (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 3 (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.

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 a particular syntax [15] as follows:

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

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.

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. A natural strategy for future work is to split the typing context into linear and classical parts.

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 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 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 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 (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,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 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

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 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: $[\![S]\!]$

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

$$[\![S]\!] = \prod_{C\in S}[\![C]\!] \qquad [\![\varnothing]\!] = \mathbf{0}$$

$$[\![L:-L_1,\ldots,L_m]\!] = !\,[\![L_1,\ldots,L_m]\!]^{\varnothing}[[\![L]\!]^+] \quad \text{for } m \geq 0$$

$$[\![p(u_1,\ldots,u_n)]\!]^+ = \mathbf{out}\ p_n(u_1,\ldots,u_n,\mathbf{ok})$$

$$[\![L_1,L_2,\ldots,L_m]\!]^{\Sigma}[\cdot] = [\![L_1]\!]^{\Sigma}\left[[\![L_2,\ldots,L_m]\!]^{\Sigma\cup fv(L_1)}[\cdot]\right] \qquad [\![\varepsilon]\!]^{\Sigma}[\cdot] = [\cdot]$$

$$[\![p(u_1,\ldots,u_n)]\!]^{\Sigma}[\cdot] = \mathbf{in}\ p_n(\underline{u}_1,\ldots,\underline{u}_n,{=}\mathbf{ok});[\cdot]$$

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 $\exists P'.P \rightarrow_{\equiv}^{*} P' \mid [\![L]\!]^+$

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

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

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 (Typability of Encoding) *Let S be a Datalog program using predicates $\widetilde{p_n}$ and names $\widetilde{y}$ with $fn(S)\subseteq\{\widetilde{y}\}$. Let $E = \widetilde{y}{:}\widetilde{\mathbf{Un}},\widetilde{p_n}{:}\widetilde{T_{n,p}}$. 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 (Correctness and Completeness) *Let S be a Datalog program and F a fact. We have $S \models F$ if and only if $[\![S]\!] \Downarrow_F$.*

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 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 also extend Datalog with notions of locality and partial trust, considering for examples facts and clauses relative to each principal. A first step will be to consider a combination of the present system with ideas from a recent work [18] 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 d of the derivation tree for $S \models F$.

- ($d = 1$): Suppose $S \models F$ because $F \in S$.
  By set theory, $F \in S \cup \{C\}$.
  By (Infer Fact), $S \cup \{C\} \models F$.
- ($d = m+1$): Suppose $S \models L\sigma$ because $L :- L_1, \ldots, L_n \in S$ and $\forall i \in 1..n\ S \models L_i\sigma$.
  By set theory, $L :- L_1, \ldots, L_n \in S \cup \{C\}$.
  By inductive hypothesis, $\forall i \in 1..n\ S \cup \{C\} \models L_i\sigma$.
  By (Infer Fact), $S \cup \{C\} \models L\sigma$. □

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

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

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

Proof By induction on the depth d of the derivation tree for $S \models F$.

- ($d = 1$): Suppose $S \models F$ because $F \in S$.
  By $F \in S$, $F\sigma \in S\sigma$.
  By (Infer Fact), $S\sigma \models F\sigma$.
- ($d = m+1$): Suppose $S \models L\rho$ because $L :- L_1, \ldots, L_n \in S$ and $\forall i \in 1..n\ S \models L_i\rho$.
  By $L :- L_1, \ldots, L_n \in S$, $L\sigma :- L_1\sigma, \ldots, L_n\sigma \in S\sigma$.
  By inductive hypothesis, $\forall i \in 1..n\ S\sigma \models L_i\rho\sigma$.
  Since ρ replaces variables with names and σ names with messages, $\rho\sigma = \sigma\rho'$ where ρ' is the result of applying σ to ρ.
  The hypotheses can be re-written as $\forall i \in 1..n\ S\sigma \models L_i\sigma\rho'$.
  By (Infer Fact), $S\sigma \models L\sigma\rho'$. □

Proof 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) *$S \cup \{L_1\sigma, \ldots, L_n\sigma\} \models L\sigma$, 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 $\widetilde{X} = fv(L_1, \ldots, L_n)$.*

Proof We prove the two implications separately.

- $((2) \Rightarrow (1))$: By induction on the structure of S'. Let $C = L :- L_1, \ldots, L_n$.
  - $(S' = \varnothing)$: By definition of $\models$, $\{F \mid \{C\} \models F\} = \varnothing \subseteq \{F \mid S \models F\}$.
  - $(S' = S'' \cup \{F'\})$ By hypothesis, $\{F \mid S'' \cup \{C\} \models F\} \subseteq \{F \mid S'' \cup S \models F\}$.
    We distinguish two cases.
    Suppose $\{F \mid S'' \cup \{C\} \models F\} = \{F \mid S' \cup \{C\} \models F\}$.
    By Lemma 4, $\{F \mid S'' \cup S \models F\} \subseteq \{F \mid S' \cup S \models F\}$.
    By transitivity, $\{F \mid S' \cup \{C\} \models F\} \subseteq \{F \mid S' \cup S \models F\}$.
    Suppose instead $\{F \mid S'' \cup \{C\} \models F\} \subset \{F \mid S' \cup \{C\} \models F\}$.
    We distinguish two further cases.
    - $\{F \mid S' \cup \{C\} \models F\} = \{F'\} \cup \{F \mid S'' \cup \{C\} \models F\}$.
      By (Infer Fact), $S' \cup S \models F'$.
      By definition, $\{F'\} \subseteq \{F \mid S' \cup S \models F\}$.
      By set theory, $\{F'\} \cup \{F \mid S'' \cup \{C\} \models F\} \subseteq \{F \mid S' \cup S \models F\}$.
    - $\{F \mid S' \cup \{C\} \models F\} = \{F''\} \cup \{F \mid S'' \cup \{C\} \models F\}$, where $F'' \neq F'$.
      By $F'' \neq F'$, an instance of (Infer Fact) must be used to derive F''.
      By construction, C is the only rule present.
      By (Infer Fact), $F'' = L\rho$, for some ρ such that $\forall i \in 1..n\ S' \cup \{C\} \models L_i\rho$.
      We show that $S' \cup \{C\} \models L_i\rho \Rightarrow S' \cup S \models L_i\rho$, for a generic $i \in 1..n$, by induction on the depth d of the derivation tree.
      - $(d = 1)$: Suppose $S' \cup \{C\} \models L_i\rho$.
        By hypothesis, $L_i\rho \in S'$.
        By (Infer Fact), $S' \cup S \models L_i\rho$.
      - $(d = m+1)$: Suppose $S' \cup \{C\} \models L_i\rho\rho'$.
        By hypothesis, $\forall i \in 1..n\ S' \cup \{C\} \models L_i\rho\rho'$.
        By inductive hypothesis, $\forall i \in 1..n\ S' \cup S \models L_i\rho\rho'$.
        By hypothesis of the theorem, $S \cup \{L_1\sigma, \ldots, L_n\sigma\} \models L\sigma$, where $\sigma = \{\widetilde{x}/\widetilde{X}\}$ is an injective substitution, $\{\widetilde{x}\} \cap (fn(S) \cup fn(L :- L_1, \ldots, L_n)) = \varnothing$ and $\widetilde{X} = fv(L_1, \ldots, L_n)$.
        By definition of σ, there exists σ' such that $\sigma\sigma' = \rho\rho'$.
        By Lemma 6, $S\sigma' \cup \{L_1\sigma\sigma', \ldots, L_n\sigma\sigma'\} \models L\sigma\sigma'$.
        By definition of σ and σ', $S\sigma' = S$.
        By Lemma 4 and Lemma 5, $S' \cup S \cup \{L_1\sigma\sigma', \ldots, L_{n-1}\sigma\sigma'\} \models L\sigma\sigma'$.
        By iterating the argument on all n, we get to $S' \cup S \models L\sigma\sigma'$.
        By definition of σ', $S' \cup S \models L\rho\rho'$.

- $((1) \Rightarrow (2))$: Suppose that $\{F \mid S' \cup \{L :- L_1, \ldots, L_n\} \models F\} \subseteq \{F \mid S' \cup S \models F\}$ for all S'.

  Consider the case for $S' = \{L_1\sigma, \ldots, L_n\sigma\}$ for σ in the hypothesis of the theorem.

  By (Infer Fact), $L\sigma \in \{F \mid S' \cup \{L :- L_1, \ldots, L_n\} \models F\}$.

  By set theory, $L\sigma \in \{F \mid S' \cup S \models F\}$.

  By definition of S', $\{L_1\sigma, \ldots, L_n\sigma\} \cup S \models L\sigma$. □

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.

Proof of Proposition 1 (Monotonicity) *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. □

Proof of Proposition 2 (Substitutivity) *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 $S \models C$.

(Infer Fact) By Lemma 6.

(Infer Clause) Suppose $S \models L :- L_1, \ldots, L_n$ because, for some injective substitution ρ, from $fv(L_1, \ldots, L_n)$ to fresh names, $S \cup \{L_1\rho, \ldots, L_n\rho\} \models L\rho$.

By Lemma 6, $(S \cup \{L_1\rho, \ldots, L_n\rho\})\sigma \models L\rho\sigma$.

By applying the substitution, $(S\sigma \cup \{L_1\rho\sigma, \ldots, L_n\rho\sigma\} \models L\rho\sigma$.

Since ρ replaces variables with fresh names and σ replaces existing names with messages, $\rho\sigma = \sigma\rho$.

The hypotheses can be re-written as $(S\sigma \cup \{L_1\sigma\rho, \ldots, L_n\sigma\rho\} \models L\sigma\rho$.

By (Infer Clause), $S\sigma \models L\sigma :- L_1\sigma, \ldots, L_n\sigma$.

By factorizing the substitution, $S\sigma \models (L :- L_1, \ldots, L_n)\sigma$. □

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 Let $\sigma = \{\widetilde{y}/\widetilde{x}\}$. By (Subst), $(S \cup S\sigma \cup S')\sigma \models C\sigma$.
By definition, $(S \cup S\sigma \cup S')\sigma = S\sigma \cup S'\sigma$, hence $S\sigma \cup S'\sigma \models C\sigma$.
Since the $\widetilde{y}$ are fresh and distinct, $\rho = \{\widetilde{x}/\widetilde{y}\}$ is the inverse of σ.
By (Subst), $(S\sigma \cup S'\sigma)\rho \models C\sigma\rho$.
By definition, $(S\sigma \cup S'\sigma)\rho = S\sigma\rho \cup S'\sigma\rho = S \cup S'$ and $C\sigma\rho = C$.
We conclude with $S \cup S' \models C$. □

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: $env(G)$

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

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

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

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 $x \notin fn(G)$)
$\mathbf{new}\ x_1{:}T_1; \mathbf{new}\ x_2{:}T_2; G \sqsubseteq$ $\mathbf{new}\ x_2{:}T_2; \mathbf{new}\ x_1{:}T_1; G$	(G Sub Res Res) (for $x_1 \neq x_2, x_1 \notin fn(T_2), x_2 \notin fn(T_1)$)

Structural Congruence for Guarantees: $G \equiv H$

$G \equiv H \triangleq G \sqsubseteq H$ 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)
$$\frac{E, x{:}T \vdash P : G \quad T\,generative}{E \vdash \mathbf{new}\ x{:}T; P : \mathbf{new}\ x{:}T; G}$$

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

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

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

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

(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 We split the proof in two parts:

(1) if $E \vdash P$ and $env(P)^{\widetilde{x}} = E'$ then $E \vdash P : G$, for some G such that $E' = env(G)^{\widetilde{x}}$;

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

(1) By induction on the derivation of $E \vdash P$.

