Do As I SaY! Programmatic Access Control with Explicit Identities

Andrew Cirillo Radha Jagadeesan Corin Pitcher James Riely

School of CTI
DePaul University
E-mail: {acirillo,rjagadeesan,cpitcher,jriely}@cs.depaul.edu

Abstract

We address the programmatic realization of the access control model of security in distributed systems. Our aim is to bridge the gap between abstract/declarative policies and their concrete/operational implementations.

We present a programming formalism (which extends the asynchronous pi-calculus with explicit principals) and a specification logic (which extends Datalog with primitives from authorization logic). We provide two kinds of static analysis methods to tie implementation to specification. Type checking determines that a program is a sound implementation of policy; i.e., that all granted accesses are safe in the face of arbitrary opponents. Model checking determines a degree of completeness; i.e., that accesses permitted by the policy are actually granted in the implementation.

1. Introduction

This paper focuses on the programmatic realization of the access control model of security [31] in a distributed system. In this model, each object has a reference monitor that mediates requests from a subject: the authorization policy of the object determines whether subject requests are granted.

In a distributed system, it is unreasonable to assume global control of the trust relationships in the system. Rather, each party in the system maintains its own beliefs about trust relationships [13]. The resulting network of trust can be complex, even under the assumption of perfect authentication. For example, subtle notions of delegation must be expressed. Logic-based policy languages are particularly effective at capturing these subtleties. Notable examples have been derived from fragments of many-sorted first-order predicate logic, with sorts for roles and time [27], and from fragments of intuitionist modal logic [2, 24]. In each case, suitable restrictions must be made to enable compilation to an efficient execution engine, such as Datalog.

There is often only an informal relationship between abstract (often declarative) policies and their concrete (often imperative) implementations. To illustrate this, consider protocols developed in the context of identity frameworks such as the Liberty Alliance [1]. These include protocols for federating identities (associating multiple accounts for a given Principal) and for *Single Sign On (SSO)* (using a federated network identity). It is attractive to implement these protocols using widely available programmatic authorization systems (such as Java Authentication and Authorization Service and .NET) where the required access checks are typically commingled with other aspects of code. Such commingling complicates arguments of correctness: we would like to know that the protocol implementation realizes its declarative specification (e.g., that SSO credentials are not used outside of some declared extent). More generally, one is interested in ensuring that the code realizing web-services in such a setting conforms with application-specific policies on creating, using, and updating identities.

We study programmatic implementations of authorization policies in a distributed system, viewing policies as part of the interface specification. We describe static analysis methods to tie the code of a component to its interface, to realize our goal of determining if a system satisfies a given policy.

1.1. Daisy: An Outline

Authorization is fundamentally about specifying permitted interactions between the principals that occur in distributed systems — users, applications, roles, etc. Thus our programming model and specification logic have explicit notions of principal. In this introduction, we provide an informal overview of both the programming and specification formalisms.

Dynamics. The programming formalism of Daisy builds upon the asynchronous pi-calculus. Recall that the pi-calculus describes processes in terms of their ability to send and receive names along communication channels, which are themselves names. Since the pi-calculus supports name generation and name passing, it can describe dynamic network topologies.

To this basic setting, we add a notion of principal. Every pi process is associated with a principal. Inspired by related prior work on locations [8, 28], we sometimes say that the code is *located* at a principal. Each principal has its own local notion of trust [37]. Following the security literature, we model these local beliefs as a security lattice of principals—a principal is (locally) more trustworthy if it is lower in the security lattice. Each local security lattice also provides a (local) interpretation of the constructions of compound principals. In concordance with the distributed context, we do not demand global consistency of local security lattices. The code located at a principal executes in the context of the trust lattice of the principal, using the local trust lattice to answer questions about the local ordering of principals in the trust lattice. The local security lattice evolves dynamically and monotonically, adding new principals and order relations during execution.

We eschew the standard "network is the opponent model" and assume that our messages have integrity, i.e., we are able to identify the sender of messages[1]. We do not address secrecy. This model is well established in the literature [30, 42, 5, 32]. By assuming integrity, we may focus on issues and attacks related directly to authorization, rather than the underlying cryptographic protocols.

Our computational model distinguishes three kinds of messages from a principal A. First, messages may be created from scratch by A — a receiver of the message can detect that the sender is A. Second, messages may be created by a distinct principal B and subsequently be forwarded by A — a receiver of the message can establish that the message from B is coming through unchanged, but via intermediary A. Third, messages may be created by A with an explicit tag, claiming to be from B — a receiver of the message can establish that A claims, without evidence, that the message is from B. The relative trust assigned to these different kinds of message is determined by local policies at the receiver, based on the receivers view of A and B. Principals may also create composite objects whose components are of different kinds.

Our formal treatment uses a sub-calculus of the calculus of compound principals [2, 3] to represent principals. Differences with standard presentations are justified by implementation concerns, which we discuss below.

Statics. We view specifications as annotations to be checked statically: they have no effect on the execution of programs. Our formal development has two ingredients, following [22]: (a) Datalog extended to incorporate authorization logics, and (b) Code annotations to enforce temporal properties.

Recall that a Datalog program is a finite set of Horn clauses, without function symbols. We adapt Datalog to intuitionist authorization logics, permitting predicates to be modified by the modalities of the authorization logic. Intuitively, we associate the principal with each predicate, indicating that the principal uttered the predicate. An important predicate is that which encodes the local trust lattices. Following authorization logics, we use distinct modalities to represent the beliefs of distinct principals, which may be compound. Our technical results reduce the execution of Datalog programs over authorization logics to the execution of regular Datalog programs. This demonstrates the efficient decidability of the properties that are required for static-analysis (e.g., whether a clause can be inferred from a program).

Extended Datalog programs over authorization logics do not encode temporal notions. For example, in SSO, one must determine if an authentication event has happened *before* a given request. To remedy this inadequacy, we follow [22] in incorporating statements and expectations as static annotations of programs. One can view these annotations as correspondence assertions [43], adapted to conjoin specifications of concurrent systems [6]. A *statement* is the analogue of the "assume" in usual program reasoning. It can be used either to record an assertion of global policy or to state assertions about a specific control point. An *expectation* is the analogue of "guarantee" in usual program reasoning. It is a falsifiable claim that a clause is a logical consequence of the current database of assertions.

Our static analysis falls into two categories: type-checking and model-checking.

We provide a type-and-effect system for our programming calculus, where the extended Datalog programs are used as effects. Typing a program establishes two properties. First, in a well-typed program every "expectation" is met. Second, the Datalog specification at any principal of a well-typed program provides a static upper-bound on the local trust lattice, i.e., if the specification does not permit principal A to be ordered below principal B, then A will not be below B in any execution of the program. In the SSO example, this permits us to conclude that the implementation does not provide more rights than those permitted by the policy. We prove *robust safety*, indicating that well-typed programs are safe in the face of arbitrary untyped opponent processes.

We provide a model-checking framework for a subset of programs by translating (a fragment of) our programming calculus into a version of the pi-calculus amenable to model-checking [7]. The fragment requires a fixed finite

1 Following [42, 30], messages/channels have integrity (resp. secrecy) if we know the possible senders (resp. receivers).

number of principals, in addition to restrictions on pi processes imposed by [7]. Specifically, [7] requires that each channel have a unique receiver and satisfy linearity restrictions, thus ensuring bounds on the use of generated names. In the SSO example, this permits us to conclude that the implementation does indeed provide the rights that are permitted by the policy. This is a liveness property, which complements the safety properties guaranteed by the type system.

1.2. Related Work

Authorization logics [3, 2, 24, 23] are the basic foundations of this paper. Our work particularly builds on compound principals and their use for distributed authentication frameworks [42, 30]. Our treatment complements this prior research by focussing on relating implementations to interfaces that specify properties in these logics. More speculatively, our work can be viewed as the first step towards exploring the programming combinators that are suggested by the language of compound principals.

Our approach to assume-guarantee reasoning is inspired by recent work on types for authorization [22]. In [22], there is no explicit notion of identity, and thus authorization is viewed as a cryptographic protocol in the context of the traditional "network is the opponent" model. As a reader of both papers will recognize immediately, we shamelessly incorporate their presentation idioms and technical methods, albeit for a rather different programming model and specification formalism.

Binder [20] is a Datalog formalism that works over authorization logics that is restricted to simple principals. We adapt these techniques to permit compound principals and yet ensure (effective) computability by imposing additional axioms on the basic operation of "quoting" on trust lattices. One can view these extra axioms as reducing the redundancy between the lattice of principals and the proof theory supported by authorization logics. On the other hand, these extra axioms reduce the expressiveness of the calculus of compound principals.

In this area, restrictions of first-order logic that ensure effective computability of specification logics have been well-explored. Our sampling of these references is perforce highly incomplete — the delegation logic [33] and RT framework [34] approach to trust-management, Binder [20] and compositional approaches to access-control [14, 40, 41] that compile down to logic programs fall into this general category. SecPAL [9] is a recent and expressive innovation that belongs in this overall research program. Generally, the focus of this line of work is specification. We focus on the complementary relationship between a given specification and a concrete implementation.

Access control in mobile process languages has been explored in a variety of settings — we consider a sampling of some of these papers. [16] explores mandatory access control in boxed ambients. Klaim (see [11] for a survey) is a Linda-tuple based programming model with a notion of named locations with access control policies that specify the capabilities of the location. A similar approach is taken in [28, 37, 36]. [15] and [19] explore role-based access control in the context of mobile process calculi. These calculi have primitives to activate and deactivate roles: these roles are used to prevent undesired mobility and/or communication. Our formal setting is similar to that of [15], where the "locality" of a process is the name of the principal (or role) on whose behalf the process acts. This style of presentation is only loosely related to other notions of locality in process calculi (see [17] for an extensive survey).

In contrast to this line of work, this paper emphasizes compound principals in dynamics and specifications. Furthermore, the type systems of the above papers are intentionally less general than our specifications, which incorporate general authorization policies.

The use of static analysis techniques to verify security properties is by now well-established, e.g., logic-programming based methods for security protocols [12]. Model-checking methods have been explored for access control in domain specific languages (e.g., [25, 44]) and in the context of systems such as SELinux [26, 29]. We use model-checking methods explored for mobile calculi — see [21] for a survey. We directly use the results of [7], which identify a fragment of the pi-calculus that is amenable to deciding the *control-reachability* problem: i.e., is a certain control point reachable in any execution of a program?

Limitations and Future work. The history and state sensitive aspects of access control are now well-accepted; see [4], temporal extensions to RBAC [10], state-transition approaches to trust management [18], and usage control systems [45]. Our paper treats temporality in the specifications indirectly via the relative placement statements and expectations in code. In future work, we will explore the incorporation of temporal connectives [35] in the specification logic.

This paper provides only a very weak approximation to revocation via garbage collection of unusable names. In future work, we will explore the incorporation of quantitative notions of time to enable the accurate description of *leases*, which facilitate revocation in distributed computing.

Organization of this paper The following section presents the dynamics of the language, which Section 3 illustrates through the SSO example. This is followed by a description of the typing system in Section 4, revisiting the SSO example. Section 5 describes model checking. A longer version of this paper is available at `http://www.teasp.org/daisy`.

2. Syntax and Evaluation

This section describes the operational semantics of Daisy. We first describe the properties expected of the calculus of compound principals. We then describe terms, local security orders, and processes.

2.1. Calculus of Compound Principals

The language of terms, A, B, C, includes atomic principals, the nullary constructors $\mathbf{0}$, $\mathbf{1}$ and del, and the binary constructors $\wedge$ and $|$. These are interpreted as a calculus of compound principals. For a detailed treatment of the intuitions underlying compound principals, we refer to the original sources [3, 42, 30].

We define a lattice ordering $A \Rightarrow B$ indicating that A is more trustworthy than B. (Papers emphasizing secrecy often use the dual ordering.) Thus $\mathbf{0}$ is the most trustworthy principal, $\mathbf{1}$ the least; del is used to encode delegation. Following [2, 3], conjunction ($\wedge$) is a meet in the lattice of principals; the quotation operator ($|$) is associative, in addition to being monotone and multiplicative in each argument. (Following standard equational presentations of lattices, one may think of $A \Rightarrow B$ as shorthand for the equality of $A \wedge B$ and A.)