(Proc Nil) Suppose $E \vdash \mathbf{0}$.

By hypothesis, $E \vdash \diamond$.

By definition, the only E' such that $env(\mathbf{0})^{\widetilde{x}} = E'$ is $E' = \varnothing$, and necessarily $\widetilde{x} = \varnothing$.

By (ProcG Nil), $E \vdash \mathbf{0} : \mathbf{0}$.

By definition, $env(\mathbf{0})^{\varnothing} = \varnothing$.

(Proc Par) Suppose $E \vdash P \mid Q$.

By hypothesis, $E, env(Q)^{\widetilde{y}} \vdash P$, $E, env(P)^{\widetilde{x}} \vdash Q$, $fn(P \mid Q) \subseteq dom(E)$.

By inductive hypothesis, $E, env(Q)^{\widetilde{y}} \vdash P : G_1$ with $env(P)^{\widetilde{x}} = env(G_1)^{\widetilde{x}}$ and $E, env(P)^{\widetilde{x}} \vdash Q : G_2$ with $env(Q)^{\widetilde{y}} = env(G_2)^{\widetilde{y}}$.

By (ProcG Par), $E \vdash P \mid Q : G_1 \mid G_2$.

By definition, $env(P \mid Q)^{\widetilde{x},\widetilde{y}} = env(P)^{\widetilde{x}}, env(Q)^{\widetilde{y}}$.

By definition, $env(G_1 \mid G_2)^{\widetilde{x},\widetilde{y}} = env(G_1)^{\widetilde{x}}, env(G_2)^{\widetilde{y}}$.

By transitivity, $env(P \mid Q)^{\widetilde{x},\widetilde{y}} = env(G_1 \mid G_2)^{\widetilde{x},\widetilde{y}}$.

(Proc Rep) Suppose $E \vdash !P$.

By hypothesis, $E \vdash P$.

By inductive hypothesis, if $env(P)^{\widetilde{x}} = E'$ then there exists a G such that $E \vdash P : G$ and $env(G)^{\widetilde{x}} = E'$.

By (ProcG Rep), $E \vdash !P : G$.

By definition, $env(!P)^{\widetilde{x}} = env(P)^{\widetilde{x}}$.

(Proc Res) Suppose $E \vdash \mathbf{new}\ x{:}T; P$.

By hypothesis, $E, x{:}T \vdash P$.

By inductive hypothesis, if $env(P)^{\widetilde{x}} = E'$ then there is G such that $E, x{:}T \vdash P : G$ and $env(G)^{\widetilde{x}} = E'$.

By (ProcG Res), $E \vdash \mathbf{new}\ x{:}T; P : \mathbf{new}\ x{:}T; G$.

By definition, $env(\mathbf{new}\ x{:}T; P)^{y,\widetilde{x}} = y{:}T, env(P)^{\widetilde{x}}$.

By definition, $env(\mathbf{new}\ x{:}T; G)^{y,\widetilde{x}} = y{:}T, env(G)^{\widetilde{x}}$.

By transitivity, $env(\mathbf{new}\ x{:}T; P)^{y,\widetilde{x}} = env(\mathbf{new}\ x{:}T; G)^{y,\widetilde{x}}$.

(Proc Fact) Suppose $E \vdash C$.

By hypothesis, $E, C \vdash \diamond$.

By definition, the only E' such that $env(C)^{\widetilde{x}} = E'$ is $E' = C$, and necessarily $\widetilde{x} = \varnothing$.

By (ProcG Fact), $E \vdash C : C$.

By definition, $env(C)^{\varnothing} = C$.

The other cases are easy.

(2) By induction on the derivation of $E \vdash P : G$, similarly to the previous point. □

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).

Lemma 9 *If $E \vdash \diamond$ and $x \in dom(E)$ then $fn(E(x)) \subseteq dom(E)$.*

Proof By structural induction on E. □

Lemma 10 (Unique Types) *If $E \vdash x : T$ and $E \vdash x : U$ then $T = U$.*

Proof By structural induction on E, noticing that since $E \vdash x : T$ then rule (Msg x) applies, and therefore $E(x) = T$. □

Lemma 11 *If $E \vdash M : T$ then $fn(T) \cup fn(M) \subseteq dom(E)$ and $E \vdash \diamond$.*

Proof By induction on the derivation of $E \vdash M : T$.

(Msg x) Suppose $E \vdash x : E(x)$.

By hypothesis, $E \vdash \diamond$ and $x \in dom(E)$.

By Lemma 9, $fn(E(x)) \subseteq dom(E)$.

(Msg Encrypt) Suppose $E \vdash \{M\}N : \mathbf{Un}$.

By hypothesis, $E \vdash M : T$ and $E \vdash N : \mathbf{Key}(T)$.

By definition, $fn(\mathbf{Un}) = \varnothing$.

By inductive hypothesis, $E \vdash \diamond$, $fn(M) \subseteq dom(E)$ and $fn(N) \subseteq dom(E)$.

By set theory and definition of free names, $fn(\{M\}N) \subseteq dom(E)$.

(Msg Encrypt Un) Suppose $E \vdash \{M\}N : \mathbf{Un}$.

By hypothesis, $E \vdash M : \mathbf{Un}$ and $E \vdash N : \mathbf{Un}$.

By definition, $fn(\mathbf{Un}) = \varnothing$.

By inductive hypothesis, $E \vdash \diamond$, $fn(M) \subseteq dom(E)$ and $fn(N) \subseteq dom(E)$.

By set theory and definition of free names, $fn(\{M\}N) \subseteq dom(E)$.

(Msg Pair) Suppose $E \vdash (M,N) : (x{:}T,U)$.

By hypothesis, $E \vdash M : T$ and $E \vdash N : U\{M/x\}$.

By inductive hypothesis, $E \vdash \diamond$, $fn(M) \cup fn(T) \subseteq dom(E)$, and $fn(N) \cup fn(U\{M/x\}) \subseteq dom(E)$.

By definition, $fn((x{:}T,U)) = fn(T) \cup (fn(U) \setminus \{x\})$.

By set theory and definition of free names, $fn((M,N)) \cup fn((x{:}T,U)) \subseteq dom(E)$.

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

By hypothesis, $E \vdash M : \mathbf{Un}$ and $E \vdash N : \mathbf{Un}$.

By definition, $fn(\mathbf{Un}) = \varnothing$.

By inductive hypothesis, $E \vdash \diamond$, $fn(M) \subseteq dom(E)$, and $fn(N) \subseteq dom(E)$.

By set theory and definition of free names, $fn((M,N)) \cup fn((x{:}T,U)) \subseteq dom(E)$.

(Msg Ok) Suppose $E \vdash \mathbf{ok} : \mathbf{Ok}(D)$.

By hypothesis, $E \vdash \diamond$, $fn(D) \subseteq dom(E)$ and $clauses(E) \models D$.

By definition, $fn(\mathbf{ok}) = \varnothing$.

(Msg Ok Un) Suppose $E \vdash \mathbf{ok} : \mathbf{Un}$.

By hypothesis, $E \vdash \diamond$. By definition, $fn(\mathbf{ok}) = fn(\mathbf{Un}) = \varnothing$. □

Lemma 12 *If $E \vdash P : G$ then $fn(G) \subseteq fn(P) \subseteq dom(E)$ and $E, env(G)^{\widetilde{z}} \vdash \diamond$ for $\{\widetilde{z}\} \cap dom(E) = \varnothing$.*

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

(ProcG Nil) Trivial.

(ProcG Par) Suppose $E \vdash P \mid Q : G_1 \mid G_2$.

By hypothesis, $E, env(G_2)^{\widetilde{y}} \vdash P : G_1$, $E, env(G_1)^{\widetilde{x}} \vdash Q : G_2$, $fn(P \mid Q) \subseteq dom(E)$.

By inductive hypotheses,
$E, env(G_2)^{\widetilde{y}}, env(G_1)^{\widetilde{x'}} \vdash \diamond$ where $\{\widetilde{x'}\} \cap dom(E, env(G_2)^{\widetilde{y}}) = \varnothing$, $fn(G_1) \subseteq fn(P)$, and $E, env(G_1)^{\widetilde{x}}, env(G_2)^{\widetilde{y'}} \vdash \diamond$ where $\{\widetilde{y'}\} \cap dom(E, env(G_1)^{\widetilde{x}}) = \varnothing$, $fn(G_2) \subseteq fn(Q)$.

Choosing $\widetilde{z} = \widetilde{x}, \widetilde{y'}$, we have $E, env(G_1 \mid G_2)^{\widetilde{z}} \vdash \diamond$, $fn(G_1 \mid G_2) \subseteq fn(P \mid Q) \subseteq dom(E)$, and $\{\widetilde{z}\} \cap dom(E) = \varnothing$.

(ProcG Rep) Suppose $E \vdash !P : G$.

By hypothesis, $E \vdash P : G$.

By inductive hypothesis, $fn(G) \subseteq fn(P) \subseteq dom(E)$ and $E, env(G)^{\widetilde{z}} \vdash \diamond$ for $\{\widetilde{z}\} \cap dom(E) = \varnothing$, and we conclude.

(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, $fn(G) \subseteq fn(P) \subseteq dom(E, x{:}T)$ and $E, x{:}T, env(G)^{\widetilde{z}} \vdash \diamond$ for $\{\widetilde{z}\} \cap dom(E, x{:}T) = \varnothing$.

By definition, $E, env(\mathbf{new}\ x{:}T; G)^{x,\widetilde{z}} = E, x{:}T, env(G)^{\widetilde{z}}$, and we conclude.

(ProcG Query) Suppose $E \vdash \mathbf{expect}\ C : \mathbf{0}$.

By hypothesis, $E, C \vdash \diamond$. By (Env C), $fn(C) \subseteq dom(E)$ and we conclude.

(ProcG Fact) Suppose $E \vdash C : C$.

By hypothesis, $E, C \vdash \diamond$. By (Env C), $fn(C) \subseteq dom(E)$ and we conclude.

(ProcG Input) Suppose $E \vdash \mathbf{in}\ M(x{:}T); P : \mathbf{0}$.

By hypothesis, $E \vdash M : \mathbf{Ch}(T)$ and $E, x{:}T \vdash P : G$.

By inductive hypothesis, $fn(G) \subseteq fn(P) \subseteq dom(E, x{:}T)$.

By set theory, $fn(\mathbf{0}) \subseteq (fn(P) \setminus \{x\}) \subseteq dom(E)$.

The remaining cases are similar to the case of (ProcG Input). □

Lemma 13 *If $env(G)^{\widetilde{x}}$ s defined, then $dom(env(G)^{\widetilde{x}}) = \{\widetilde{x}\}$.*

Proof By structural induction on G. □

Lemma 14 *If $\widetilde{y}$ is a vector of distinct names and $env(G)^{\widetilde{x}}$ and $env(G)^{\widetilde{y}}$ are defined then $clauses(env(G)^{\widetilde{x}}) = clauses(env(G)^{\widetilde{y}})\{\widetilde{x}/\widetilde{y}\}$.*

Proof By structural induction on G. □

Lemma 15 (Unique Guarantees) *If $E \vdash P : G$ and $E' \vdash P : G'$ then $G = G'$.*

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

(ProcG Par) Suppose $E \vdash P \mid Q : G_1 \mid G_2$.

By hypothesis, $E, env(G_2)^{\widetilde{y}} \vdash P : G_1$, $E, env(G_1)^{\widetilde{x}} \vdash Q : G_2$, $fn(P \mid Q) \subseteq dom(E)$.

Only (ProcG Par) can derive $E' \vdash P \mid Q : G'$, therefore we have $G' = G'_1 \mid G'_2$, $E', env(G'_2)^{\widetilde{y'}} \vdash P : G'_1$, $E', env(G'_1)^{\widetilde{x}} \vdash Q : G'_2$, $fn(P \mid Q) \subseteq dom(E')$.

By inductive hypotheses, $G_1 = G'_1$ and $G_2 = G'_2$.

We conclude with $G_1 \mid G_2 = G'_1 \mid G'_2$.

(ProcG Rep) Suppose $E \vdash !P : G$.

By hypothesis, $E \vdash P : G$.

Only (ProcG Rep) can derive $E' \vdash !P : G'$, so we have $E' \vdash P : G'$.

By inductive hypothesis, $G = G'$.

(ProcG Res) Suppose $E \vdash \mathbf{new}\ x{:}T; P : (x{:}T)G$.

By hypothesis, $E, x{:}T \vdash P : G$.

Only (ProcG Res) can derive $E' \vdash \mathbf{new}\ x{:}T; P : G'$, so we have $E', x{:}T \vdash P : G'$.

By inductive hypothesis, $G = G'$.

(ProcG Fact) Suppose $E \vdash C : C$.

By hypothesis, $E, C \vdash \diamond$.

Only (ProcG Fact) can derive $E' \vdash C : G'$, so we have $G' = C = G$, and we conclude.

The other cases are trivial, because both G and G' in $E \vdash P : G$ and $E' \vdash P : G'$ are always forced to be $\mathbf{0}$. □

Lemma 16 (Strengthening) *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 $E, E' \vdash \diamond$.*

Proof Both cases follow by induction on the depths of the derivation. □

Lemma 17 *If $E, env(G)^{\widetilde{x}}, env(G)^{\widetilde{y}}, E' \vdash \mathscr{J}$ and $\{\widetilde{x}\} \cap (fn(E') \cup fn(\mathscr{J})) = \varnothing$ then $E, env(G)^{\widetilde{y}}, E' \vdash \mathscr{J}$.*

Proof

- The case for $E, env(G)^{\widetilde{x}}, env(G)^{\widetilde{y}}, E' \vdash \diamond$ follows from Lemma 16.
- The case for $E, env(G)^{\widetilde{x}}, env(G)^{\widetilde{y}}, E' \vdash M : T$ is by induction on the derivation. Let $E_1 = E, env(G)^{\widetilde{x}}, env(G)^{\widetilde{y}}, E'$ and $E_2 = E, env(G)^{\widetilde{y}}, E'$.

  By the previous point, we have that if $E_1 \vdash \diamond$ then $E_2 \vdash \diamond$.

  (Msg x) Suppose $E_1 \vdash x : E_1(x)$.

  By hypothesis, $E_1 \vdash \diamond$.

  By hypothesis of the lemma, $x \in (dom(E_1) \setminus \{\widetilde{x}\})$.

  By (Msg x), $E_2 \vdash x : E_2(x)$.

(Msg Ok) Suppose $E_1 \vdash \mathbf{ok} : \mathbf{Ok}(S)$.

By hypothesis, $E_1 \vdash \diamond$ and $fn(S) \subseteq (dom(E_1) \setminus \{\widetilde{x}\})$ and $\forall C \in S$ $clauses(E_1) \models C$.

By definition, $clauses(E_1) = clauses(E_2) \cup clauses(env(G)^{\widetilde{x}})$.

By $E_1 \vdash \diamond$, and by hypothesis of the lemma, $\{\widetilde{x}\} \cap (fn(clauses(E_2)) \cup fn(S)) = \varnothing$.

By Lemma 14, $clauses(env(G)^{\widetilde{y}}) = clauses(env(G)^{\widetilde{x}})\{\widetilde{y}/\widetilde{x}\}$.

By Lemma 7, $\forall C \in S.clauses(E_2) \models C$.

The others cases are easy.

- The case for $E, env(G)^{\widetilde{x}}, env(G)^{\widetilde{y}}, E' \vdash P : H$ is by induction on the derivation.

  Let $E_1 = E, env(G)^{\widetilde{x}}, env(G)^{\widetilde{y}}, E'$ and $E_2 = E, env(G)^{\widetilde{y}}, E'$.

  By the first point of this lemma, we have that if $E_1 \vdash \diamond$ then $E_2 \vdash \diamond$.

  (ProcG Nil) Suppose $E_1 \vdash \mathbf{0} : \mathbf{0}$.

  By hypothesis, $E \vdash \diamond$.

  By $E_2 \vdash \diamond$ and (ProcG Nil), $E_2 \vdash \mathbf{0} : \mathbf{0}$.

  (ProcG Par) Suppose $E_1 \vdash P \mid Q : G_1 \mid G_2$.

  By Lemma 12, $fn(G_1 \mid G_2) \subseteq fn(P \mid Q)$.

  By hypothesis, $E_1, env(G_2)^{\widetilde{z}} \vdash P : G_1$ and $E_1, env(G_1)^{\widetilde{w}} \vdash Q : G_2$, and $fn(P \mid Q) \subseteq dom(E_1)$.

  By inductive hypothesis, $E_2, env(G_2)^{\widetilde{z}} \vdash P : G_1$ and $E_2, env(G_1)^{\widetilde{w}} \vdash Q : G_2$.

  By (ProcG Par), $E_2 \vdash P \mid Q : G_1 \mid G_2$.

  (ProcG Rep) Suppose $E_1 \vdash !P : G'$.

  By hypothesis, $E_1 \vdash P : G'$.

  By inductive hypothesis, $E_2 \vdash P : G'$.

  By (ProcG Rep), $E_2 \vdash !P : G'$.

  (ProcG Res) Suppose $E_1 \vdash \mathbf{new}\ x{:}T; P : \mathbf{new}\ x{:}T; G'$.

  By hypothesis, $E_1, x{:}T \vdash P : G'$.

  By inductive hypothesis, $E_2, x{:}T \vdash P : G'$.

  By (ProcG Res), $E_2 \vdash \mathbf{new}\ x{:}T; P : \mathbf{new}\ x{:}T; G'$.

  (ProcG Query) Suppose $E_1 \vdash \mathbf{expect}\ F : \mathbf{0}$.

  By hypothesis, $E_1, C \vdash \diamond$, $clauses(E_1) \models F$.

  By the first point of this lemma, $E_2, C \vdash \diamond$.

  By definition, $clauses(E_1) = clauses(E_2) \cup clauses(env(G)^{\widetilde{x}})$.

  By $E_1, C \vdash \diamond$, and by hypothesis of the lemma, $\{\widetilde{x}\} \cap (fn(clauses(E_2)) \cup fn(C)) = \varnothing$.

  By Lemma 14, $clauses(env(G)^{\widetilde{y}}) = clauses(env(G)^{\widetilde{x}})\{\widetilde{y}/\widetilde{x}\}$.

  By Lemma 7, $clauses(E_2) \models C$.

  By (ProcG Query), $E_2 \vdash \mathbf{expect}\ F : \mathbf{0}$.

(ProcG Fact) Suppose $E_1 \vdash C : C$.

By hypothesis, $E_1, C \vdash \diamond$.

By the first point of this lemma, $E_2, C \vdash \diamond$.

By (ProcG Fact), $E_2 \vdash C : C$.

(ProcG Input) Suppose $E_1 \vdash \mathbf{in}\ M(x{:}T); P : \mathbf{0}$.

By hypothesis, $E_1 \vdash M : \mathbf{Ch}(T)$, $E_1, x{:}T \vdash P : G'$.

By the previous point of this lemma, $E_2 \vdash M : \mathbf{Ch}(T)$.

By Lemma 12, $fn(G') \subseteq fn(P)$.

By $E_1, x{:}T \vdash \diamond$, $\{\tilde{x}\} \cap fn(G') = \varnothing$.

By inductive hypothesis, $E_2, x{:}T \vdash P : G'$.

By (ProcG Input), $E_2 \vdash \mathbf{in}\ M(x{:}T); P : \mathbf{0}$.

The other cases are similar to the case for (ProcG Input). □

Lemma 18 (Exchange) *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* $E_1, E_2, E_3, E_4 \vdash \mathscr{J}$.

Proof We split the proof depending on $\mathscr{J}$.

- By induction on the depth of the derivation of $E_1, E_2, E_3, E_4 \vdash \diamond$. Consider the last rule.

  (Env $\varnothing$) Trivial.

  (Env x) Suppose $E, x{:}T \vdash \diamond$.

  By hypothesis, $E \vdash \diamond$, $fn(T) \subseteq dom(E)$ and $x \notin dom(E)$.

  If $E = E_1, E_2, E_3, E_4$ where $E_4 = E_4', x{:}T$ we conclude applying the inductive hypothesis.

  If $E = E_1, E_2, E_3$ and $E_3 = E_3', x{:}T$, by inductive hypothesis $E_1, E_3', E_2 \vdash \diamond$.

  By Lemma 16, $E_1, E_3' \vdash \diamond$.

  By (Env x), $E_1, E_3 \vdash \diamond$.

  By Lemma 22, $E_1, E_3, E_2 \vdash \diamond$.

  The case for $E = E_1, E_2$ is trivial.

  (Env C) Similar to the previous case.

- By a straightforward induction on the depth of the derivation of $E_1, E, E', E_2 \vdash M : T$, using point (1).

- By induction on the depth of the derivation of $E_1, E, E', E_2 \vdash P : G$.

  (ProcG Nil) Suppose $E_1, E_2, E_3, E_4 \vdash \mathbf{0} : \varnothing$.

  By hypothesis, $E_1, E_2, E_3, E_4 \vdash \diamond$.

  By point (1), $E_1, E_3, E_2, E_4 \vdash \diamond$.

  By (ProcG Nil), $E_1, E_3, E_2, E_4 \vdash \mathbf{0} : \varnothing$.

(ProcG Input) Suppose $E_1,E_2,E_3,E_4 \vdash \textbf{in}\ M(x{:}T);P : \varnothing$.