We additionally take quotation ($|$) to be commutative, idempotent and extensive. Further, we identify the speaks-for relation with the lattice order; thus "$\Rightarrow$" can be read as "speaks-for". These extra axioms facilitate the finiteness principle of Remark 1. For further discussion, see Section 4.1.

The following axioms define the principal order, where $\Leftrightarrow$ is used to abbreviate bidirectional axioms.

Lattice Axioms $(A \Rightarrow B)$

$A \wedge A \Leftrightarrow A$	$\wedge$ Idempotent
$A \wedge B \Leftrightarrow B \wedge A$	$\wedge$ Commutative
$A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge C$	$\wedge$ Associative
$A \wedge \mathbf{1} \Leftrightarrow A$	$\wedge$ Absorptive
$A \wedge \mathbf{0} \Leftrightarrow \mathbf{0}$	$\wedge$ Bound
$A \,\|\, A \Leftrightarrow A$	$\|$ Idempotent
$A \,\|\, B \Leftrightarrow B \,\|\, A$	$\|$ Commutative
$A \,\|\, (B \,\|\, C) \Leftrightarrow (A \,\|\, B) \,\|\, C$	$\|$ Associative
$A \,\|\, \mathbf{0} \Leftrightarrow A$	$\|$ Absorptive
$A \,\|\, \mathbf{1} \Leftrightarrow \mathbf{1}$	$\|$ Bound
$A \Rightarrow A \,\|\, B$	$\|$ Extensive
$A \,\|\, (B \wedge C) \Leftrightarrow (A \,\|\, B) \wedge (A \,\|\, C)$	$\|$-$\wedge$ Distributive

From these axioms one can derive that $\wedge$ and $|$ are monotone in $\Rightarrow$. We use the following abbreviation [3].

$$A \mathbin{for} B \triangleq (A \wedge \mathsf{del}) \,|\, B$$

for is reflexive, monotone in both arguments, and preserves more trust than quotation ($|$). The original source (B) is always more trustworthy than a delegated source ($A \mathbin{for} B$); however, delegation via $\mathbf{0}$ does not decrease trust.

Some Derived Facts

$A \mathbin{for} B \Rightarrow A \,\|\, B$	*for*-$\|$ Strength
$B \Rightarrow A \mathbin{for} B$	*for* Extensive
$\mathbf{0} \mathbin{for} A \Leftrightarrow A$	*for* Left Absorptive
$A \mathbin{for} A \Leftrightarrow A$	*for* Reflexive
$A \mathbin{for} C \Rightarrow B \mathbin{for} C$ if $A \Rightarrow B$	*for* Left Monotone
$C \mathbin{for} A \Rightarrow C \mathbin{for} B$ if $A \Rightarrow B$	*for* Right Monotone
$A \mathbin{for} (B \mathbin{for} C) \Rightarrow (A \mathbin{for} B) \mathbin{for} C$	*for* Semiassociative

Remark 1. For any finite lattice $\mathscr{L}$ of atomic principals, there is a finite lattice of principals that interprets the quoting combinator freely, such that the only equations that hold are those induced by $\mathscr{L}$ and the entailment axioms given above. (We elide the standard formalization as a free construction in the vocabulary of category theory.) The proof follows the observation that the the axioms on the quoting combinator coincide with those of the Hoare powerdomain [38].

We sketch the proof here. Define a partial order with carrier as the set of all finite subsets of $\mathscr{L}$. View a finite subset, say $\{A_1, \ldots, A_n\}$, as standing for $A_1 \,|\, A_2 \,|\, \cdots \,|\, A_n$. $S_1 \leq S_2$ iff $(\forall A \in S_1)\ (\exists B \in S_2) A \Rightarrow B$. This is a complete lattice with the quoting combinator interpreted as union and conjunction given by $S_1 \wedge S_2 = \{A \wedge B \mid A \in S_1, B \in S_2\}$. $\square$

2.2. Terms

We describe the vocabulary of terms, which represent the values that can be created and sent during computation.

Terms

a, b, c	Atomic Principals
n, m, ℓ	Names
x, y, z	Variables
$\eta ::= a \mid n \mid x$	Identifiers
$A, B, C, M, N, L ::=$	Terms
$\quad \eta$	Identifier
$\quad \mathsf{del} \mid \mathbf{0} \mid \mathbf{1} \mid A \,\|\, B \mid A \wedge B$	Principal
$\quad N(\vec{M})$	Labeled Tuple
$\quad \mathsf{sig}A(M) \mid \mathsf{tag}A(M)$	Signature, Tag
$\quad M.\mathsf{val} \mid M.\mathsf{src}$	Value, Source

We presume mutually disjoint syntactic categories for atomic principals, a, b, c, names, n, m, ℓ, and variables, x, y, z. Names are used both as *labels* and as communication channels. We use ℓ for names used a labels and n–m for names used as channels.

To improve readability, we use A, B, C for terms interpreted as principals and M, N, L for terms in other contexts.

Atomic principals are used to identify code. Non-atomic principals occur in policies and in terms but do not identify code. Principals were discussed in the previous section.

Tuples $N(\vec{M})$ are labeled by a name N. Tuple labels are used in matching; for example, the term $\ell(M,N)$ matches the pattern $\ell(x,y)$.

The signature term $\mathsf{sig}A(M)$ represents a term with integrity: the origin is guaranteed to be the principal A. One can imagine the straightforward use of digital signature schemes to efficiently realize this abstraction; our notation acknowledges this potential implementation. The term $\mathsf{tag}A(M)$ on the other hand is a term whose putative origin is A: the trust placed in this term depends on the application context.

The term $M.\mathsf{val}$ indicates the *value* of M, ignoring signatures and tags, whereas $M.\mathsf{src}$ indicates the *source* of M, ignoring its value. val can be used to forget the provenance of a term, as for example in an anonymizer.

A *ground* term contains no variables. We treat ground terms up to an equational algebra on terms, which defines val and src. Let $\simeq$ be the smallest congruence on ground terms that satisfies the following[2].

Ground Term Equivalence

$M.\mathsf{val} \simeq N.\mathsf{val}$	if $M = \mathsf{sig}B(N)$ or $M = \mathsf{tag}B(N)$
$M.\mathsf{val} \simeq M$	otherwise
$M.\mathsf{src} \simeq M.\mathsf{src}(\mathbf{0})$	
$M.\mathsf{src}(A) \simeq A\,\mathit{for}\,(N.\mathsf{src}(B))$	if $M = \mathsf{sig}B(N)$
$M.\mathsf{src}(A) \simeq A \mid (N.\mathsf{src}(B))$	if $M = \mathsf{tag}B(N)$
$M.\mathsf{src}(A) \simeq A$	

Example 2. Note that if M is an name, principal or tuple then $M.\mathsf{val} \simeq M$ and $M.\mathsf{src} \simeq \mathbf{0}$. Further note that $M.\mathsf{src}.\mathsf{val} \simeq M.\mathsf{src}$ and $M.\mathsf{val}.\mathsf{src} \simeq \mathbf{0}$ for any ground term M.

The terms $\mathsf{sig}B(\mathsf{sig}A(n))$ and $\mathsf{sig}B(\mathsf{tag}A(n))$ are both signed by B. The first is forwarded from B, whereas the second is tagged by A. Both equal n under val; however, src distinguishes them. Because $\mathbf{0}$ is a left zero of *for* and $\mid$, we have

$$\begin{aligned} \mathsf{sig}A(n).\mathsf{src} &\simeq A\,\mathit{for}\,\mathbf{0} \Leftrightarrow A \\ \mathsf{tag}A(n).\mathsf{src} &\simeq A \mid \mathbf{0} \quad \Leftrightarrow A \end{aligned}$$

and thus

$$\begin{aligned} \mathsf{sig}B(\mathsf{sig}A(n)).\mathsf{src} &\simeq \Leftrightarrow B\,\mathit{for}\,A \\ \mathsf{sig}B(\mathsf{tag}A(n)).\mathsf{src} &\simeq \Leftrightarrow B \mid A. \end{aligned}$$

In this way, the provenance of the quoted message can be established. □

The lattice axioms prove that $[(A \mid B) \wedge B] \Rightarrow (A\,\mathit{for}\,B)$. Consider a message $\mathsf{sig}B(n)$ sent by A. The reference implementation of sig using digital signatures clearly satisfies $A \mid B$, since the message is coming from A quoting B. It also satisfies B since the digital signature vouchsafes for B. Thus, the reference implementation of sig is sound with respect to trustworthiness.

2.3. Local Security Order

The ordering of principals can vary from site to site. The calculus of compound principals (Section 2.1) is lifted to terms to define a *local security order* at each atomic principal. A collection of formulas, $a\langle\vec{s}\rangle$, reflects the policies and acquired beliefs of atomic principal a, where each s_i is an *order formula* $M \Rightarrow N$.

Order Formulas and Entailment

$s,t ::= M \Rightarrow N$

$\vec{s} \Vdash A \Rightarrow B$ if $A \Rightarrow B$
$\vec{s} \Vdash A \Rightarrow B$ if $(A \Rightarrow B) \in \vec{s}$
$\vec{s} \Vdash A \Rightarrow B$ if $fv(A) = fv(B) = \emptyset$ and
  $A.\mathsf{val} \simeq A'$ and $B.\mathsf{val} \simeq B'$ and $\vec{s} \Vdash A' \Rightarrow B'$
$\vec{s} \Vdash A \Rightarrow B$ if $\vec{s} \Vdash A \Rightarrow C$ and $\vec{s} \Vdash C \Rightarrow B$
$\vec{s} \Vdash A \wedge A' \Rightarrow B \wedge B'$ if $\vec{s} \Vdash A \Rightarrow B$ and $\vec{s} \Vdash A' \Rightarrow B'$
$\vec{s} \Vdash A \mid A' \Rightarrow B \mid B'$ if $\vec{s} \Vdash A \Rightarrow B$ and $\vec{s} \Vdash A' \Rightarrow B'$

These rules quotient the lattice by the congruence generated by $\vec{s}$. The first rule injects the lattice axioms into entailment. The second allows the use of assumptions. The third interprets terms up to ground term equivalence. The remaining rules encode transitivity and congruence.

The definition validates judgments such as

$$x.\mathsf{src} \Rightarrow y,\ y \Rightarrow A,\ y \Rightarrow B \ \Vdash\ x.\mathsf{src} \Rightarrow A \mid B$$

and

$$A \mid B \Rightarrow C \ \Vdash\ \mathsf{sig}A(\mathsf{tag}B(n)).\mathsf{src} \Rightarrow C.$$

2.4. Processes and Configurations

The basic entity of computation is a *process*, or *thread*.

Processes

Z	Process Variables
$P,Q,R ::=$	Processes (Threads)
$0 \mid P \mid Q \mid \mu Z.P \mid Z$	Composition, Recursion
$\mathsf{new}\,n{:}T.P \mid \mathsf{new}\,a\ \mathsf{with}\,P$	Restriction
$M!N \mid M?x{:}T.P$	Communication
$\mathsf{match}\ M\ \mathsf{as}\ N(\vec{x}).P$	Match
$\mathsf{learn}\,s.P$	Learn Order Formula
$\mathsf{check}\ s\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Check Order Formula
$\mathbb{C} \mid \mathsf{expect}\ \mathbb{C}$	Correspondence

2 One could lift ground term equivalence to open terms simply by restricting the axioms to closed terms.

We observe the normal scope rules for pi calculi[3].

Definition 3. We write *fn*(P) for the set of *free identifiers* in P, and similarly for other syntactic categories. Likewise, write *fv*(P) for the set of *free variables* in P. Write $P\{x := M\}$ for the capture avoiding substitution of M for x in P. As usual, we identify syntax up to renaming, drop types when uninteresting, and assume that occurrences of process variables are guarded by input. □

Threads incorporate primitives from the asynchronous pi-calculus with pairs. These include composition, recursion, restriction, output, input, and match. The match construct is blocking. Computation of "match M as $\ell(\vec{x}).P$" proceeds if M is a tuple of arity $|\vec{x}|$ labeled with ℓ; for example, $\ell(n,m)$ matches $\ell(x,y)$, but fails to match $n(x,y)$ or $\ell(x)$.

The learn primitive adds information to the local security order, which the check primitive may query.

Correspondences are used in the type system, as discussed in Section 4; they have no effect on dynamics and thus will be ignored for the rest of this section.

Running processes are collected into *configurations*.