By hypothesis, $E_1,E_2,E_3,E_4 \vdash M : \textbf{Ch}(T)$, $E_1,E_2,E_3,E_4,x{:}T \vdash P : G$.

By point (ii), $E_1,E_3,E_2,E_4 \vdash M : \textbf{Ch}(T)$.

By inductive hypothesis, $E_1,E_3,E_2,E_4,x{:}T \vdash P : G$.

By (ProcG Input), $E_1,E_3,E_2,E_4 \vdash \textbf{in}\ M(x{:}T);P : \varnothing$.

The other cases are similar. □

Lemma 19 *(i) If $G \sqsubseteq G'$ then $fn(G) \subseteq fn(G')$. (ii) If $G \equiv G'$ then $fn(G) = fn(G')$.*

Proof (i) By induction on the length of the derivation of $G \sqsubseteq G'$.
(ii) By definition, of $G \equiv G'$ and by point (i). □

Lemma 20 *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 $\{\widetilde{z}\} \cap (fn(E') \cup fn(\mathscr{J})) = \varnothing$.*

Proof By induction on the derivation of $G \sqsubseteq G'$.

(G Sub Refl) Suppose $G \sqsubseteq G$.

By hypothesis of the lemma, $E, env(G)^{\widetilde{x}}, E' \vdash \mathscr{J}$.

(G Sub Trans) Suppose $G \sqsubseteq G'$.

By hypothesis, $G \sqsubseteq H$ and $H \sqsubseteq G'$.

By inductive hypotheses, $E, env(H)^{\widetilde{w}}, E' \vdash \mathscr{J}$ and $E, env(G')^{\widetilde{z}}, E' \vdash \mathscr{J}$ for some $\widetilde{w}$ and $\widetilde{z}$ such that $\{\widetilde{w}\} \cap (fn(E') \cup fn(\mathscr{J})) = \varnothing$ and $\{\widetilde{z}\} \cap (fn(E') \cup fn(\mathscr{J})) = \varnothing$, and we conclude.

(G Sub Res) Suppose $\textbf{new}\ x{:}T;G \sqsubseteq \textbf{new}\ x{:}T;H$. By hypothesis, $G \sqsubseteq H$.

By hypothesis of the lemma, $E, env(\textbf{new}\ x{:}T;G)^{\widetilde{x}}, E' \vdash \mathscr{J}$.

By definition, $env(\textbf{new}\ x{:}T;G)^{\widetilde{x}} = x{:}T, env(G)^{\widetilde{y}}$ for $\widetilde{x} = x,\widetilde{y}$.

By inductive hypothesis, $E, x{:}T, env(H)^{\widetilde{w}}, E' \vdash \mathscr{J}$, where $\{\widetilde{w}\} \cap (fn(E') \cup fn(\mathscr{J})) = \varnothing$.

By definition, $env(\textbf{new}\ x{:}T;H)^{\widetilde{z}} = x{:}T, env(H)^{\widetilde{w}}$ where $\widetilde{z} = x,\widetilde{w}$, and we conclude.

(G Sub Par) Suppose $G \mid H \sqsubseteq G' \mid H$.

By hypothesis, $G \sqsubseteq G'$.

By hypothesis of the lemma, $E, env(G \mid H)^{\widetilde{x}}, E' \vdash \mathscr{J}$.

By definition, $env(G \mid H)^{\widetilde{x}} = env(G)^{\widetilde{x_1}}, env(H)^{\widetilde{x_2}}$, where $\{\widetilde{x_1}\} \cap fn(H) = \varnothing$.

By inductive hypothesis, $E, env(G')^{\widetilde{z_1}}, env(H)^{\widetilde{x_2}}, E' \vdash \mathscr{J}$, where $\{\widetilde{z_1}\} \cap fn(H) = \varnothing$.

By definition, $env(G')^{\widetilde{z_1}}, env(H)^{\widetilde{x_2}} = env(G' \mid H)^{\widetilde{z}}$, where $\widetilde{z} = \widetilde{z_1},\widetilde{x_2}$, and we conclude.

(G Sub Par Zero) Suppose $G \mid \mathbf{0} \sqsubseteq G$.

By hypothesis of the lemma, $E, env(G \mid \mathbf{0})^{\widetilde{x}}, E' \vdash \mathscr{J}$.

By definition, $env(G \mid \mathbf{0})^{\widetilde{x}} = env(G)^{\widetilde{x}}$, and we conclude.

(G Sub Par Comm) Suppose $G \mid H \sqsubseteq H \mid G$.

Suppose $G \mid H \sqsubseteq H \mid G$.

By hypothesis of the lemma, $E, env(G \mid H)^{\widetilde{x}}, E' \vdash \mathscr{J}$.

By definition, $env(G \mid H)^{\widetilde{x}} = env(G)^{\widetilde{x_1}}, env(H)^{\widetilde{x_2}}$, where $\{\widetilde{x_1}\} \cap fn(H) = \varnothing$ and $\{\widetilde{x_2}\} \cap fn(G) = \varnothing$.

By Lemma 18, $E, env(H)^{\widetilde{x_2}}, env(G)^{\widetilde{x_1}}, E' \vdash \mathscr{J}$.

By definition, $env(H \mid G)^{\widetilde{x_2},\widetilde{x_1}} = env(H)^{\widetilde{x_2}}, env(H)^{\widetilde{x_1}}$, and we conclude.

(G Sub Par Assoc) Suppose $(G \mid G') \mid H \sqsubseteq G \mid (G' \mid H)$.

By definition, $env((G \mid G') \mid H)^{\widetilde{x}} = env(G \mid G' \mid H))^{\widetilde{x}}$, and we conclude.

(G Sub Idem) Suppose $G \mid G \sqsubseteq G$.

By hypothesis of the lemma, $E, env(G \mid G)^{\widetilde{x}}, E' \vdash \mathscr{J}$.

By definition, $env(G \mid G)^{\widetilde{x}} = env(G)^{\widetilde{y}}, env(G)^{\widetilde{z}}$, where $\{\widetilde{y}\} \cap (fn(E') \cup fn(\mathscr{J})) = \varnothing$.

By Lemma 17, $E, env(G)^{\widetilde{z}}, E' \vdash \mathscr{J}$.

(G Sub Order) Suppose $G \sqsubseteq G \mid H$. By hypothesis of the lemma, $E, env(G)^{\widetilde{x}}, E' \vdash \mathscr{J}$.

By definition, $env(G \mid H)^{\widetilde{z}} = env(G)^{\widetilde{x}}, env(H)^{\widetilde{y}}$ choosing $\widetilde{z} = \widetilde{x}, \widetilde{y}$.

By hypothesis of the lemma, $fn(G) = fn(G')$.

By Lemma 12, $E, env(G)^{\widetilde{x}}, E' \vdash \diamond$, and therefore $fn(G) \cap dom(E') = \varnothing$.

By repeatedly applying Lemma 22, $E, env(G)^{\widetilde{x}}, env(H)^{\widetilde{y}}, E' \vdash \mathscr{J}$ and we conclude.

The remaining cases are analogous to the one for (G Sub Par Comm). □

Lemma 21 *If $E, env(G)^{\widetilde{x}}, E' \vdash P : G$ and $\{\widetilde{x}\} \cap (fn(P) \cup fn(E')) = \varnothing$ then $E, E' \vdash P : G$.*

Proof By induction on the derivation of $E, env(G)^{\widetilde{x}}, E' \vdash P : G$.

(ProcG Par) Suppose $E, env(G_1 \mid G_2)^{\widetilde{x},\widetilde{y}}, E' \vdash P \mid Q : G_1 \mid G_2$.

By hypothesis, $E, env(G_1 \mid G_2)^{\widetilde{x},\widetilde{y}}, E', env(G_2)^{\widetilde{z}} \vdash P : G_1$ and
$E, env(G_1 \mid G_2)^{\widetilde{x},\widetilde{y}}, E', env(G_1)^{\widetilde{w}} \vdash Q : G_2$, and $fn(P \mid Q) \subseteq dom(E)$.

By definition, $env(G_1 \mid G_2)^{\widetilde{x},\widetilde{y}} = env(G_1)^{\widetilde{x}}, env(G_2)^{\widetilde{y}}$.

By inductive hypotheses,
$E, env(G_2)^{\widetilde{y}}, E', env(G_2)^{\widetilde{z}} \vdash P : G_1$ and $E, env(G_1)^{\widetilde{x}}, E', env(G_1)^{\widetilde{w}} \vdash Q : G_2$.

By Lemma 18,
$E, env(G_2)^{\widetilde{y}}, env(G_2)^{\widetilde{z}}, E' \vdash P : G_1$ and $E, env(G_1)^{\widetilde{x}}, env(G_1)^{\widetilde{w}}, E' \vdash Q : G_2$.

By Lemma 17, $E, E', env(G_2)^{\widetilde{z}} \vdash P : G_1$ and $E, E', env(G_1)^{\widetilde{w}} \vdash Q : G_2$.

By (ProcG Par), $E, E' \vdash P \mid Q : G_1 \mid G_2$.

(ProcG Rep) Suppose $E, G, E' \vdash !P : G$.

By hypothesis, $E, G, E' \vdash P : G$.

By inductive hypothesis, $E, E' \vdash P : G$.

By (ProcG Rep), $E, G, E' \vdash !P : G$.

(ProcG Res) Suppose $E, env(\mathbf{new}\ x{:}T; G)^{y,\widetilde{x}}, E' \vdash \mathbf{new}\ x{:}T; P : \mathbf{new}\ x{:}T; G$.

By hypothesis, $E, env(\mathbf{new}\ x{:}T; G)^{y,\widetilde{x}}, E', x{:}T \vdash P : G$.

By definition, $env(\mathbf{new}\ x{:}T; G)^{y,\widetilde{x}} = y{:}T, env(G)^{\widetilde{x}}$.

By inductive hypothesis, $E, y{:}T, E', x{:}T \vdash P : G$, where $y \notin fn(P) \cup fn(E')$.

By Lemma 16, $E, E', x{:}T \vdash P : G$.

By (ProcG Res), $E, E' \vdash \mathbf{new}\ x{:}T; P : \mathbf{new}\ x{:}T; G$.

(ProcG Fact) Suppose $E, C, E' \vdash C : C$.

By hypothesis, $E, C, E', C \vdash \diamond$.

By Lemma 16, $E, E' \vdash \diamond$.

By (ProcG Fact), $E, E' \vdash C : C$.

All the other cases are trivial, as $env(\mathbf{0})^{\varnothing} = \varnothing$. □

Lemma 22 (Weakening) *(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 $E, x{:}T, E' \vdash \mathscr{J}$.*

Proof We split the proof of each point depending on $\mathscr{J}$.

(i) (1) If $E, E' \vdash \diamond$ and $fn(C) \subseteq dom(E)$ then $E, C, E' \vdash \diamond$.

By induction on the depth of the derivation of $E, E' \vdash \diamond$.

(2) If $E, E' \vdash M : T$ and $fn(C) \subseteq dom(E)$ then $E, C, E' \vdash M : T$.