Configurations

G,H ::=	Configurations
$\mathsf{0} \mid G \mid H$	Composition
$\mathsf{new}\, n{:}T.G \mid \mathsf{new}\, a.G$	Restriction
$a[P]$	Located Process
$a\langle\vec{s}\rangle$	Located Security Lattice

Composition and restriction in the configuration language are related to the analogous constructs in the process language by structural rules, discussed below.

Each thread $a[P]$ of a configuration is located at a unique atomic principal a. Any number of threads may be located at the same atomic principal.

Each atomic principal has an associated local security order $a\langle\vec{s}\rangle$. The check and learn primitives operate on this local order. As stated before, the security orders of different principals are unrelated. We assume that each atomic principal has at most one local order.

Definition 4. A configuration is *well-formed* if it contains at most one trust lattice $a\langle\vec{s}\rangle$ for each atomic principal a. □

In the sequel, we assume that all configurations are well-formed. To make use of learn and check, an atomic principal must therefore have exactly one local security order.

Initial configurations may contain any number of tags; however, sigs are generated at runtime.

3 "$\mu Z.P$" binds Z; "$\mathsf{new}\, n{:}T.P$" binds n; "$\mathsf{new}\, a$ with P" binds a; "$M?x{:}T.P$" binds x; and "match M as $N(\vec{x}).P$" binds $\vec{x}$. In each case, the scope is P.

Definition 5. A configuration is *initial* if it contains no instance of sig. □

This initiality restriction mirrors initial key distribution conditions in the formal analysis of cryptographic protocols. It ensures that signatures are unforgeable.

2.5. Evaluation

Structural equivalence relates configurations that differ only in the order of static combinators (composition and restriction).

Structural Equivalence $(G \equiv H)$

$\mathsf{0} \mid G \equiv G$	
$G \mid H \equiv H \mid G$	
$G \mid (H \mid F) \equiv (G \mid H) \mid F$	
$G \mid \mathsf{new}\,\eta.H \equiv \mathsf{new}\,\eta.(G \mid H)$	if $\eta \notin fn(G)$
$\mathsf{new}\,\eta.G \equiv G$	if $\eta \notin fn(G)$
$G \mid H \equiv G' \mid H$	if $G \equiv G'$
$\mathsf{new}\,\eta.G \equiv \mathsf{new}\,\eta.G'$	if $G \equiv G'$

The structural equivalence is standard. It encodes the monoid laws of composition and the extrusion and garbage collection laws of restriction.

The evaluation rules describe the evolution of configurations over time. We describe evaluation in two tables. The first describes the behaviour of processes with respect to static combinators.

Evaluation—Structural Rules $(G \to H)$

$a[\mathsf{0}] \to \mathsf{0}$	
$a[P \mid Q] \to a[P] \mid a[Q]$	
$a[\mu Z.P] \to a[P\{Z := \mu Z.P\}]$	
$a[\mathsf{new}\, n.P] \to \mathsf{new}\, n.a[P]$	
$a[\mathsf{new}\, b \text{ with } P] \to \mathsf{new}\, b.(b[P] \mid b\langle a \Rightarrow b\rangle)$	
$G \to G'$	if $G \equiv H \to H' \equiv G'$
$G \mid H \to G' \mid H$	if $G \to G'$
$\mathsf{new}\,\eta.G \to \mathsf{new}\,\eta.G'$	if $G \to G'$

The structural evaluation rules relate static combinators of the process language to those of the configuration language. For example, $a[P \mid Q] \equiv a[P] \mid a[Q]$. The treatment of recursion through unfolding of process variables is standard.

The evaluation rule for new principals establishes a local security order for the new principal, preserving well-formedness and enabling it to use learn and check. The local lattice states that the new principal believes that its parent is at least as trustworthy as itself. (Because the new principal itself states this, it becomes a globally acknowledged fact—see Remark 13.)

The second table of evaluation describes communication, matching, learn and check.

Evaluation—Reduction Rules $(G \to H)$

$a[M!N] \mid b[M'?x.P] \to b[P\{x := \mathsf{sig}\,a(N)\}]$
  if $M.\mathsf{val} \simeq M'.\mathsf{val} \simeq n$, for some n
$a[\mathsf{match}\; M\; \mathsf{as}\; L(\vec{x}).P] \to a[P\{\vec{x} := \mathsf{tag}\,B(\vec{N})\}]$
  if $M.\mathsf{val} \simeq L'(\vec{N})$, and $M.\mathsf{src} \simeq B$,
  and $L.\mathsf{val} \simeq L'.\mathsf{val}$, and $|\vec{x}| = |\vec{N}|$
$a[\mathsf{learn}\, t.P] \mid a\langle\vec{s}\rangle \to a[P] \mid a\langle\vec{s},t\rangle$
$a[\mathsf{check}\; t\; \mathsf{then}\; P\; \mathsf{else}\; Q] \mid a\langle\vec{s}\rangle \to a[P] \mid a\langle\vec{s}\rangle$ if $\vec{s} \Vdash t$
$a[\mathsf{check}\; t\; \mathsf{then}\; P\; \mathsf{else}\; Q] \mid a\langle\vec{s}\rangle \to a[Q] \mid a\langle\vec{s}\rangle$ if $\vec{s} \nVdash t$

The rule for communication is notable in two respects. First, the source of channel names is ignored; that is, channel names are considered modulo val. Thus, $a[\mathsf{tag}A(n)!M] \mid b[\mathsf{sig}B(n)?x.P]$ evaluates to $b[P]$ with a suitable substitution. Second, the source of a message is recorded in the recipient. Thus, the substitution generated by preceding example is $b[P\{x := \mathsf{sig}\,a(M)\}]$. The source (in this case a) is unforgeably recorded in the receiving process. As initial configurations (Definition 5) evaluate, terms carry sigs to indicate their provenance.

The evaluation rule for match M as $\ell(\vec{x})$ checks that the value of M is a tuple of arity $|\vec{x}|$ labeled with ℓ. If these conditions hold, then the match succeeds. In the consequent, the elements of the tuple are tagged with the source of M, so as not to lose the provenance of the data. The use of tag rather than sig for this purpose is motivated by the reference implementation in terms of digital signatures — A cannot in general create $\mathsf{sig}B()$. This is further discussed in Remark 7.

The learn and check primitives allow interaction between a thread and the local security order. learn is used to add new order relations, monotonically, to the local order. check is used to query the local order dynamically.

Remark 6. From the reflexivity of *for*, $\mathsf{sig}\,a(N).\mathsf{src} \Leftrightarrow \mathsf{sig}\,a(\mathsf{sig}\,a(N)).\mathsf{src}$. Thus, multiple reforwardings of a message by a principal A to itself are both useless and harmless. □

Remark 7. If $\mathsf{sig}\,a(N)$ occurs as a subterm of a configuration reachable from an initial configuration, then it must be that a thread located at a communicated N at some point in the past. This intuition can be formalized by considering traces where communication reductions are annotated with the substitution performed in the receiver.

For example, $a[n!N] \mid b[n?x.0]$ has trace $\mathsf{sig}\,a(N)$ to $b[0]$. Suppose that s is such a trace of an initial configuration G, reaching H after some number of evaluation steps, including an arbitrary number of communication steps ($G \stackrel{s}{\Rightarrow} H$). Then if $\mathsf{sig}\,a(N)$ is a subterm of H, it must be the case that $\mathsf{sig}\,a(N)$ appears in s, indicating that a itself sent N in some prior communication. □

Remark 8 (Conventions). In many cases, the label on a tuple is uninteresting. We therefore presuppose a set of standard labels zero, one, two, etc, indicating the cardinality of the tuple. We elide these standard labels in both terms and patterns, writing simply "(M,N)" rather than "$\mathsf{two}(M,N)$". We also use the following abbreviations.

$$*n?x.P \triangleq \mu Z.n?x.(P \mid Z)$$
$$n!M.P \triangleq P \mid n!M$$
$$n!(\mathsf{new}\,m{:}T).P \triangleq \mathsf{new}\,m{:}T.(P \mid n!m)$$

In multiline programs, write $\mid P \mid Q$ for $P \mid Q$. We use sans serif in examples, to distinguish variables from metavariables (which appear in italics); keywords are written in boldface. □

3. Encoding Single Sign On (SSO)

We consider the following simplified use case from SSO: a user process running as principal uid is attempting to access a protected resource res at a service provider running as sp. We will adopt the policy that only members of institution inst may access res.

Since this is an SSO protocol, uid is asked to establish its identity only if it is unknown already. In the case that sp grants access after the initial request from uid, the message sequence is as follows.

uid ⟶ sp : sp-req!(**new** yes, **new** no)
uid ⟵ sp : yes! -- *access to* res *granted*

uid sends a a request to sp on channel sp-req, passing two new continuation channels. If a response is heard on the first of these continuations then the operation was successful and access to res has been granted.

In the case that sp initially refuses access, uid contacts srv to get a certificate vouching for its identity. uid then forwards the certificate to sp and retries its initial request.

The certificate indicates that the server believes that uid belongs to institution inst. In this example, however, the server signs the certificate as srv | ip, indicating that srv does not itself vouch for the claim, but rather is quoting another identity provided ip. In order to provide access, sp must believe that certificates forwarded from srv | ip may speak as authorized identity providers.

In this case, successful access proceeds as follows.

uid ⟶ sp : sp-req!(**new** yes_1, **new** no_1)
uid ⟵ sp : no_1! -- *access to* res *denied*
uid ⟶ srv : ip-req!(**new** c)
uid ⟵ srv : c!(**tag** ip(okcert(uid, inst)))
uid ⟶ sp : sp-auth!(**sig** srv(**tag** ip(okcert(uid, inst))),
  new yes_2, **new** no_2)
uid ⟵ sp : yes_2! -- *certificate accepted by* sp
uid ⟶ sp : sp-req!(**new** yes_3, **new** no_3)
uid ⟵ sp : yes_3! -- *access to* res *granted*

After the initial refusal by sp, uid sends a request to srv on ip-req with continuation channel c, and srv replies with a certificate. Crucial here is the form of the certificate created by srv: this is a pair (uid,inst) labeled by okcert. The label is used to communicate the intent of the certificate via types, as discussed in Section 4.4; we ignore it here. Before sending the certificate, srv tags it by ip, indicating its qualified endorsement of the claims therein. As per the definition of evaluation, the certificate received and then forwarded by uid is signed by srv. Thus, the message received by sp on sp-auth has the form

$$\textbf{sig}\,\mathsf{uid}(\underbrace{\textbf{sig}\,\mathsf{srv}(\textbf{tag}\,\mathsf{ip}(\mathsf{okcert}(\mathsf{uid},\mathsf{inst})))}_{\text{Forwarded from srv}},\ldots)$$

sp accepts the certificate, notifying uid on the yes continuation, at which point uid repeats its initial request, which is now granted.

3.1. Encoding the SSO Protocol

With this introduction, we now describe the implementation, narrating the second use case above. The user configuration has the following form.

```
uid[μloop.
  sp-req!(new yes1,new no1).
  | yes1?  -- access granted
  | no1?  -- access denied
     ip-req!(new c).
     c?cert.
     sp-auth!(cert,new yes2,new no2).
      | yes2? loop
      | no2?  -- go to another id provider or give up ]
```

If the initial request to sp on sp-req fails, then the user issues a certificate request on ip-req and forwards the result to sp on sp-auth. If the certificate is accepted, then the user repeats its initial request on sp-req via loop. (For simplicity, we have written the code assuming that ip-req is always granted.)

We now present the code running at sp and srv, starting with the local security order at sp.

```
sp<inst ⇒ res, 1 for (srv | ip) ⇒ authorized-ip>
```

In the example execution, sp initially believes that members of inst may access res, and that certificates from srv | ip are authorized to provide identity information for inst, even when forwarded via an unknown sequence of intermediaries. A proof that this policy achieves the desired effect follows from the monotonicity and semi-associativity of *for*. (Although we treat this as the initial policy of sp, such policies may be built dynamically following the techniques discussed in this example.)

The code servicing sp-req is as follows.

```
sp[*sp-req?x.
  match x as (yes,no).
   check x.src ⇒ res then yes! else no!]
```

After receiving the message from uid, x is bound to **sig** uid(yes,no), and thus x.src is (equivalent to) uid. With only the initial facts, i.e., the user has not been validated earlier, the test uid ⇒ res fails.