By induction on the depth of the derivation of $E, E' \vdash M : T$.

The most interesting case is the base case for (Msg Ok).

Suppose $E, E' \vdash \mathbf{ok} : \mathbf{Ok}(S)$.

By hypothesis, $E, E' \vdash \diamond$ and $fn(S) \subseteq dom(E, E')$, and for any $C' \in S$, $clauses(E, E') \models C'$.

By (i).1, we have $E, C, E' \vdash \diamond$.

By definition of *dom*, $fn(S) \subseteq dom(E, C, E')$.

By property (Mon) of the authorization logic, for any C' is S, $clauses(E, C, E') \models C'$.

By (Msg Ok), $E, C, E' \vdash \mathbf{ok} : \mathbf{Ok}(S)$.

(3) If $E, E' \vdash P : G$ and $fn(C) \subseteq dom(E)$ then $E, C, E' \vdash P : G$.

By induction on the depth of the derivation of $E, E' \vdash P : G$.

The case for (ProcG Query) uses property (Mon) of the authorization logic.

(ii) (1) If $E, E' \vdash \diamond$, $fn(T) \subseteq dom(E)$ and $x \notin dom(E, E')$, then $E, x{:}T, E' \vdash \diamond$.
By induction on the depth of the derivation of $E, E' \vdash \diamond$.

(2) If $E, E' \vdash M : T$, $fn(T) \subseteq dom(E)$ and $x \notin dom(E, E')$, then $E, x{:}T, E' \vdash M : T$.
By induction on the depth of the derivation of $E, E' \vdash M : T$.

(3) If $E, E' \vdash P : G$, $fn(T) \subseteq dom(E)$ and $x \notin dom(E, E')$, then $E, x{:}T, E' \vdash P : G$.
By induction on the depth of the derivation of $E, E' \vdash P : G$.
The most interesting case is the inductive case for (ProcG Res).
Suppose $E, E' \vdash \mathbf{new}\ y{:}U; P : \mathbf{new}\ y{:}U; G$.
By alpha conversion, consider an instance of y such that $y \notin dom(E, E') \cup \{x\}$.
By hypothesis, $E, E', y{:}U \vdash P : G$.
By hypothesis of the lemma, $x \notin dom(E, E') \cup \{y\}$.
By inductive hypothesis, $E, x{:}T, E', y{:}U \vdash P : G$.
By (ProcG Res), $E, x{:}T, E' \vdash \mathbf{new}\ y{:}U; P : \mathbf{new}\ y{:}U; G$.

□

Lemma 23 (Substitution) *If* $E_1, x{:}T, E_2 \vdash \mathscr{J}$ *and* $E_1 \vdash M : T$ *then* $E_1, E_2\{M/x\} \vdash \mathscr{J}\{M/x\}$.

Proof We split the proof depending on $\mathscr{J}$.

(1) By induction on the depth of the derivation of $E_1, x{:}T, E_2 \vdash \diamond$. Consider the last rule used.

(Env *x*) Suppose $E, y{:}U \vdash \diamond$.
By hypothesis, $E \vdash \diamond$, $fn(U) \subseteq dom(E)$ and $x \notin dom(E)$.
Suppose $E = E_1, x{:}T, E_2$ where $E_2 = E_2', y{:}U$.
By inductive hypothesis, $E_1, E_2'\{M/x\} \vdash \diamond$.
By hypothesis of the lemma, $E_1 \vdash M : T$.
By Lemma 11, $fn(T) \cup fn(M) \subseteq dom(E_1)$.
By definition, $dom(E_1, E_2'\{M/x\}) = dom(E_1, E_2') \setminus \{x\}$.
Applying the substitution, $fn(U\{M/x\}) \subseteq dom(E_1, E_2'\{M/x\})$.
By (Env *x*), $E_1, E_2'\{M/x\}, y{:}U\{M/x\} \vdash \diamond$.

(Env *C*) Similar to the previous case.

(2) By induction on the depth of the derivation of $E_1, x{:}T, E_2 \vdash N : U$. Let $E = E_1, x{:}T, E_2$.

(Msg *x*) Suppose $E \vdash y : E(y)$ because $E \vdash \diamond$ and $y \in dom(E)$.
By point (1), $E_1, E_2\{M/x\} \vdash \diamond$.
We distinguish two cases. Suppose $y \neq x$.

Since $y \neq x$, $y \in dom(E_1, E_2\{M/x\})$.
By (Msg x), $E_1, E_2\{M/x\} \vdash y : E\{M/x\}(y)$.
Suppose instead $y = x$.
By hypothesis of the lemma, $E_1 \vdash M : T$.
By Lemma 22, $E_1, E_2\{M/x\} \vdash M : T$.
Since $y = x$, by definition, $E(y) = T$.
By substitution, $(y : E(y))\{M/x\} = M : T$, and we conclude.

(Msg Ok) Suppose $E \vdash \mathbf{ok} : \mathbf{Ok}(S)$ because $E \vdash \diamond$, $fn(S) \subseteq dom(E)$ and $\forall C \in S.clauses(E) \models C$.
By point (1), $E_1, E_2\{M/x\} \vdash \diamond$.
By definition, $fn(S\{M/x\}) \subseteq dom(E_1, E_2\{M/x\})$.
By $E \vdash \diamond$, $x \notin fn(clauses(E_1))$.
By definition, $clauses(E_1) = clauses(E_1)\{M/x\}$ and therefore $clauses(E_1, E_2\{M/x\} = clauses(E_1, E_2)\{M/x\}$.
By property (Subst) of the authorization logic, $\forall C \in S.clauses(E_1, E_2)\{M/x\} \models C\{M/x\}$.
By (Msg Ok), $E_1, E_2\{M/x\} \vdash \mathbf{ok} : \mathbf{Ok}(S\{M/x\})$.

The other cases are easy, and follow using the inductive hypothesis and point (1).

(3) By induction on the depth of the derivation of $E_1, x{:}T, E_2 \vdash P : G$, in particular using point (1), point (2) and property (Subst) of the authorization logic. □

Lemma 24 (Subject Congruence) *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 $E \vdash P : G$.

(Struct Refl) Suppose $P \equiv P$.

Both (1) and (2) are immediate.

(Struct Symm) Suppose $P \equiv Q$.

By hypothesis, $Q \equiv P$.

Both (1) and (2) follow immediately applying the inductive hypotheses (2) and (1).

(Struct Trans) Suppose $P \equiv R$.

By hypothesis, $P \equiv Q, Q \equiv R$.

Both cases follow easily from transitivity of implication and the inductive hypotheses.

(Struct Res) Suppose **new** $x{:}T; P \equiv$ **new** $x{:}T; P'$.

By hypothesis, $P \equiv P'$.

By hypothesis of (1), $E \vdash$ **new** $x{:}T; P : G$.

By (ProcG Res), $E, x{:}T \vdash P : G'$ where $G =$ **new** $x{:}T; G'$.

By inductive hypothesis, $E, x{:}T \vdash P' : G'' \equiv G'$.

By (ProcG Res), $E \vdash$ **new** $x{:}T; P' :$ **new** $x{:}T; G''$.

By definition of $\equiv$ and (G Sub Res), $G \equiv$ **new** $x{:}T; G''$.

The proof for (2) is symmetric.

(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 19, $fn(G_1) = fn(G_1')$.

By Lemma 20, $E, env(G_1')^{\tilde{x}} \vdash Q : G_2$.

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 Repl) Suppose $!P \equiv !P'$.

By hypothesis, $P \equiv P'$.

By hypothesis of (1), $E \vdash !P : G$.

By (ProcG Rep), $E \vdash P : G$.

By inductive hypothesis, $E \vdash P' : G' \equiv G$.

By (ProcG Rep), $E \vdash !P' : G'$.

The proof for (2) is symmetric.

(Struct Par Zero) Suppose $P \mid \mathbf{0} \equiv P$.

By hypothesis of (1), $E \vdash P \mid \mathbf{0} : G$.

By (ProcG Par), $E, env(G_2) \vdash P : G_1$ and $E, env(G_1) \vdash \mathbf{0} : G_2$ and $fn(P \mid \mathbf{0}) \subseteq dom(E)$ where $G = G_1 \mid G_2$.

By (ProcG Nil), $E, env(G_1) \vdash \mathbf{0} : \mathbf{0}$, $G_2 = \mathbf{0}$ and $G = G_1 \mid \mathbf{0}$.

By definition of $env(\mathbf{0})$, $E \vdash P : G_1$.

By (G Sub Par Zero), $G \equiv G_1$.

The proof for (2) is similar.

(Struct Par Comm) Suppose $P \mid Q \equiv Q \mid P$.

By hypothesis of (1), $E \vdash P \mid Q : G$.

By (ProcG Par), $E, env(G_2) \vdash P : G_1$ and $E, env(G_1) \vdash Q : G_2$ and $fn(P \mid Q) \subseteq dom(E)$ where $G = G_1 \mid G_2$.

By (ProcG Par), $E \vdash Q \mid P : G_2 \mid G_1$.

By (G Sub Par Comm), $G \equiv G_2 \mid G_1$.

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), $E, env(G_2)^{\widetilde{z}} \vdash P \mid Q : G_1$ 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), $E, env(G_2)^{\widetilde{z}}, env(G_4)^{\widetilde{y}} \vdash P : G_3$ and $E, env(G_2)^{\widetilde{z}}, env(G_3)^{\widetilde{x}} \vdash Q : G_4$, $fn(P \mid Q) \subseteq dom(E, env(G_2))$, and $G_1 = G_3 \mid G_4$.

By Lemma 18, $E, env(G_4)^{\widetilde{y}}, env(G_2)^{\widetilde{z}} \vdash P : G_3$ and $E, env(G_3)^{\widetilde{x}}, env(G_2)^{\widetilde{z}} \vdash Q : G_4$.

By (ProcG Par), $E, env(G_3)^{\widetilde{x}} \vdash Q \mid R : G_4 \mid G_2$.

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 22, $E, env(G) \vdash !P : G$ 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 15, $G_1 = G_2$.

By Lemma 21, $E \vdash !P : G_2$.

By (G Sub Idem), $G \equiv G_2$.

(Struct Repl Repl) Suppose $!!P \equiv !P$.

By hypothesis of (1), $E \vdash !!P : G$.

By (ProcG Rep), $E \vdash !P : G$.

The proof for (2) is similar.

(Struct Repl Par) Suppose $!(P \mid Q) \equiv !P \mid !Q$.

By hypothesis of (1), $E \vdash !P \mid Q : G$.

By (ProcG Rep), $E \vdash P \mid Q : G$.

By (ProcG Par), $E, env(G_2) \vdash P : G_1$ and $E, env(G_1) \vdash Q : G_2$, where $G = G_1 \mid G_2$.

By (ProcG Rep), $E, env(G_2) \vdash !P : G_1$ and $E, env(G_1) \vdash !P : G_2$.

By (ProcG Rep), $E \vdash !P \mid !Q : G$.

The proof for (2) is similar.

(Struct Repl Zero) Suppose $!\mathbf{0} \equiv \mathbf{0}$.