At this point, uid contacts srv on ip-req to get a certificate that will prove its identity to sp. The local policy and code for srv are as follows.

```
srv<uid ⇒ inst>
srv[*ip-req?x.
  check x.src ⇒ inst
    then x!(tag ip(okcert(x.src,inst)))]
```

After receiving the message from uid, x is bound to **sig** uid(c), and thus x.src is uid. Since srv believes that uid belongs to inst, it replies with a certificate that it is willing to sign as srv | ip.

uid now forwards the certificate from srv to sp on sp-auth. The message is received as follows.

```
sp[*sp-auth?x.
  match x as (cert,yes,no).
  check cert.src ⇒ authorized-ip
   then match cert as okcert(z_uid,z_inst).
        learn z_uid ⇒ z_inst. yes!
   else no!]
```

After receiving the message from uid and performing the match, cert is bound to

$$\textbf{tag}\,\mathsf{uid}(\textbf{sig}\,\mathsf{srv}(\textbf{tag}\,\mathsf{ip}(\mathsf{okcert}(\mathsf{uid},\mathsf{inst}))))$$

and thus cert.src is uid *for* (srv | ip). Recall that the local policy of sp specifies that **1** *for* (srv | ip) ⇒ authorized-ip. Therefore the check cert.src ⇒ authorized-ip succeeds. The contents of the certificate are then recovered using match, and added to the local order of sp using learn. Subsequent requests from uid on sp-req will grant access to res.

Remark 9. It is worth noting that correspondence between the check (x.src ⇒ inst) in ip-req and the learn ($z_{\mathsf{uid}} \Rightarrow z_{\mathsf{inst}}$) in sp-auth is entirely programmatic, and therefore prone to error. The type system makes explicit such implicit correspondences, eliminating potential programming errors. □

3.2. Variations

Full identity-management protocols permit variations in the flow of information. For example, the certificate may be sent directly from srv to sp. This can be accommodated in our example by simple changes to the code for uid and srv, without modifying the local orders. Interestingly, we can force such a change by modifying the local policy of sp to:

$$\mathsf{sp}\texttt{<}\mathsf{inst} \Rightarrow \mathsf{res},(\mathsf{srv}\,|\,\mathsf{ip}) \Rightarrow \mathsf{authorized\text{-}ip}\texttt{>}$$

This forces the protocol to directly communicate the authorization token from srv to sp.

Rather than perform SSO operations as itself, the user uid may perform them using a fresh identity anon to which it delegates rights. In the simplest case, uid allows anon to speak for uid with respect to srv. This can be achieved by adding a new channel ip-auth and coding the necessary communication.

We start by defining some syntactic sugar. The new principal anon is known only to itself and therefore has no rights in the system. We define "new b at a with $P.Q$" as a "symmetric" form of atomic principal creation, in which child and parent agree on their relation in the principal order.

$$\begin{array}{l}
\mathsf{new}\,b\ \mathsf{at}\ a\ \mathsf{with}\,P.Q \triangleq \\
\quad \mathsf{new}\,n. \\
\quad\ \ |\ \mathsf{new}\,b\ \mathsf{with}\,n\,\texttt{!}\,\mathsf{tag}\,\mathsf{okcert}(a,b)\,.\ P \\
\quad\ \ |\ (n?x.\ \mathsf{match}\ x\ \mathsf{as}\ \mathsf{okcert}(y,z)\,. \\
\quad\quad\ \mathsf{check}\ x.\mathsf{src} \Rightarrow z\ \mathsf{then}\ \mathsf{learn}\,y \Rightarrow z.\ Q\{b := z\})
\end{array}$$

Evaluation proceeds as follows.

$$\begin{array}{l}
a\texttt{<}\vec{s}\texttt{>}\,|\,a\,[\mathsf{new}\,b\ \mathsf{at}\ a\ \mathsf{with}\,P.Q] \ \rightarrow^{*} \\
\quad \mathsf{new}\,b.(b\texttt{<}a \Rightarrow b\texttt{>}\,|\,b\,[P]\,|\,a\texttt{<}\vec{s}, a \Rightarrow b\texttt{>}\,|\,a\,[Q]\,)
\end{array}$$

This definition allows us to easily describe systems in which parent and child principals have mutual knowledge.

The uid code is moved to anon, and uid informs srv of the new principal.

```
uid[new anon with (μloop. -- uid code from before).
  ip-auth!okcert(anon,uid)]
```

The user creates the fresh principal name (anon) and registers it with the srv, telling srv that anon speaks for uid. For this to work, of course, srv must be willing to accept new order relations.

```
srv[ip-auth?x.
  match x as okcert(y,z).
  check x.src ⇒ z then learn y ⇒ z]
```

The server will allow anyone to say that others speak for them.

When the modified code for uid and srv are added to the system, the uid process carries on as before, but with identity anon instead of uid. After srv learns that anon $\Rightarrow$ uid (and therefore anon $\Rightarrow$ inst) it will gladly issue the certificate **sig** srv(okcert(anon,inst)) which is valid for authentication, but does not mention uid.

4. Types

We present a type-and-effect system where effects are extended Datalog programs. Our formal presentation closely follows [22]. We begin this section with a review of the underlying authorization logic and an extended Datalog built on top of authorization logic. Next, we discuss the typing system and illustrate with code fragments drawn from the SSO example.

4.1. Background: Authorization Logics

We refer the reader to [24, 2] for the intuitions underlying authorization logics. Our presentation satisfies more commutativity properties than [24] in the proof theory. In comparison to [2], we have no second-order quantifiers.

The formulas are given by the following grammar: for expository purposes, we only consider conjunction & and implication $\rightarrow$.

$$\alpha, \beta \ ::= \ \mathsf{true} \ \mid \ \alpha \,\&\, \beta \ \mid \ \alpha \rightarrow \beta \ \mid \ A \ \mathit{says}\ \alpha \ \mid \ A \Rightarrow B$$

A *says* α connects the calculus of principals to the logic: this is the quoting combinator of the logic and is related to the quoting combinator of the lattice by defining $A\,|\,B$ *says* α to be A *says* B *says* α.

We describe Hilbert-style axioms to describe the tautologies. We first define B-protected formulas [2, 39]. Informally, if there is a proof of a B-protected formula, then there is one that does not require statements of principals that are more trustworthy than B.

Definition 10. The class of B-protected formulas is defined inductively as follows: (a) true is B-protected. (b) A *says* α is B-protected if either α is B-protected or the ordering $B \Rightarrow A$ holds. (c) $\alpha \,\&\, \beta$ (resp. $\alpha \rightarrow \beta$) is B-protected if α and β (resp. β) are B-protected. (d) The ordering formula $A \Rightarrow C$ is B-protected if $B \Rightarrow C$. □

In concordance with the informal intuitions, the following axiom system satisfies the property that if a formula is B-protected and $A \Rightarrow B$, then the formula is also A-protected.

Definition 11. The axioms of authorization logic ($\vdash \alpha$) are as follows. (a) *Propositional validity:* If α is an instance of a intuitionist propositional tautology, then $\vdash \alpha$. (b) *Modus Ponens:* If $\vdash \alpha$ and $\vdash \alpha \rightarrow \beta$, then $\vdash \beta$. (c) *Modality-Unit:* If $\vdash \alpha$, then $\vdash A$ *says* α (d) *Modality-Mult:* If $\vdash \alpha \,\&\, \alpha' \rightarrow \beta$. (e) *Lattice:* If $A \Rightarrow B$ in the security lattice, then $\vdash A \Rightarrow B$. □

Following [2], examples of provable theorems are (a) Order Naturality: if $\vdash A$ *says* α and $A \Rightarrow B$, then B *says* α; (b) Reflexivity: A *says* A *says* $\alpha \leftrightarrow A$ *says* α; (c) Commutativity: A *says* B *says* $\alpha \leftrightarrow B$ *says* A *says* α; and (d) Extensivity: A *says* $\alpha \rightarrow B$ *says* A *says* α.

Remark 12. The primary use of principals in the logic is via the quoting formulas constructed with *says*. So, it is conceptually consistent to assume that properties (b)–(d) are reflected back into the security lattice, i.e., $|$ is reflexive, commutative, and extensive. $\square$

Remark 13. In contrast to [2], we identify the lattice order $\Rightarrow$ and the speaks-for relation. The two important consequences of "speaks-for" are derived as follows. (a) Order-Naturality: if $A \Rightarrow B$, then A *says* $\alpha \rightarrow B$ *says* α. (b) Since $B \Rightarrow A$ is A-protected, we can deduce $B \Rightarrow A$ from A *says* $B \Rightarrow A$. $\square$

The above remarks are motivated by finiteness considerations (Remark 1), although they do reduce the expressiveness of the calculus of principals.

4.2. Extended Datalog

We describe the syntax and semantics of a variant of Datalog extended to work over the authorization logic. As with regular Datalog, a program will be built from a set of Horn clauses without function symbols. In contrast to regular Datalog, the literals are in the form of quotes of principals. Despite this extra generality, the extended formalism has decidable clause inference. We establish this by a translation of extended Datalog into Datalog.

Syntax of Extended Datalog

$\mathbb{X}$	Variables
p	Predicates (Including $\Rightarrow$)
$u, v, w \ ::= \ \mathbb{X} \mid M$	Terms
$\mathbb{L}, \mathbb{K} \ ::= \ u \textit{ says } p(\vec{v})$	Literals
$\mathbb{C}, \mathbb{D} \ ::= \ \mathbb{L} \text{ :- } \mathbb{K}_1, \ldots, \mathbb{K}_n$	Clauses ($fv(\mathbb{L}) \subseteq \cup_i fv(\mathbb{K}_i)$)

Extended Datalog terms include variables and terms from the underlying process calculus.

Clauses in extended Datalog are a subset of the language presented in Section 4.1. We write the predicate $\Rightarrow$ infix as in a *says* $B \Rightarrow C$ (or a *says* $\vec{s}$) and define A-protected clauses as follows.

Definition 14. A clause $\mathbb{L} \text{ :- } \vec{\mathbb{K}}$ is *u-protected* if $\mathbb{L}$ is u-protected according to Definition 10. $\square$

For example, the clause $(u \textit{ says } v \Rightarrow w) \text{ :- } \vec{\mathbb{K}}$ is A-protected if $A \Rightarrow u$ or $A \Rightarrow w$.

Definition 15. We define nested uses of *says* as a meta-operation using compound principals: $u \textit{ says } (v \textit{ says } p(\vec{w})) \triangleq (u \mid v) \textit{ says } p(\vec{w})$. $\square$

The following example is a variant of one presented in [20]. It illustrates the kind of distributed policy that can be represented in this language.

Example 16. Consider a company A. It is agreed globally that A' is a subsidiary of A. It is also globally agreed that the employee of a subsidiary is also an employee of the parent company. $\mathsf{A_{HR}}$ is the HR service of A. $\mathsf{A_{HR}}$ believes that B is an employee of A'. $\mathsf{R_{con}}$ believes that all employees of A can access some resource R. As far as C is concerned, $\mathsf{R_{con}}$ is the authority on access to R. We will deduce that C permits B to access R. This policy is encoded in the following extended Datalog program.

$$
\begin{array}{lll}
\mathbf{0} \textit{ says } \mathsf{A}' \Rightarrow \mathsf{A} & \text{:-} & \\
\mathbf{0} \textit{ says } \mathsf{emp}(\mathsf{X}, \mathsf{par}) & \text{:-} & \mathbf{0} \textit{ says } \mathsf{emp}(\mathsf{X}, \mathsf{sub}), \\
 & & \mathbf{0} \textit{ says } \mathsf{sub} \Rightarrow \mathsf{par} \\
\mathsf{A_{HR}} \textit{ says } \mathsf{emp}(\mathsf{B}, \mathsf{A}') & \text{:-} & \\
\mathsf{R_{con}} \textit{ says } \mathsf{X} \Rightarrow \mathsf{R} & \text{:-} & \mathsf{A_{HR}} \textit{ says } \mathsf{emp}(\mathsf{X}, \mathsf{A}) \\
\mathsf{C} \textit{ says } \mathsf{X} \Rightarrow \mathsf{R} & \text{:-} & \mathsf{R_{con}} \textit{ says } \mathsf{X} \Rightarrow \mathsf{R}
\end{array}
$$

Globally agreed clauses are represented as quotes of $\mathbf{0}$. The last three clauses may be represented by local policies at $\mathsf{A_{HR}}$, $\mathsf{R_{con}}$ and C, respectively. $\square$

Semantics of Inference. The predicate "$\Rightarrow$" is special because of its use in the definition of *protected* literals. We require that the following bootstrap clause be included in all extended Datalog programs: $\mathbf{0} \textit{ says } \mathbb{X} \Rightarrow \mathbb{Y} \text{ :- } \mathbf{0} \textit{ says } \mathbb{X} \Rightarrow \mathbb{Y}$.