By hypothesis of (1), $E \vdash !\mathbf{0} : G$.

By (ProcG Rep), $E \vdash \mathbf{0} : G$, where by (ProcG Nil) $G = \mathbf{0}$.

The proof for (2) is similar.

(Struct Res Par) Suppose $\mathbf{new}\ x{:}T; (P \mid Q) \equiv P \mid \mathbf{new}\ x{:}T; Q$.

By hypothesis, $x \notin fn(P)$.

By hypothesis of (1), $E \vdash \mathbf{new}\ x{:}T; (P \mid Q) : G$.

By (ProcG Res), $E, x{:}T \vdash P \mid Q : G'$ where $G = \mathbf{new}\ x{:}T; G'$.

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), $E, env(G_1)^{\tilde{x}} \vdash \mathbf{new}\ x{:}T; Q : G_2$.

Since $x \notin fn(P)$, by Lemma 16, $E, env(G_2)^{\tilde{y}} \vdash P : G_1$.

By (ProcG Par), $E \vdash P \mid \mathbf{new}\ x{:}T; Q$.

The proof for (2) is similar, using Lemma 22 instead of Lemma 16.

(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 18, $E, x_2{:}T_2, x_1{:}T_1 \vdash P : G$.

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 25 (Subject Reduction) *If $E \vdash P : G$ and $P \rightarrow P'$ 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 $P \rightarrow P'$.

(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 10, $E \vdash \mathbf{in}\ a(x{:}T);P : \mathbf{0}$ is derived by (ProcG Input), and $T = U$ and $E, x{:}U \vdash P : G'$ for some G'.
  By Lemma 23, $E \vdash P\{M/x\} : G'\{M/x\}$.
- 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 10, $E \vdash \mathbf{in}\ a(x{:}T);P : \mathbf{0}$ is derived by (ProcG Input Un), and $T = \mathbf{Un}$ and $E, x{:}\mathbf{Un} \vdash P : G'$ for some G'.
  By Lemma 23, $E, x{:}\mathbf{Un} \vdash P\{M/x\} : G'\{M/x\}$.

(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 23, $E \vdash P\{M/y\} : G'\{M/y\}$.

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 23, $E, y{:}U\{M/x\} \vdash P\{M/x\} : G'\{M/x\}$.

By Lemma 23, $E \vdash P\{M/x\}\{N/y\} : G'\{M/x\}\{N/y\}$.

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 23, $E \vdash P\{N/y\} : G'\{N/y\}$.

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 22, $E, env(G') \vdash Q : G_2$.

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 24 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 24, $E \vdash P' : G_3 \equiv G_2$.

By definition of $\equiv$ and by (G Sub Trans), $G \sqsubseteq G_3$. □

Proof of Lemma 1 (Type Preservation). *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 24 and 25. □

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

Lemma 26 *For any M, if $fn(M) = \{\tilde{x}\}$ then $\tilde{x}{:}\mathbf{Un} \vdash M : \mathbf{Un}$.*

Proof By structural induction on M.

- $(M = x)$ Let $E = x{:}\mathbf{Un}$, where $fn(M) = \{x\}$.

  By (Env *x*) and (Env ∅), $E \vdash \diamond$.

  By (Msg *x*), $E \vdash M : \mathbf{Un}$.

- $(M = \{M\}N)$ Let $E = \widetilde{x}{:}\widetilde{\mathbf{Un}}, \widetilde{y}{:}\widetilde{\mathbf{Un}}, \widetilde{z}{:}\widetilde{\mathbf{Un}}$, where $\{\widetilde{x}\} = fn(M) \cap fn(N)$, $\{\widetilde{y}\} = fn(M) \setminus \{\widetilde{x}\}$, and $\{\widetilde{z}\} = fn(N) \setminus \{\widetilde{z}\}$.

  By inductive hypothesis, $\widetilde{x}{:}\widetilde{\mathbf{Un}}, \widetilde{y}{:}\widetilde{\mathbf{Un}} \vdash M : \mathbf{Un}$ and $\widetilde{x}{:}\widetilde{\mathbf{Un}}, \widetilde{z}{:}\widetilde{\mathbf{Un}} \vdash N : \mathbf{Un}$.

  By Lemma 22, $E \vdash M : \mathbf{Un}$ and $E \vdash N : \mathbf{Un}$.

  By (Msg Encrypt Un), $E \vdash \{M\}N : \mathbf{Un}$.

- $(M = (M,N))$ Let $E = \widetilde{x}{:}\widetilde{\mathbf{Un}}, \widetilde{y}{:}\widetilde{\mathbf{Un}}, \widetilde{z}{:}\widetilde{\mathbf{Un}}$, where $\{\widetilde{x}\} = fn(M) \cap fn(N)$, $\{\widetilde{y}\} = fn(M) \setminus \{\widetilde{x}\}$, and $\{\widetilde{z}\} = fn(N) \setminus \{\widetilde{z}\}$.

  By inductive hypothesis, $\widetilde{x}{:}\widetilde{\mathbf{Un}}, \widetilde{y}{:}\widetilde{\mathbf{Un}} \vdash M : \mathbf{Un}$ and $\widetilde{x}{:}\widetilde{\mathbf{Un}}, \widetilde{z}{:}\widetilde{\mathbf{Un}} \vdash N : \mathbf{Un}$.

  By Lemma 22, $E \vdash M : \mathbf{Un}$ and $E \vdash N : \mathbf{Un}$.

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

- $(M = \mathbf{ok})$ We have $fn(M) = \varnothing$.

  By (Env $\varnothing$), $\varnothing \vdash \diamond$. By (Msg Ok Un), $\varnothing \vdash \mathbf{ok} : \mathbf{Un}$. □

Lemma 27 (Opponent Typability) *For any opponent P, $\widetilde{x}{:}\widetilde{\mathbf{Un}} \vdash P : G$, where $fn(P) \subseteq \{\widetilde{x}\}$.*

Proof The proof is by induction on the structure of the opponent P, which by definition cannot contain queries, and assigns type **Un** to every name. Let $E = \widetilde{x}{:}\widetilde{\mathbf{Un}}$.

- $(P = \mathbf{0})$ By construction, E is well-formed.

  By (ProcG Nil), $E \vdash \mathbf{0} : \mathbf{0}$.

- $(P = Q \mid R)$ By inductive hypothesis, assume $E \vdash Q : G_1$ and $E \vdash R : G_2$, where $env(G_1)^{\widetilde{y}} = \widetilde{y}{:}\widetilde{Un}$ and $env(G_2)^{\widetilde{z}} = \widetilde{z}{:}\widetilde{Un}$.

  By hypothesis, $fn(Q \mid R) \subseteq dom(E)$.

  By Lemma 22, $E, \widetilde{z}{:}\widetilde{Un} \vdash Q : G_1$ and $E, \widetilde{y}{:}\widetilde{Un} \vdash R : G_2$.

  By (ProcG Par), $E \vdash Q \mid R : G_1 \mid G_2$.

- $(P = !P')$ By inductive hypothesis, assume $E \vdash P' : G$.

  By (ProcG Rep), $E \vdash !P : G$.

- $(P = \mathbf{new}\ x{:}\mathbf{Un}; P')$
  By inductive hypothesis (and Lemma 18 or Lemma 22 where needed), $E, x{:}\mathbf{Un} \vdash P' : G$.

  By (ProcG Res), $E \vdash P : \mathbf{new}\ x{:}\mathbf{Un}; G$.

- $(P = \mathbf{in}\ M(x{:}\mathbf{Un}); P')$ Let $\{\widetilde{z}, x\} = \{\widetilde{x}\} \cup \{x\}$.

  By inductive hypothesis, assume $\widetilde{z}{:}\widetilde{\mathbf{Un}}, x{:}\mathbf{Un} \vdash P' : G$.

  By Lemma 26, $E \vdash M : \mathbf{Un}$.

  By (ProcG Input Un), $E \vdash P : \mathbf{0}$.

- $(P = \textbf{out}\ M(N))$ By Lemma 26, $E \vdash M : \textbf{Un}$ and $E \vdash N : \textbf{Un}$.
  By (ProcG Output Un), $E \vdash P : \mathbf{0}$.
- $(P = \textbf{decrypt}\ M\ \textbf{as}\ \{y{:}\textbf{Un}\}N; P')$ Let $\{\widetilde{z}, y\} = \{\widetilde{x}\} \cup \{y\}$.
  By inductive hypothesis, assume $\widetilde{z}{:}\widetilde{\textbf{Un}}, y{:}\textbf{Un} \vdash P' : G$.
  By Lemma 26, $E \vdash M : \textbf{Un}$ and $E \vdash N : \textbf{Un}$.
  By (ProcG Decrypt Un), $E \vdash P : \mathbf{0}$.
- $(P = \textbf{match}\ M\ \textbf{as}\ (N, y{:}\textbf{Un}); P')$ Let $\{\widetilde{z}, y\} = \{\widetilde{x}\} \cup \{y\}$.
  By inductive hypothesis, assume $\widetilde{z}{:}\widetilde{\textbf{Un}}, y{:}\textbf{Un} \vdash P' : G$.
  By Lemma 26, $E \vdash M : \textbf{Un}$ and $E \vdash N : \textbf{Un}$.
  By (ProcG Match Un), $E \vdash P : \mathbf{0}$.
- $(P = \textbf{split}\ M\ \textbf{as}\ (x{:}\textbf{Un}, y{:}\textbf{Un}); P')$ Let $\{\widetilde{z}, x, y\} = \{\widetilde{x}\} \cup \{x, y\}$.
  By inductive hypothesis, assume $\widetilde{z}{:}\widetilde{\textbf{Un}}, x{:}\textbf{Un}, y{:}\textbf{Un} \vdash P' : G$.
  By Lemma 26, $E \vdash M : \textbf{Un}$.
  By (ProcG Split Un), $E \vdash P : \mathbf{0}$.
- $(P = C)$ By construction, $fn(C) = dom(E)$.
  By (Env C), $E, C \vdash \diamond$.
  By (ProcG Fact), $E \vdash P : C$. □

Proof of Lemma 2. *For any opponent P, $\widetilde{x}{:}\widetilde{\textbf{Un}} \vdash P$, where $fn(P) \subseteq \{\widetilde{x}\}$.*

Proof Follows directly from Lemma 27 and Lemma 8. □

B.3.2 Safety and Robust Safety

Lemma 28 (Normal Form) *If $E \vdash P : G$ and $clauses(env(G)^{\widetilde{x}}) = \{C_1, \ldots, C_n\}$ then there exists a P' such that $P \equiv \textbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \ldots \mid C_n \mid P')$.*

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 $E \vdash \textbf{new}\ x{:}T; P : \textbf{new}\ x{:}T; G$.

By hypothesis, $E, x{:}T \vdash P : G$.

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

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, $clauses(env(\textbf{new}\ x{:}T; G)^{x,\widetilde{x}}) = \{C_1, \ldots, C_n\}$.

(ProcG Rep) Suppose $E \vdash !P : G$.

By hypothesis, $E \vdash P : G$.