Let θ range over substitutions of variables $\vec{\mathbb{X}}$ for terms $\vec{u}$. The inference rules for ground literals ($\vec{\mathbb{C}} \vDash \mathbb{L}$) are as follows. The rule can be lifted to clauses ($\vec{\mathbb{C}} \vDash \mathbb{D}$) in the standard way.

Inference for Ground Literals $(\vec{\mathbb{C}} \vDash \mathbb{L})$

$$\frac{\mathbb{L} \text{ :- } \vec{\mathbb{K}} \in \vec{\mathbb{C}} \quad (\forall i)\ \vec{\mathbb{C}} \vDash \mathbb{K}_i\theta}{\vec{\mathbb{C}} \vDash \mathbb{L}\theta}$$

$$\frac{\vec{s} \Vdash M \Rightarrow N \quad \big(\forall (L \Rightarrow L') \in \vec{s}\big)\ \vec{\mathbb{C}} \vDash u \textit{ says } L \Rightarrow L'}{\vec{\mathbb{C}} \vDash u \textit{ says } M \Rightarrow N}$$

$$\frac{\vec{\mathbb{C}} \vDash \mathbb{K}\theta}{\vec{\mathbb{C}} \vDash u \textit{ says } \mathbb{K}\theta}$$

$$\frac{\mathbb{L}\theta \text{ is } u\text{-protected} \quad \mathbb{L} \text{ :- } \vec{\mathbb{K}} \in \vec{\mathbb{C}} \quad (\forall i)\ \vec{\mathbb{C}} \vDash u \textit{ says } \mathbb{K}_i\theta}{\vec{\mathbb{C}} \vDash \mathbb{L}\theta}$$

We comment on the relation between extended Datalog and the axioms for authorization logic of Definition 11. The first rule is the usual inference rule for Datalog, reflecting *modus ponens*. The *lattice* axiom is reflected in the next rule. The third rule reflects *modality-unit*, whereas the fourth reflects *modality-mult*.

Lemma 17. *Let σ range over substitutions of variables x for terms M. Extended Datalog inference satisfies the following:*

- *If $\vec{\mathbb{C}} \vDash \mathbb{D}$ then $(\forall \vec{\mathbb{E}})\ \vec{\mathbb{C}}, \vec{\mathbb{E}} \vDash \mathbb{D}$.*
- *If $\vec{\mathbb{C}} \vDash \mathbb{D}$ and $\vec{\mathbb{C}}, \mathbb{D} \vDash \mathbb{E}$ then $\vec{\mathbb{C}} \vDash \mathbb{E}$.*
- *If $\vec{\mathbb{C}} \vDash \mathbb{D}$ then $(\forall \sigma)\ \vec{\mathbb{C}}\sigma \vDash \mathbb{D}\sigma$.* $\square$

Translating into Regular Datalog. Consider an extended Datalog program $\vec{\mathbb{C}}$. In the full version of this paper, we adapt [20] to describe a translation of $\vec{\mathbb{C}}$ into regular Datalog that is sound and complete for the inference of ground literals.

We sketch the key step in the construction, namely that $\mathscr{L}$ — the lattice of all the possible principals that can occur during execution of $\vec{\mathbb{C}}$ — is finite. The generators of $\mathscr{L}$ includes all atomic principals occurring in $\vec{\mathbb{C}}$. It also includes a fresh "symbolic" atomic principal for each quoting-free program term that occurs in $\vec{\mathbb{C}}$, where program terms are considered up to ground term equivalence. $\mathscr{L}$ is constructed using Remark 1 as the free interpretation of the quoting operation on the free $\wedge$-semilattice on this set of generators. $|\mathscr{L}|$ is doubly exponential in the number of generators for $\vec{\mathbb{C}}$.

The translation yields a regular Datalog program $\vec{\mathbb{C}}'$ of size $|\vec{\mathbb{C}}'| \leq O(|\mathscr{L}|^2 + |\vec{\mathbb{C}}|)$. Thus we have a decision procedure for clause inference: Suppose that θ maps the free extended Datalog variables of $\mathbb{L}_1, \ldots, \mathbb{L}_n$ to fresh, distinct names. Further, suppose that $\vec{\mathbb{C}}, \mathbb{L}_1\theta, \ldots, \mathbb{L}_n\theta \vDash \mathbb{L}\theta$. Then $\vec{\mathbb{C}} \vDash \mathbb{L} \mathrel{:\!-} \mathbb{L}_1, \ldots, \mathbb{L}_n$.

4.3. Types

We first sketch the goals of typing, which are formalized later. Recall that the syntax of processes (and therefore configurations) includes extended Datalog clauses ($\mathbb{C}$) and *expectations* (expect $\mathbb{C}$). The interpretation of a clause $a[\mathbb{C}]$ is modulated by the atomic principal a that utters it (using the meta-operation a *says* $\mathbb{C}$).

An *opponent* is an untrustworthy atomic principal. Opponents may utter any clause and may have unreasonable expectations. We model opponents as principals equivalent to $\mathbf{1}$, the least trustworthy principal. We then require that $\mathbf{1}$ *says* α is valid for any α, and thus clauses of opponents are effectively ignored. In typing, we assume that all sets of extended Datalog clauses are closed with respect to this requirement, though we will often elide the necessary clauses in the interests of succinctness.

A configuration G has a *runtime error* if it contains an expectation that cannot be justified by the accumulated clauses of G (in addition to those statically defined). A configuration is *safe* if it has no runtime errors (and this property is preserved by evaluation). Our typing system ensures *robust safety*, that is, safety of typed configurations when combined with arbitrary opponents.

The typing system does not attempt to prevent representation errors, e.g., using a tuple as a channel. Thus the only nontrivial types are those for labels, which carry effects.

Types and Environments

$T, U ::= \mathsf{Un} \mid \mathsf{Label}(\vec{x}:\vec{T})\vec{\mathbb{C}}$ — Types

$E ::= \cdot \mid E, Z \mid E, \eta : T \mid E, \mathbb{C}$ — Environments

$E(\eta) \triangleq T$ if $E = E', \eta : T, E''$.

The type $\mathsf{Label}(\vec{x}:\vec{T})\vec{\mathbb{C}}$ represent a latent effect, labeled with a name. In this type, the variables $\vec{x}$ are bound in $\vec{\mathbb{C}}$. The match construct is required to unlock the label and expose the effects. For simplicity, we treat the enclosed elements $\vec{x}$ at type Un; it is straightforward to generalize the typing system to allow these to be label types as well.

All other terms are assigned type Un. The type system is designed to permit all opponents to circumvent checks by using Un.

Unlike [22], the type associated with an identifier $(\eta : T)$ carries no meaningful information with respect to logical inference.

Definition 18. Logical inference is lifted to environments $(E \vDash \mathbb{C})$ simply by ignoring non-clauses in E. Let $dom(E)$ be the domain E, including identifiers and process variables. □

In the composition $G \mid H$, assumptions in G may be discharged in H. The typing rules use *env* to collect the clauses in G, producing a suitable environment. To simplify the definition, we assume (without loss of generality) that all names bound by new are distinct and fresh.

Env

$env(0) = \cdot$
$env(G \mid H) = env(G), env(H)$
$env(\mathsf{new}\, n : T . G) = n : T, env(G)$
$env(\mathsf{new}\, b . G) = b : \mathsf{Un}, env(G)$
$env(a[P]) = env_a(P)$
$env(a\langle\vec{s}\rangle) = a \text{ says } \vec{s}$

Typing. We describe the rules for environments, terms, processes and configurations. An environment is well-typed if it binds all free names, variables and atomic principals in label types and clauses.

Environment $(E \vdash \diamond)$

$$\frac{}{\cdot \vdash \diamond} \qquad \frac{E \vdash \diamond \quad Z \notin dom(E)}{E, Z \vdash \diamond} \qquad \frac{E \vdash \diamond \quad fn(\mathbb{C}) \subseteq dom(E)}{E, \mathbb{C} \vdash \diamond}$$

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

The typing of terms is relative to the principal at which the term occurs, and similarly for processes.

Term $(E \vDash_A M : T)$

$$\frac{E \vdash \diamond \quad E(\eta) = T}{E \vDash_A \eta : T} \qquad \frac{E \vdash \diamond \quad E(\eta) = T}{E \vDash_A \eta : \mathsf{Un}}$$

$$\frac{}{E \vDash_A \mathsf{del} : \mathsf{Un}} \qquad \frac{}{E \vDash_A \mathbf{0} : \mathsf{Un}} \qquad \frac{}{E \vDash_A \mathbf{1} : \mathsf{Un}}$$

$$\frac{E \vdash_{\overline{A}} B : \mathsf{Un} \quad E \vdash_{\overline{A}} C : \mathsf{Un}}{E \vdash_{\overline{A}} B \,|\, C : \mathsf{Un}} \qquad \frac{E \vdash_{\overline{A}} B : \mathsf{Un} \quad E \vdash_{\overline{A}} C : \mathsf{Un}}{E \vdash_{\overline{A}} B \wedge C : \mathsf{Un}}$$

$$\frac{E \vdash_{\overline{A}} N : \mathsf{Label}(\vec{x}{:}\vec{T})\vec{\mathbb{C}} \quad (\forall i)\, E \vdash_{\overline{A}} M_i : T_i \quad E \models A \textit{ says } \vec{\mathbb{C}}\{\vec{x} := \mathsf{tag}\, A(\vec{M})\}}{E \vdash_{\overline{A}} N(\vec{M}) : \mathsf{Un}}$$

$$\frac{E \vdash_{\overline{A}} N : \mathsf{Un} \quad (\forall i)\, E \vdash_{\overline{A}} M_i : \mathsf{Un} \quad E \models \mathbf{0} \textit{ says } \mathbf{1} \Rightarrow A}{E \vdash_{\overline{A}} N(\vec{M}) : \mathsf{Un}}$$

$$\frac{E \vdash_{\overline{A}} B : \mathsf{Un} \quad E \vdash_{\overline{B}} M : T}{E \vdash_{\overline{A}} \mathsf{sig}\, B(M) : T} \qquad \frac{E \vdash_{\overline{A}} B : \mathsf{Un} \quad E \vdash_{\overline{A|B}} M : T}{E \vdash_{\overline{A}} \mathsf{tag}\, B(M) : T}$$

$$\frac{E \vdash_{\overline{A}} M : T \quad E \models A \textit{ says } M.\mathsf{src} \Rightarrow A}{E \vdash_{\overline{A}} M.\mathsf{val} : T} \qquad \frac{E \vdash_{\overline{A}} M : T}{E \vdash_{\overline{A}} M.\mathsf{src} : \mathsf{Un}}$$

Variables, names and principals may be viewed at type Un in addition to any type contained in the environment. The first rule for labeled tuples allows honest processes to create tuples as long as the effect of the label is respected; the second rule allows opponents to create tuples with arbitrary labels. The rules for sig and tag cause the effective location of the enclosed term to change. The soundness of these rules follows from the extensivity of *for* and $|$. The typing rule for val allows a principal to discard the source of a term if that source is at least as trusted as itself; if this is not the case, val may still be used inside an appropriate tag ().

Processes $(E \vdash_{\overline{a}} P)$

$$\frac{E, env(Q) \vdash_{\overline{a}} P \quad E, env(P) \vdash_{\overline{a}} Q \quad fn(P \,|\, Q) \subseteq dom(E)}{E \vdash_{\overline{a}} P \,|\, Q}$$

$$\frac{E \vdash \diamond}{E \vdash_{\overline{a}} 0} \qquad \frac{E, Z \vdash_{\overline{a}} P}{E \vdash_{\overline{a}} \mu Z.P} \qquad \frac{E \vdash \diamond \quad Z \in dom(E)}{E \vdash_{\overline{a}} Z}$$

$$\frac{E, n{:}T \vdash_{\overline{a}} P}{E \vdash_{\overline{a}} \mathsf{new}\, n{:}T.P} \qquad \frac{E, b{:}\mathsf{Un}, b \textit{ says } a \Rightarrow b \vdash_{\overline{b}} P}{E \vdash_{\overline{a}} \mathsf{new}\, b \,\mathsf{with}\, P}$$

$$\frac{E \vdash_{\overline{a}} M : \mathsf{Un} \quad E \vdash_{\overline{a}} N : \mathsf{Un}}{E \vdash_{\overline{a}} M!N} \qquad \frac{E \vdash_{\overline{a}} M : \mathsf{Un} \quad E, x{:}\mathsf{Un} \vdash_{\overline{a}} P}{E \vdash_{\overline{a}} M?x.P}$$

$$\frac{E \vdash_{\overline{a}} M : \mathsf{Un} \quad E \vdash_{\overline{a}} N : \mathsf{Label}(\vec{x}{:}\vec{T})\vec{\mathbb{C}} \quad E, \vec{x}{:}\vec{T}, M.\mathsf{src} \textit{ says } \vec{\mathbb{C}} \vdash_{\overline{a}} P}{E \vdash_{\overline{a}} \mathsf{match}\, M \,\mathsf{as}\, N(\vec{x}).P}$$

$$\frac{E \vdash_{\overline{a}} M : \mathsf{Un} \quad E \vdash_{\overline{a}} N : \mathsf{Un} \quad E, \vec{x}{:}\mathsf{Un} \vdash_{\overline{a}} P}{E \vdash_{\overline{a}} \mathsf{match}\, M \,\mathsf{as}\, N(\vec{x}).P}$$

$$\frac{E \vdash_{\overline{a}} M : \mathsf{Un} \quad E \vdash_{\overline{a}} N : \mathsf{Un} \quad E \vdash_{\overline{a}} P \quad E \models a \textit{ says } M \Rightarrow N}{E \vdash_{\overline{a}} \mathsf{learn}\, M \Rightarrow N.P}$$

$$\frac{E \vdash_{\overline{a}} M : \mathsf{Un} \quad E \vdash_{\overline{a}} N : \mathsf{Un} \quad E, a \textit{ says } M \Rightarrow N \vdash_{\overline{a}} P \quad E \vdash_{\overline{a}} Q}{E \vdash_{\overline{a}} \mathsf{check}\, M \Rightarrow N \,\mathsf{then}\, P \,\mathsf{else}\, Q}$$

$$\frac{E, \mathbb{C} \vdash \diamond}{E \vdash_{\overline{a}} \mathbb{C}} \qquad \frac{E, \mathbb{C} \vdash \diamond \quad E \models \mathbb{C}}{E \vdash_{\overline{a}} \mathsf{expect}\, \mathbb{C}} \qquad \frac{E, \mathbb{C} \vdash \diamond \quad E \models \mathbf{0} \textit{ says } \mathbf{1} \Rightarrow a}{E \vdash_{\overline{a}} \mathsf{expect}\, \mathbb{C}}$$

The rule for parallel composition should be viewed as a conjoining of specifications: each component can assume the exposed clauses of the other component. The rules for 0, recursive processes, new names, input and output are standard. Note that in the rule for new principals, the residual is typed at the new principal. In addition, new principals may make use of the fact that they are ordered below their parent.

As there are two rules for creating tuples, there are also two rules for matching them. The first rule allows honest processes to use the latent effect of the label when typing in the residual. Thus, the match construct can be viewed as a "dynamic cast" operation acting on an untyped message. The second rule allows matching in opponents.

The rule for learn demands static validation of modifications to the local security lattice. Since check is a conditional, the typing rule expands the environment in case that the check is satisfied.

The rule for clauses ensures syntactic validity. The first rule for expectations ensures derivability from the clauses in the environment. The second allows arbitrary expectations in opponents.

Configurations $(E \vdash G)$

$$\frac{E, env(H) \vdash G \quad E, env(G) \vdash H \quad fn(G \,|\, H) \subseteq dom(E)}{E \vdash G \,|\, H}$$

$$\frac{E \vdash_{\overline{a}} P \quad a \in dom(E)}{E \vdash a[P]} \qquad \frac{E \vdash \diamond}{E \vdash 0} \qquad \frac{E, n{:}T \vdash G}{E \vdash \mathsf{new}\, n{:}T.G} \qquad \frac{E, b{:}\mathsf{Un} \vdash G}{E \vdash \mathsf{new}\, b.G}$$

$$\frac{a \in dom(E) \quad (\forall i)\, E \vdash_{\overline{a}} M_i : \mathsf{Un} \quad (\forall i)\, E \vdash_{\overline{a}} N_i : \mathsf{Un} \quad (\forall i)\, E \Vdash a \textit{ says } M_i \Rightarrow N_i}{E \vdash a\langle M_1 \Rightarrow N_1, \ldots, M_n \Rightarrow N_n\rangle}$$

Each process in a configuration is typed at its locating principal. The rules for composition and restriction follow those for processes. The final rule ensures that each local order is consistent with the environment.

Example 19. Consider the following program.

$$\begin{array}{l} \mathbf{new}\, \mathsf{l}{:}\mathsf{Label}(\mathsf{x}{:}\mathsf{Un},\mathsf{y}{:}\mathsf{Un})\{\mathsf{B} \textit{ says } \mathsf{x} \Rightarrow \mathsf{y}\}. \\ \qquad\qquad \mathsf{a}[\mathsf{B} \textit{ says } \mathsf{c} \Rightarrow \mathsf{d} \text{ :- }] \mid \mathsf{a}[\_!\mathsf{l}(\mathsf{c},\mathsf{d})] \end{array}$$

If a is not an opponent, then typing term l(c,d) requires that $(\mathsf{a} \,|\, \mathsf{B})$ *says* $\mathsf{c} \Rightarrow \mathsf{d}$. Typechecking of the right process succeeds using the assumptions of the left, via *env*. □

A typed program validates all learn and expect statements, even in the presence of opponents. The result relies on initiality (Definition 5), but not well-formedness.

Definition 20 (Runtime Error). A configuration G is *erroneous at* a (notation $G \not\downarrow a$) if either (a) $G \rightarrow^* G' \,|\, a[\mathsf{learn}\ M \Rightarrow N.P]$ and $E, env(G') \not\models a \textit{ says } M \Rightarrow N$, or (b) $G \rightarrow^* G' \,|\, a[\mathsf{expect}\ \mathbb{C}]$ and $E, env(G') \not\models \mathbb{C}$. □

Definition 21 (Opponent). An atomic principal a is an *E-opponent principal* if $E \models \mathbf{0} \textit{ says } \mathbf{1} \Rightarrow a$ or $a \notin dom(E)$.

An *E-opponent configuration* is a configuration $a_1[P_1] \,|\, \ldots \,|\, a_n[P_n] \,|\, b_1\langle\vec{s}_1\rangle \,|\, \ldots \,|\, b_m\langle\vec{s}_m\rangle$ such that every a_i and b_j is an E-opponent principal. □

Definition 22 (Robust Safety). A configuration G is *robustly E-safe* if for every initial E-opponent configuration H, we have that $(G \mid H) \lightning a$ implies that a is an E-opponent principal. $\square$

Theorem 23 (Robust Safety). *If $E \vdash G$, then G is robustly E-safe.*

As usual, the proof of robust safety depends on lemmas for opponent typability (if H is an initial E-opponent then $E, E' \vdash H$, for some suitable E') and type preservation (if $E \vdash G$ and $G \to H$ then $E \vdash H$).

4.4. Typing the SSO Example

We describe how correspondences such as that discussed in Remark 9 can be statically checked. We start with the following static policy.

$$\begin{array}{l}
\mathsf{sp}\ \mathit{says}\ \mathbb{X} \Rightarrow \mathbb{Y} \texttt{ :- } \mathsf{sp}\ \mathit{says}\ \mathbb{Z} \Rightarrow \mathsf{authorized\text{-}ip}, \mathbb{Z}\ \mathit{says}\ \mathbb{X} \Rightarrow \mathbb{Y} \\
\mathsf{sp}\ \mathit{says}\ (\mathbf{1}\,\mathit{for}\,(\mathsf{srv} \mid \mathsf{ip})) \Rightarrow \mathsf{authorized\text{-}ip} \texttt{ :-} \\
\mathsf{sp}\ \mathit{says}\ \mathsf{inst} \Rightarrow \mathsf{res} \texttt{ :-} \\
\mathsf{srv}\ \mathit{says}\ \mathsf{uid} \Rightarrow \mathsf{inst} \texttt{ :-}
\end{array}$$

Conformance ensures that the local orders obey this static policy. This is true of the initial orders:

$$\begin{array}{l}
\mathsf{sp}\langle\mathsf{inst} \Rightarrow \mathsf{res}, \mathbf{1}\,\mathit{for}\,(\mathsf{srv} \mid \mathsf{ip}) \Rightarrow \mathsf{authorized\text{-}ip}\rangle \\
\mathsf{srv}\langle\mathsf{uid} \Rightarrow \mathsf{inst}\rangle
\end{array}$$

With respect to typing, the most significant fragment of sp is the following code servicing sp-auth.

$$\begin{array}{l}
\textbf{check}\ \mathsf{cert.src} \Rightarrow \mathsf{authorized\text{-}ip}\ \textbf{then} \\
\quad \textbf{match}\ \mathsf{cert}\ \textbf{as}\ \mathsf{okcert}(\mathsf{z_{uid}}, \mathsf{z_{inst}}).\ \textbf{learn}\ \mathsf{z_{uid}} \Rightarrow \mathsf{z_{inst}}
\end{array}$$

In order to typecheck the learn, we must deduce

$$\mathsf{sp}\ \mathit{says}\ \mathsf{z_{uid}} \Rightarrow \mathsf{z_{inst}}. \qquad (*)$$

This is achieved by assigning okcert an appropriate type.

$$\mathsf{okcert} : \mathsf{Label}(\mathsf{z_{uid}}, \mathsf{z_{inst}})\{\mathbf{0}\ \mathit{says}\ \mathsf{z_{uid}} \Rightarrow \mathsf{z_{inst}}\}$$

The type of okcert states that the sender of the tuple believes that the first enclosed principal dominates the second. The learn is then typed using assumptions:

$$\begin{array}{ll}
\mathsf{sp}\ \mathit{says}\ \mathsf{cert.src} \Rightarrow \mathsf{authorized\text{-}ip} & \ldots\ \textit{from check} \\
\mathsf{cert.src}\ \mathit{says}\ \mathsf{z_{uid}} \Rightarrow \mathsf{z_{inst}} & \ldots\ \textit{from match}
\end{array}$$

Combined with the first clause of the static policy, this is sufficient to deduce $(*)$. One may also annotate the learn with an explicit expectation, such as the following.

$$\textbf{expect}\ \mathbb{X}\ \mathit{says}\ \mathsf{z_{uid}} \Rightarrow \mathsf{z_{inst}} \texttt{ :- } \mathsf{sp}\ \mathit{says}\ \mathbb{X} \Rightarrow \mathsf{authorized\text{-}ip}$$

This can be typed under the same assumptions.

Having discussed the certificate's receiver, sp, we now turn attention to its creator, srv. The relevant code fragment is the following.

$$\begin{array}{l}
\textbf{check}\ \mathsf{x.src} \Rightarrow \mathsf{inst} \\
\quad \textbf{then}\ \mathsf{x!}(\textbf{tag}\ \mathsf{ip}(\mathsf{okcert}(\mathsf{x.src}, \mathsf{inst})))]
\end{array}$$

In order to typecheck the output, we must deduce

$$(\mathsf{srv} \mid \mathsf{ip})\ \mathit{says}\ \mathsf{x.src} \Rightarrow \mathsf{inst}. \qquad (\dagger)$$

The output is typed under the following assumption.

$$\mathsf{srv}\ \mathit{says}\ \mathsf{x.src} \Rightarrow \mathsf{inst} \quad \ldots\ \textit{from check}$$

Combining this with the static policy, $(\dagger)$ follows from order naturality and the extensivity of $\mid$.