By inductive hypothesis, $P \equiv \mathbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \ldots \mid C_n \mid P')$ where $\mathit{clauses}(\mathit{env}(G)^{\widetilde{x}}) = \{C_1, \ldots, C_n\}$.

By (Struct Repl) and (Struct Res Par), $!P \equiv \mathbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \ldots \mid C_n \mid (P' \mid !P)) \mid !P \equiv \mathbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \ldots \mid C_n \mid (P' \mid !P))$.

(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, $P \equiv \mathbf{new}\ \widetilde{x}{:}\widetilde{T}; (C_1 \mid \ldots \mid C_n \mid P')$ where $\mathit{clauses}(\mathit{env}(G_1)^{\widetilde{x}}) = \{C_1, \ldots, C_n\}$ and $Q \equiv \mathbf{new}\ \widetilde{y}{:}\widetilde{U}; (C'_1 \mid \ldots \mid C'_m \mid Q')$ where $\mathit{clauses}(\mathit{env}(G_2)^{\widetilde{y}}) = \{C'_1, \ldots, C'_m\}$.

By α-conversion and commutativity, $P \mid Q \equiv \mathbf{new}\ \widetilde{x}{:}\widetilde{T}, \widetilde{y}{:}\widetilde{U}; (C_1 \mid \ldots \mid C_n \mid C'_1 \mid \ldots \mid C'_m \mid (P' \mid Q'))$.

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 $P = P'$. □

Proof of Theorem 2. *If $E \vdash P$ and E is generative then P is safe.*

Proof We need to show that whenever $P \rightarrow^*_{\equiv} \mathbf{new}\ \tilde{x}{:}\tilde{T}; (\mathbf{expect}\ C \mid P')$, we can refactor P' so that $P' \equiv \mathbf{new}\ \tilde{y}{:}\tilde{U}; (C_1 \mid \ldots \mid C_n \mid P'')$, and $\{C_1, \ldots, C_n\} \models C$, with $\{\tilde{y}\} \cap \mathit{fn}(C) = \varnothing$.
By hypothesis, $E \vdash P$.
By Lemma 25 and Lemma 24, if $P \rightarrow^*_{\equiv} \mathbf{new}\ \tilde{x}{:}\tilde{T}; (\mathbf{expect}\ C \mid P')$ then $E \vdash \mathbf{new}\ \tilde{x}{:}\tilde{T}; (\mathbf{expect}\ C \mid P') : G$, for some G.
This must follow from repeatedly applying (ProcG Res) from the premise $E, \widetilde{x}{:}\tilde{T} \vdash \mathbf{expect}\ C \mid P' : G_1$, where $G = \mathbf{new}\ \widetilde{x}{:}\widetilde{T}; G_1$.
This must follow from (ProcG Par) and (ProcG Query), from the premises
(i) $E, \widetilde{x}{:}\tilde{T}, \mathit{env}(G_1)^{\tilde{y}} \vdash \mathbf{expect}\ C : \varnothing$ and
(ii) $E, \widetilde{x}{:}\tilde{T} \vdash P' : G_1$, where $\mathit{fn}(\mathbf{expect}\ C) = \mathit{fn}(C) \subseteq \mathit{dom}(E)$ and $\mathit{clauses}(E, \widetilde{x}{:}\tilde{T}, \mathit{env}(G_1)^{\tilde{y}}) \models C$, and $\{\tilde{y}\} \cap \mathit{fn}(C) = \varnothing$.
Assume, without loss of generality, that $\mathit{clauses}(\mathit{env}(G_1)^{\tilde{y}}) = \{C_1, \ldots, C_n\}$.
By generativity of E and by definition, $\{C_1, \ldots, C_n\} \models C$.
By Lemma 28 on (ii), $P' \equiv \mathbf{new}\ \widetilde{y}{:}\widetilde{U}; (C_1 \mid \ldots \mid C_n \mid P'')$. □

Proof 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 27, $\widetilde{z}{:}\widetilde{\mathbf{Un}} \vdash O : G'$, for some G'.
By Lemma 22, $\widetilde{z}{:}\widetilde{\mathbf{Un}}, \mathit{env}(G) \vdash O : G'$ 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 first 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

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

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

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

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

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

$\mathbf{tuple}\ M\ \mathbf{as}\ (=N);P = \mathbf{match}\ (M,M)\ \mathbf{as}\ (N,y{:}\mathrm{Ty}(M));P$ (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 $T, \mathbf{Ch}(T), \mathbf{Key}(T), (x:T,U)$ 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)

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

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

Lemma 29 *Rule (ProcG Input Der) is admissible.*

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

- $(i = 0)$: We need to show that $E, y{:}\mathbf{Ok}(p(u_1,\ldots,u_n)) \vdash \mathbf{tuple}\ y\ \mathbf{as}\ ({=}\mathbf{ok}); P : \mathbf{0}$.
  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, $E, \widetilde{u}{:}\widetilde{\mathbf{Un}}, y : \mathbf{Ok}(p(u_1,\ldots,u_n)) \vdash P : G$.
  By (ProcG Match) and Lemma 22 we conclude.
- $(i = j+1)$: We need to show that $E, y{:}(u_{n-i+1}{:}\mathbf{Un},\ldots,u_n{:}\mathbf{Un},\mathbf{Ok}(p(u_1,\ldots,u_n)))$ $\vdash \mathbf{tuple}\ y\ \mathbf{as}\ (\underline{u}_{n-i+1},\ldots,\underline{u}_n,{=}\mathbf{ok}); P : \mathbf{0}$.
  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), $\mathbf{tuple}\ y\ \mathbf{as}\ (\underline{u}_{n-i+1},\ldots,\underline{u}_n,{=}\mathbf{ok}); P = \mathbf{split}\ y\ \mathbf{as}\ (u_{n-i+1}{:}\mathrm{Ty}_L(y), y{:}\mathrm{Ty}_R(y)); \mathbf{tuple}\ y\ \mathbf{as}\ (u_{n-j+1}\ldots,\underline{u}_n,{=}\mathbf{ok}); P$.
    By definition, $\mathrm{Ty}_R(y) = (u_{n-j+1}{:}\mathbf{Un},\ldots,u_n{:}\mathbf{Un},\mathbf{Ok}(p(u_1,\ldots,u_n)))$ 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: $\mathrm{pred}(S)$

$\mathrm{pred}(\varnothing) = \varnothing \quad \mathrm{pred}(\{C\} \cup S) = \mathrm{pred}(C) \cup \mathrm{pred}(S) \quad \mathrm{pred}(p(u_1,\ldots,u_n)) = \{p_n\}$
$\mathrm{pred}(L_1,\ldots,L_n) = \bigcup_{i \in 1..n} \mathrm{pred}(L_i) \quad \mathrm{pred}(L_0 :- \widetilde{L}) = \mathrm{pred}(L_0) \cup \mathrm{pred}(\widetilde{L})$
Notation: $\widetilde{L} = L_1,\ldots,L_n$

Extracting Bindings from Literals: $env^{\Sigma}(L_1,\ldots,L_n)$

$env^{\Sigma}(L_1,\ldots,L_n) = env^{\Sigma \cup fv(L_{n-1})}(L_1,\ldots,L_{n-1}), env^{\Sigma \cup fv(L_{n-1})}(L_n)$
$env^{\Sigma}(p(u_1,\ldots,u_n)) = env^{\Sigma}(u_1,\ldots,u_n), y{:}\mathbf{Ok}(p(u_1,\ldots,u_n)) \qquad$ (where y is fresh)
$env^{\Sigma}(u_1,\ldots,u_n) = env^{\Sigma}(u_1,\ldots,u_{n-1}), env^{\Sigma \cup fv(u_1,\ldots,u_{n-1})}(u_n)$

$env^{\Sigma}(X) = X{:}\mathbf{Un}$ 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 30 *Consider a clause $C = L :- L_m, \ldots, L_1$, 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

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

- $i = 0$: $E, C, env^{\Sigma_1}(L_m, \ldots, L_1) \vdash [\![L]\!]^+ : \mathbf{0}$ easily follows from (ProcG Output) and (Infer Fact).
- $i = j + 1$: We are to show $E, C, env^{\Sigma_{i+1}}(L_m, \ldots, L_{i+1}) \vdash [\![L_i, L_j, \ldots, L_1]\!]^{\Sigma_i}[[\![L]\!]^+] : \mathbf{0}$.

  Suppose, without loss of generality, that $L_i = p(u_1, \ldots, u_h)$.

  By definition of encoding,
  $[\![L_i, L_j, \ldots, L_1]\!]^{\Sigma_i}[[\![L]\!]^+] = \mathbf{in}\ p(\underline{u}_1, \ldots, \underline{u}_h, {=}\mathbf{ok}); [\![L_j, \ldots, L_1]\!]^{\Sigma_i \cup fv(L_i)}[[\![L]\!]^+]$.

  By definition of Σ_j, $\Sigma_j = \Sigma_i \cup fv(L_i)$.

  By inductive hypothesis, $E, C, env^{\Sigma_i}(L_m, \ldots, L_i) \vdash [\![L_j, \ldots, L_1]\!]^{\Sigma_j}[[\![L]\!]^+] : \mathbf{0}$.

  By (ProcG Input Der),
  $E, C, env^{\Sigma_{i+1}}(L_m, \ldots, L_{i+1}) \vdash \mathbf{in}\ p(\underline{u}_1, \ldots, \underline{u}_h, {=}\mathbf{ok}); [\![L_j, \ldots, L_1]\!]^{\Sigma_j}[[\![L]\!]^+] : \mathbf{0}$.