The static policy we started with is quite permissive, in that sp allows an authorized-ip to say anything at all. More realistically, we may restrict authorized-ip to speak only for inst by replacing the first line of the static policy with the following.

$$\begin{array}{l}
\mathsf{sp}\ \mathit{says}\ \mathbb{X} \Rightarrow \mathbb{Y} \texttt{ :- } \mathsf{sp}\ \mathit{says}\ \mathbb{Z} \Rightarrow \mathsf{authorized\text{-}ip}, \\
\qquad\qquad \mathbb{Z}\ \mathit{says}\ \mathbb{X} \Rightarrow \mathbb{Y}, \mathsf{sp}\ \mathit{says}\ \mathsf{inst} \Rightarrow \mathbb{Y}
\end{array}$$

With this policy, however, the code servicing sp-auth does not typecheck. To correct the problem, we must add an additional check.

$$\begin{array}{l}
\textbf{check}\ \mathsf{cert.src} \Rightarrow \mathsf{authorized\text{-}ip}\ \textbf{then} \\
\quad \textbf{match}\ \mathsf{cert}\ \textbf{as}\ \mathsf{okcert}(\mathsf{z_{uid}}, \mathsf{z_{inst}}). \\
\qquad \textbf{check}\ \mathsf{inst} \Rightarrow \mathsf{z_{inst}}\ \textbf{then}\ \textbf{learn}\ \mathsf{z_{uid}} \Rightarrow \mathsf{z_{inst}}
\end{array}$$

Then the learn is typed successfully under the additional assumption "sp *says* inst $\Rightarrow \mathsf{z_{inst}}$".

With a few modifications, one can establish the expectation "sp *says* uid $\Rightarrow$ inst" after the input on $\mathsf{yes_2}$ in uid; the user can determine that sp has the necessary information to grant access on a subsequent sp-req.

This does not, however, imply that sp has correspondingly updated its local order. Despite the validity of the expectation, subsequent requests may be denied. Type checking guarantees that all permitted accesses are justified; it does not, however, address the dual question of whether every permitted access is actually granted. For that, we turn to model checking.

5. Model Checking

We apply reachability analysis to Daisy programs to analyze the converse of the problem studied via typechecking: does a configuration incorporate order information entailed by the policy into local trust lattices? We outline our approach to model checking and the fragment of the language that can be analyzed, then sketch the translation used to reduce the reachability problem to an existing decidability result for reachability in a fragment of pi-calculus.

Analysis via Reachability. We can use reachability to encode interesting questions about SSO systems, building on Section 3.

Example 24 (Single Sign On Revisited). If a user of an SSO system has previously signed in, they expect to be granted access to a resource for which they are entitled. One may specify this by stating that certain states of a system are unreachable, e.g., the uid code from Section 3 might be modified to:

ip-req**!**(**new** c).c?cert. ... *credentials received from* srv
  sp-auth**!**(cert, **new** yes_1, **new** no_1).
    yes_1? ... *logged in to* sp
      sp-req**!**(**new** yes_2, **new** no_2).
        no_2? END ... *access denied by* sp

This user proactively logs in to srv, then communicates the login credentials to sp before requesting access to sp's resources. If analysis reveals that END is unreachable, then the user can be assured that their behavior will be rewarded with access to the desired resource. □

The analysis tactic in Example 24 can be generalized to the converse of the problem addressed by typechecking, i.e., does a configuration incorporate order information entailed by the policy into local trust lattices? For example, for the SSO example of Section 3, does sp's local trust lattice entail uid ⇒ res whenever statements made by the configuration entail sp *says* uid ⇒ res? This approach allows us to move from the hand-crafted analysis in Example 24 to an analysis based upon the policy adopted for type checking.

There may be a delay between deducibility of a statement (with respect to the combined global policy) and updates to a local trust order. In the absence of an additional temporal specification we adopt a late interpretation, verifying the *completeness* of the local trust order with respect to global deducibility at the point that a process performs a check.

Definition 25 (Complete). A configuration G is *complete* if whenever $G \rightarrow^* G' \,|\, a[\mathsf{check}\ M \Rightarrow N\ \mathsf{then}\ P\ \mathsf{else}\ Q]$ and $env(G') \vDash a\ says\ M \Rightarrow N$, then there exists G'' such that $G' \equiv G'' \,|\, a\langle\vec{s}\rangle$ and $\vec{s} \Vdash M \Rightarrow N$. □

Below, we distinguish a bounded fragment of our language, dubbed $\mathsf{Daisy_B}$, for which completeness is decidable. All variants of the SSO example can be modified to conform to this fragment (in part by replacing parallel composition with internal choice).

Assuming that the static policy of Section 4.4 is included in the initial configuration, then sp *says* uid ⇒ res is immediately deducible from the global policy. We can verify that the SSO code of Example 24 is complete; e.g., when the check uid ⇒ res is executed in sp, it will succeed. In contrast, the original code of Section 3 is not complete; the initial service request from uid to sp (before login) will cause the check uid ⇒ res to fail.

A Bounded Fragment and its Translation. The reachability problem is decidable for an expressive bounded fragment $\mathsf{Daisy_B}$ of Daisy via translation to the expressive fragment of asynchronous polyadic pi-calculus for which reachability is shown to be decidable in [7]. The language of [7] includes parameterized process definitions with name generation but places two restrictions on process definitions: the *bounded input* condition (each process definition has exactly one continuation or ends with the internal choice between two continuations) and the *unique receiver* condition (there is at most one process that can receive input for each channel name). The source of the translation, $\mathsf{Daisy_B}$, is therefore also defined in terms of parameterized processes with the bounded input and unique receiver conditions. $\mathsf{Daisy_B}$ also presumes a finite lattice of principals and a routine polyadic typing system (as opposed to the type-and-effect system of Section 4).

A parameterized process definition is either the internal choice of two parameterized processes, or has the form:

$$\begin{aligned} Z(\vec{x}_1 : \mathsf{Ch}(\vec{T}_1); \vec{x}_2 : \vec{T}_2) = {} & \\ & y_1 ? y_2 : U_2 . \mathsf{new}\, \vec{n}_3 : \mathsf{Ch}(\vec{T}_3). \\ & \mathsf{match}\ y_3\ \mathsf{as}\ N(\vec{x}_4 : \vec{T}_4). \\ & \mathbb{C} \,|\, (\mathsf{learn}\, M_1 \Rightarrow M_2. \\ & \quad \mathsf{check}\ M_3 \Rightarrow M_4 \\ & \quad\ \mathsf{then}\ (\vec{\eta}_1 ! \vec{N}_1 \,|\, Z_1(\vec{\eta}_2; \vec{N}_2)) \\ & \quad\ \mathsf{else}\ (\vec{\eta}_3 ! \vec{N}_3 \,|\, Z_2(\vec{\eta}_4; \vec{N}_4))) \end{aligned}$$

Following [7], the parameters $\vec{x}_1$ are bound only to channel names, and y_1 must be chosen from this list. In addition, the names $\vec{\eta}_2$, $\vec{\eta}_4$ must be chosen from $\vec{x}_1$ or $\vec{n}_3$. Unlike [7], the parameters $\vec{x}_2$ may be bound not only to names, but more generally to terms. This allows terms to be carried into continuations without imposing additional communication, which would alter the source of the term.

The operational semantics of $\mathsf{Daisy_B}$ is obtained from that of Section 2 by modifying the structural rule for unfolding to operate on such declared names by performing an appropriate substitution into the body of the declaration.

$\mathsf{Daisy_B}$ parameterized processes must typecheck using a routine polyadic type system. The type system keeps label names distinct from channel names. The form of process definitions precludes dynamic generation of new principals or names for labeled tuples, thus allowing name matching to be encoded in the target language.

As is the case for the target language, $\mathsf{Daisy_B}$ is expressive. Trivial uses of clauses, match, learn, and check are easily written (e.g., learning an inequality that is already known). Sequential uses of multiple clauses, matches, learns, and checks can be encoded using multiple process definitions with continuations.

We require that $\Rightarrow$ be the only predicate to occur in a clause of a $\mathsf{Daisy_B}$ program. In conjunction with the finite lattice of principals, the collection of instantiations of clauses in this form is finite. This permits tracking the clauses stated by a configuration and checking that each principal's local trust lattice is consistent with these clauses.

The key ingredients of the translation from $\mathsf{Daisy_B}$ to asynchronous polyadic pi-calculus are:

Name Matching: The elements of the finite lattice of principal names, and the names for labeled tuples, are translated to channel names. Internal choice is used to encode name matching, which suffices for reachability questions.

Runtime Principal Computations: Computation of $a \wedge b$ and $a \mid b$ takes place at runtime, which is possible because of principal name matching and the fact that the lattice is finite, so every case can be encoded into a server process that handles requests for these computations.

Labeled Tuples as Lists: A labeled tuple in a $\mathsf{Daisy_B}$ process is transmitted as a list of names in the (polyadic pi-calculus) translation. The size corresponds to the number of leaves of the labeled tuple, i.e., the number of principal and channel names nested inside the labeled tuple.

Terms with Principal: The translation of a $\mathsf{Daisy_B}$ term is also transmitted with a principal name computed at runtime from the combination of one or more sig/tag constructors. For example, the term $\mathsf{sig}\,A\,(\mathsf{sig}\,B\,(m))$ would be translated into two names: the first carrying the principal $A\,\mathit{for}\,B$ and the second representing m.

Located Trust Lattices: A single process is created to store the current state of each principal's trust lattice and, if checking completeness (Definition 25), the collection of statements made by the configuration. Checks, learns, and statements in $\mathsf{Daisy_B}$ are translated into a request-response dialogue with the process storing this information. Since the initial trust lattice is finite, both the set of inequations that can be learned by each principal, and the set of clauses that can be issued as statements, are finite. Therefore the process has finite state. The response of the process to execution of the translation of a $\mathsf{Daisy_B}$ check is defined in terms of order entailment and clause inference. If clause inference establishes the inequation in a check, but the principal in question's current trust lattice does not entail the inequation, then a well-known end state is reached.

Results. The translation extends from parameterized process definitions to configurations, and, critically, both preserves and reflects reachability of $\mathsf{Daisy_B}$ parameterized processes. In conjunction with [7], we obtain a decision procedure for reachability questions.

Proposition 26. *Reachability is preserved and reflected by the translation. Moreover, it is decidable whether a process definition is reachable from a* $\mathsf{Daisy_B}$ *configuration.* □

Theorem 27. *It is decidable whether a* $\mathsf{Daisy_B}$ *configuration is complete.* □

6. Conclusions

Daisy falls into the broad area of language-based approaches to security, specifically access control. Our results bridge the gap between specifications (based on authorization logics) with implementations (based on programming with explicit identities).

We advance the technical state-of-the-art with three results: (a) robust safety for an asynchronous pi-calculus enriched with compound identities, (b) translation of a distributed authorization logic into Datalog, and (c) decidability for a suitable notion of completeness on a bounded fragment of our language.

Acknowledgements. Andrew Cirillo and James Riely were supported by NSF Career 0347542. Radha Jagadeesan and Corin Pitcher were supported by NSF Cybertrust 0430175.

References

[1] R. Aarts and P. Madsen. Liberty ID-WSF interaction service specification. http://www.projectliberty.org/, 2006.

[2] M. Abadi. Access control in a core calculus of dependency. In *ICFP*, pages 263–273. ACM, 2006.

[3] M. Abadi, M. Burrows, B. W. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, 1993.

[4] M. Abadi and C. Fournet. Access control based on execution history. In *Proc. Network and Distributed System Security Symp.*, 2003.

[5] M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *POPL*, pages 302–315, 2000.

[6] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, 1995.

[7] R. M. Amadio and C. Meyssonnier. On decidability of the control reachability problem in the asynchronous π-calculus. *Nord. J. Comput.*, 9(1):70–101, 2002.

[8] R. M. Amadio and S. Prasad. Localities and failures (extended abstract). In *FSTTCS*, volume 880 of *LNCS*, pages 205–216. Springer, 1994.

[9] M. Y. Becker, A. D. Gordon, and C. Fournet. SecPAL: Design and semantics of a decentralized authorization language. Technical Report MSR-TR-2006-120, Microsoft Research, 2006.

[10] E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.

[11] L. Bettini, V. Bono, R. D. Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim project: Theory and practice. In *Global Computing*, volume 2874 of *LNCS*, pages 88–150. Springer, 2003.

[12] B. Blanchet. Automatic verification of cryptographic protocols: a logic programming approach. In *PPDP*, pages 1–3. ACM, 2003.

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

[14] P. Bonatti, S. D. C. di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.*, 5(1):1–35, 2002.