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

Proof 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 $E = \widetilde{y}{:}\widetilde{\mathbf{Un}}, \widetilde{p_n}{:}\widetilde{T_{n,p}}$. 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 30 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 31 *Let $L = p(u_1, \ldots, u_n)$ 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 29. □

Lemma 32 *Let $C = L_0 :- L_1, \ldots, L_n$ 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 $[\![C]\!] \mid [\![L_1\sigma]\!]^{+} \mid \ldots \mid [\![L_n\sigma]\!]^{+} \rightarrow^{*}_{\equiv} P \mid ([\![L_0]\!]^{+})\sigma$.*

Proof By definition of encoding,
$[\![C]\!] \mid [\![L_1\sigma]\!]^{+} \mid \ldots \mid [\![L_n\sigma]\!]^{+} \equiv [\![C]\!] \mid [\![L_1, \ldots, L_n]\!]^{\varnothing}[[\![L_0]\!]^{+}] \mid [\![L_1\sigma]\!]^{+} \mid \ldots \mid [\![L_n\sigma]\!]^{+}$. We show, by induction on n, that $([\![L_1, \ldots, L_n]\!]^{\Sigma}[[\![L_0]\!]^{+}])\sigma \mid [\![L_1\sigma\rho]\!]^{+} \mid \ldots \mid [\![L_n\sigma\rho]\!]^{+} \rightarrow^{*}_{\equiv} [\![L_0]\!]^{+}\sigma\rho$ 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 $L_{m+1} = p(u_1, \ldots, u_h)$.
  By definition of encoding, $[\![L_{m+1}, L_1, \ldots, L_m]\!]^{\Sigma}[[\![L_0]\!]^{+}] = Q$ where
  $Q = \mathbf{in}\ p(\underline{u}_1, \ldots, \underline{u}_h, =\mathbf{ok}); [\![L_1, \ldots, L_m]\!]^{\Sigma \cup fv(L_{m+1})}[[\![L_0]\!]^{+}]$.
  By Lemma 31, $Q\sigma \mid [\![L_1\sigma\rho]\!]^{+} \mid \ldots \mid [\![L_m\sigma\rho]\!]^{+} \mid [\![L_{m+1}\sigma\rho]\!]^{+} \rightarrow^{h+1}$
  $([\![L_1, \ldots, L_m]\!]^{\Sigma \cup fv(L_{m+1})}[[\![L_0]\!]^{+}])\sigma\rho \mid [\![L_1\sigma\rho]\!]^{+} \mid \ldots \mid [\![L_m\sigma\rho]\!]^{+}$,
  where $dom(\rho) = fv(L_{m+1})$.
  By inductive hypothesis,
  $([\![L_1, \ldots, L_m]\!]^{\Sigma \cup fv(L_{m+1})}[[\![L_0]\!]^{+}])\sigma\rho \mid [\![L_1\sigma\rho]\!]^{+} \mid \ldots \mid [\![L_m\sigma\rho]\!]^{+} \rightarrow^{*}_{\equiv} [\![L_0]\!]^{+}\sigma\rho$.
  □

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

Lemma 33 *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. Completeness follows by induction on the derivations of $\models$; correctness follows by subject reduction.

Proof 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 $S \models F$.

- ($d = 1$): By hypothesis, $F \in S$. Let $S = S' \cup \{F\}$.
  By definition of encoding, $[\![S]\!] = [\![S']\!] \mid !{[\![F]\!]}^{+} \equiv [\![S]\!] \mid [\![F]\!]^{+}$.
  By definition of $\Downarrow$, $[\![S]\!] \Downarrow_F$.

- ($d = m+1$): By hypothesis, $F = L\sigma$ for some grounding substitution of messages for variables σ and for some clause $C = L :- L_1, \ldots, L_n$ such that $S = S' \cup \{C\}$. Moreover, $S \models L_i\sigma$ for all i, and each $L_i\sigma$ is ground.

  By inductive hypothesis, $[\![S]\!] \Downarrow_{L_i\sigma}$ for all i.

  By definition of $\Downarrow$, $\exists P_i.[\![S]\!] \rightarrow^*_{\equiv} P_i \mid [\![L_i\sigma]\!]^+$ for all i.

  By Lemma 33, for each i there exists P'_i such that $P_i \equiv [\![S]\!] \mid P'_i$.

  By reordering the reductions, we have that $[\![S]\!] \rightarrow^*_{\equiv} Q = [\![S]\!] \mid P''_1 \mid [\![L_1\sigma]\!]^+ \mid \ldots \mid P''_n \mid [\![L_n\sigma]\!]^+$, where each $P'_i = P''_i \mid [\![L_i\sigma]\!]^+$.

  By definition of encoding, $Q = [\![S']\!] \mid [\![L :- L_1, \ldots, L_n]\!] \mid P'_1 \mid [\![L_1\sigma]\!]^+ \mid \ldots \mid P'_n \mid [\![L_n\sigma]\!]^+$.

  By Lemma 32, there exists P' such that $Q \rightarrow^*_{\equiv} P' \mid [\![L\sigma]\!]^+$.

  By definition of $\Downarrow$, $[\![S]\!] \Downarrow_{L\sigma}$.

($\Leftarrow$) 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, $E \vdash S \mid P \mid [\![F]\!]^+$.

Without loss of generality, suppose $F = p(u_1, \ldots, u_n)$.

By definition of encoding, $[\![p(u_1, \ldots, u_n)]\!]^+ = \mathbf{out}\ p(u_1, \ldots, u_n, \mathbf{ok})$ and $T_{p,n} = \mathbf{Ch}(u_1{:}\mathbf{Un}, \ldots, u_n{:}\mathbf{Un}, \mathbf{Ok}(p(u_1, \ldots, u_n)))$.

The judgment $E \vdash S \mid P \mid [\![F]\!]^+$ implies that rule (ProcG Par) has been applied twice, with premises: $E, env(G_2), env(G_3) \vdash S : G_1$ and $E, env(G_1), env(G_3) \vdash P : G_2$ and $E, env(G_1), env(G_2) \vdash [\![F]\!]^+ : G_3$.

By construction, $G_1 = S$, $G_2 = G_3 = \varnothing$, since the process $[\![S]\!]$ contains no statements.

Simplifying, the premises become: $E \vdash S : G_1$ and $E, S \vdash P : \varnothing$ and $E, S \vdash [\![F]\!]^+ : \varnothing$.

The last judgment must follow by rule (ProcG Output) for channel type $T_{p,n}$, and some applications of message rules ending with an instance of (Msg Ok) for type $\mathbf{Ok}(F)$.

Since E is generative we have that $clauses(E) = \varnothing$.

We conclude because the necessary premise of the rule is $S \models F$. □

D Listing of Programme Committee Example

The following shows the sample application from Section 5 as processed by our typechecker and symbolic interpreter. The syntax accepted by our implementation is slightly more verbose than in the paper; we require square brackets around statements, and we require additional round brackets in the syntax of terms.

The symbolic interpreter simulates each of the processes introduced in **trace** statements in a context consisting of the clauses declared by **global** statements, and the process abbreviations declared by **process** statements. Beforehand, the typechecker

tests that each of the processes mentioned in **trace** statements is well-typed in an environment consisting of all typed names and clauses declared by **global** statements, and given the process abbreviations declared by **process** statements. We deem global names of type **Un** to be public, and available to the attacker, whereas we deem global names of channel or key types to be private, and not initially available to the attacker. Suppose we compose each such process with statements of the global clauses, and enclose the result in a series of restrictions for each of the global names with a type other than **Un** (the names pwdb, refereedb, kp, and ka). By Theorem 3, the outcome is robustly safe.

```
global [Report(U,ID,R):−Referee(U,ID),Opinion(U,ID,R)]. // clause A
global [Report(U,ID,R):−PCMember(U),Opinion(U,ID,R)]. // clause B
global [Referee(V,ID) :− Referee(U,ID),Delegate(U,V,ID)]. // clause C

// Section 5.1: Online Delegation, with Local State

global pwdb : Ch((u:Un,
       (Key((v:Un,(id:Un,Ok(Delegate(u,v,id))))),
        Key((id:Un,(report:Un,Ok(Opinion(u,id,report)))))))).
global refereedb : Ch((u:Un,(id:Un,Ok(Referee(u,id))))).

global createReviewer:Un, sendreportonline:Un, delegateonline:Un.
global filereport:Un, filedelegate:Un.
process CreateReviewer() =
 !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) )).

process FileReport() =
 !in filereport(v,e);
  in pwdb(=v,kdv,krv); decrypt e as {id,report,_}krv;
  in refereedb(=v,=id,_); expect Report(v,id,report).
process FileDelegate() =
 !in filedelegate(v,sigd);
  in pwdb(=v,kdv,krv); decrypt sigd as {w,id,_}kdv;
  in refereedb(=v,=id,_); out refereedb(w,id,ok).

global Alice:Un,Bob:Un. // Two reviewers
global Paper058:Un. // A paper identifier
global delta:Un,milestone:Un,breakthrough:Un. // Some grades

// Example 1: Direct reporting via online database of referees
trace CreateReviewer() | FileReport() |
      out createReviewer(Alice) |
      [Referee(Alice,Paper058)] | out refereedb(Alice,Paper058,ok) |
```

```
        out sendreportonline(Alice,Paper058,delta).

// Example 2: Delegation and reporting via online database of referees
trace CreateReviewer() | FileReport() | FileDelegate() |
        out createReviewer(Alice) |
        [Referee(Alice,Paper058)] | out refereedb(Alice,Paper058,ok) |
        out createReviewer(Bob) | out delegateonline(Alice,Bob,Paper058) |
        out sendreportonline(Bob,Paper058,milestone).

// Reviews from PC members, using capabilities

global createPCMember:Un,filepcreport: Un.

process CreatePCMemberAndFilePCReport() =
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)).

// Example 3: PC member registering review via appointment capability
trace CreateReviewer() | CreatePCMemberAndFilePCReport() |
        // PC chair appoints Alice as a PCMember
        out createReviewer(Alice) | new k:Un; out createPCMember(Alice,k) |

        // Alice uses the capability pctoken to register a review
        in k(pctoken); in pwdb(=Alice,_,krAlice);
        [Opinion(Alice,Paper058,milestone)] |
        out filepcreport(Alice,{(Paper058,(milestone,ok))}krAlice,pctoken).

// Section 5.2: Offline Delegation, with Certificate Chains

global [Delegate(U,W,ID):−Delegate(U,V,ID),Delegate(V,W,ID)].
global [Delegate(U,U,ID):−Opinion(U,ID,R)].
global ka : Key((u:Un,(id:Un,Ok(Referee(u,id))))), filedelegatereport:Un.
process FileDelegateReport() =
 !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)).

// Example 4: Offline delegation.
trace CreateReviewer() | FileDelegateReport() |
        // PC chair appoints Alice and Bob as reviewers.
        out createReviewer(Alice) | out createReviewer(Bob) |

        // PC chair uses ka to appoint Alice as reviewer of Paper058
```

```
[Referee(Alice,Paper058)] |
new m1:Un; out m1 ({(Alice,(Paper058,ok))}ka) |

// Alice delegates Paper058 to Bob
in m1 (cAlice:Un); in pwdb(=Alice,kdAlice,_);
[Delegate(Alice,Bob,Paper058)] |
new m2:Un; out m2 ((Alice,({(Bob,(Paper058,ok))}kdAlice,cAlice))) |

// Bob sends in his review
in m2(cBob:Un); in pwdb(=Bob,_,krBob);
[Opinion(Bob,Paper058,milestone)] |
out filedelegatereport(Bob,{(Paper058,(milestone,ok))}krBob,cBob).
```

References

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1–2):3–21, 1998.

[2] M. Abadi. Secrecy by typing in security protocols. *Journal of the 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] D. Duggan. Cryptographic types. In *15th IEEE Computer Security Foundations Workshop*, pages 238–252. IEEE Computer Society Press, 2002.

[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. *Journal of Computer Security*, 11(4):451–521, 2003.

[17] A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300:379–409, 2003.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *16th IEEE Computer Security Foundation Workshop (CSFW'03)*, pages 89–103, 2003.

[24] 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.

[25] R. D. Nicola, G. Ferrari, and R. Pugliese. Programming access control: The KLAIM experience. In *CONCUR 2000—Concurrency Theory*, volume 1877 of *LNCS*, pages 48–65. Springer, 2000.

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

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

[28] P. Samarati and S. de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD 2000*, volume 2171 of *LNCS*, pages 137–196. Springer, 2001.

[29] 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.