[15] C. Braghin, D. Gorla, and V. Sassone. A distributed calculus for role-based access control. In *CSFW*, pages 48–60, 2004.

[16] M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: The calculus of boxed ambients. *ACM Trans. Program. Lang. Syst.*, 26(1):57–124, 2004.

[17] I. Castellani. Process algebras with localities. In *Handbook of Process Algebra*, chapter 15, pages 945–1045. North-Holland, 2001.

[18] A. Chander, D. Dean, and J. C. Mitchell. Reconstructing trust management. *Journal of Computer Security*, 12(1):131–164, 2004.

[19] A. Compagnoni, P. Garralda, and E. Gunter. Role-based access control in a mobile environment. In *Symposium on Trustworthy Global Computing*, 2005.

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

[21] G. L. Ferrari, U. Montanari, and E. Tuosto. Model checking for nominal calculi. In *FoSSaCS*, volume 3441 of *LNCS*, pages 1–24. Springer, 2005.

[22] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *ESOP*, volume 3444 of *LNCS*, pages 141–156. Springer, 2005.

[23] D. Garg, L. Bauer, K. D. Bowers, F. Pfenning, and M. K. Reiter. A linear logic of authorization and knowledge. In *ESORICS*, pages 297–312, 2006.

[24] D. Garg and F. Pfenning. Non-interference in constructive authorization logic. *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, 0:283–296, 2006.

[25] D. P. Guelev, M. Ryan, and P.-Y. Schobbens. Model-checking access control policies. In *ISC*, volume 3225 of *LNCS*, pages 219–230. Springer, 2004.

[26] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in Security-Enhanced Linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[27] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *CSFW '03: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 118–130, 2003.

[28] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.

[29] S. Hinrichs and P. Naldurg. Attack-based domain transition analysis. In *Proceedings of the 2006 Security-Enhanced Linux Symposium*, 2006. http://selinux-symposium.org/2006/agenda.php.

[30] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992.

[31] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, 1974.

[32] S. Landau. Liberty ID-WSF security and privacy overview. http://www.projectliberty.org/, 2006.

[33] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1):128–171, 2003.

[34] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, page 114, 2002.

[35] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[36] C. Pitcher and J. Riely. Dynamic policy discovery with remote attestation. In *FoSSaCS*, volume 3921 of *LNCS*, pages 111–125. Springer, 2006.

[37] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. *Journal of Automated Reasoning*, 31(3-4):335–370, 2003.

[38] M. B. Smyth. Power domains. *J. Comput. Syst. Sci.*, 16(1):23–36, 1978.

[39] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *ICFP*, pages 115–125. ACM, 2004.

[40] D. Wijesekera and S. Jajodia. Policy algebras for access control - the predicate case. In *Computer and Communications Security*, pages 171–180. ACM Press, 2002.

[41] D. Wijesekera and S. Jajodia. A propositional policy algebra for access control. *ACM Trans. Inf. Syst. Secur.*, 6(2):286–325, 2003.

[42] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Trans. Comput. Syst.*, 12(1):3–32, 1994.

[43] T. Y. Woo and S. S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Research in Security and Privacy*, 1993.

[44] N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In *ISC*, volume 3650 of *LNCS*, pages 446–460. Springer, 2005.

[45] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *SACMAT '04: Proceedings of the Ninth ACM Symposium on Access Control Models And Technologies*, pages 1–10. ACM Press, 2004.

A. Translating into Regular Datalog

Closure for Lattice Properties Consider an extended Datalog program $\vec{\mathbb{C}}$. Our first step is to construct $\mathscr{L}$, the lattice of all possible principals that can occur during execution of the extended datalog program.

We first describe the generators arising from a literal u *says* $p(\vec{v})$. These are given by (a) the atomic principals that occur in $u, \vec{v}$ and (b) terms in $\vec{v}$ and quoting-free subterms of u that are not datalog variables (upto ground term equivalence) viewed as symbolic atomic principals.

The generators for $\vec{\mathbb{C}}$ is given by the union of these sets for all literals in all the clauses of $\vec{\mathbb{C}}$. $\mathscr{L}$ is constructed using Remark 1 as the free interpretation of the quoting operation on the free $\wedge$-semilattice on this set of generators. $|\mathscr{L}|$ is doubly exponential in the number of generators for $\vec{\mathbb{C}}$.

Let κ and λ range over the elements of $\mathscr{L}$. We write $\kappa \mathsf{M} \lambda$ (resp. $\kappa \mathsf{Q} \lambda$) to refer to the meet (resp. quote) of κ and λ in $\mathscr{L}$.

Having constructed $\mathscr{L}$, we compute $close(\vec{\mathbb{C}})$, the *closure* of the $\vec{\mathbb{C}}$ with clauses that validates the axioms of Section 2.1, which includes the following clauses.

(a) $\Rightarrow$ *clauses:* If $\kappa \Rightarrow \lambda$ in $\mathscr{L}$, add $\mathbf{0}$ *says* $\kappa \Rightarrow \lambda$:-. Add clauses to encode reflexivity (i.e. $\mathbf{0}$ *says* $\mathbb{X} \Rightarrow \mathbb{X}$:-), and transitivity.

(b) $\wedge$-*clauses:* Let the 3-ary predicate meet represent the relational interpretation of meet ($\mathbf{0}$ *says* $\mathsf{meet}(\kappa \mathsf{M} \lambda, \kappa, \lambda)$:-). Then encode the properties of $\wedge$ from section 2.1. For example, the encoding of $\wedge$-monotonicity is as follows.

$$\mathbf{0}\ \mathit{says}\ \mathbb{X} \Rightarrow \mathbb{Y} \text{ :- } \mathbf{0}\ \mathit{says}\ \mathbb{X}' \Rightarrow \mathbb{Y}' \,, \mathbf{0}\ \mathit{says}\ \mathbb{X}'' \Rightarrow \mathbb{Y}'' \\ \mathbf{0}\ \mathit{says}\ \mathsf{meet}(\mathbb{X}, \mathbb{X}', \mathbb{X}'') \,, \mathbf{0}\ \mathit{says}\ \mathsf{meet}(\mathbb{Y}, \mathbb{Y}', \mathbb{Y}'')$$

(c) $|$-*clauses:* Similarly let the 3-ary predicate quot represent the relational interpretation of quoting ($\mathbf{0}$ *says* $\mathsf{quot}(\kappa \mathsf{Q} \lambda, \kappa, \lambda)$:-), with the necessary clauses to capture the properties form Section 2.1. For example, the encoding of $|$-monotonicity is as follows.

$$\mathbf{0}\ \mathit{says}\ \mathbb{X} \Rightarrow \mathbb{Y} \text{ :- } \mathbf{0}\ \mathit{says}\ \mathbb{X}' \Rightarrow \mathbb{Y}' \,, \mathbf{0}\ \mathit{says}\ \mathbb{X}'' \Rightarrow \mathbb{Y}'' \\ \mathbf{0}\ \mathit{says}\ \mathsf{quot}(\mathbb{X}, \mathbb{X}', \mathbb{X}'') \,, \mathbf{0}\ \mathit{says}\ \mathsf{quot}(\mathbb{Y}, \mathbb{Y}', \mathbb{Y}'')$$

Consider an extended Datalog program that has been closed as described above. We will translate a n-ary predicate as a $n+1$-ary predicate, the extra position being used to record the quoter of the predicate. For a ground fact $\mathbb{L} = u$ *says* $p(\vec{u})$ in extended Datalog, the corresponding ground fact $\mathbb{L}'$ in regular Datalog is $p(u, \vec{u})$.

Encoding Modality-Unit. For each n-ary predicate in source extended Datalog program, say $p(\cdot)$, add:
$p(\mathbb{X}, \mathbb{X}_1, \ldots, \mathbb{X}_n) \text{ :- } p(\mathbb{X}', \mathbb{X}_1, \ldots, \mathbb{X}_n), \mathsf{quot}(\mathbb{X}, \mathbb{X}', \mathbb{X}'')$

Encoding the program. For each clause u *says* $p(\vec{v})$:- $\vec{\mathbb{K}}$, add a clause $p(u, \vec{u})$:- $\vec{\mathbb{K}}'$, where $\vec{\mathbb{K}}'$ is defined from $\vec{\mathbb{K}}$ as follows: for each v *says* $p'(\vec{v})$ in $\vec{\mathbb{K}}$, there are two literals $p'(\mathbb{X}, \vec{v}), \mathsf{quot}(\mathbb{X}, u, v)$ in $\vec{\mathbb{K}}'$.

In addition, for each clause of the form u *says* $v \Rightarrow v'$:- $\vec{\mathbb{K}}$, add a clause $p(u, \vec{u})$:- $\vec{\mathbb{K}}'$, where $\vec{\mathbb{K}}'$ is the translation of $\vec{\mathbb{K}}$ defined as follows: for each v'' *says* $p'(\vec{v})$ in $\vec{\mathbb{K}}$, there are two literals $p'(\mathbb{X}, \vec{v}), \mathsf{quot}(\mathbb{X}, v'', v')$ in $\vec{\mathbb{K}}'$.

There are at most two clauses for each clause in the source program. So, given an extended Datalog program $\vec{\mathbb{C}}$, the size of the translated program is at most $O(|\mathscr{L}|^2 + |\vec{\mathbb{C}}|)$.

B. Proofs

Let σ range over substitutions of variables x for terms M. We first prove that if $\vec{\mathbb{C}} \models \mathbb{D}$ then $(\forall \sigma)\ \vec{\mathbb{C}}\sigma \models \mathbb{D}\sigma$.

The proof relies on some other lemmas.

Lemma 28 (Substitutivity of Order Entailment). *If* $\vec{s} \Vdash M \Rightarrow N$ *then* $(\forall \sigma)\ \vec{s}\sigma \Vdash M\sigma \Rightarrow N\sigma$

PROOF. Induction on the number of rules required to prove $\vec{s} \Vdash M \Rightarrow N$. ☐

Lemma 29 (Substitutivity of Protected). *If* $\mathbb{L}$ *is* u-*protected, then* $(\forall \sigma)$, $\mathbb{L}\sigma$ *is* $u\sigma$-*protected.*

PROOF. There are two cases depending on the two forms of $\mathbb{L}$: (a) v *says* w and $u \Rightarrow v$, or (b) v *says* $w \Rightarrow w'$ and $u \Rightarrow w'$. In either case, result follows from Substitutivity of order formulas and order entailment ☐

Corollary 30. *If* $\mathbb{L}$:- $\vec{\mathbb{K}}$ *is* u-*protected, then* $(\forall \sigma)$, $(\mathbb{L} \text{ :- } \vec{\mathbb{K}})\sigma$ *is* $u\sigma$-*protected.* ☐

Proposition 31 (Substitutivity of Inference for Clauses). *If* $\vec{\mathbb{C}} \models \mathbb{D}$ *then* $(\forall \sigma)\ \vec{\mathbb{C}}\sigma \models \mathbb{D}\sigma$.

PROOF. Induction on the number of rules required to prove $\vec{\mathbb{C}} \models \mathbb{D}$. ☐

We now turn to proofs related to robust safety. First, we observe that under certain circumstances, environments can be reordered.

Lemma 32 (Permutation). *If* $E_1, E_2, E_3 \vdash \diamond$ *and* $fn(E_2) \subseteq dom(E_1)$ *and* $fn(E_3) \subseteq dom(E_1)$*, then*

(a) $E_1, E_3, E_2 \vdash \diamond$,

(b) $E_1, E_2, E_3 \models_a M : T$ *implies* $E_1, E_3, E_2 \models_a M : T$,

(c) $E_1, E_2, E_3 \models_a P$ *implies* $E_1, E_3, E_2 \models_a P$*, and*

(d) $E_1, E_2, E_3 \vdash G$ *implies* $E_1, E_3, E_2 \vdash G$.

PROOF. Straightforward induction on the derivation of each judgement. ☐

Lemma 33 (Weakening). *Let* E, E' *be environments such that,* $E \vdash \diamond$, $dom(E) \cap dom(E') = \emptyset$*, and* $fn(E') \subseteq dom(E)$*:*

(a) $E, E' \vdash \diamond$.

(b) *If* $E \models_A M : T$*, and* $E \models B$ *says* $A \Rightarrow B$*, then* $E, E' \models_B M : T$.

(c) *If* $E \models_a P$ *then* $E, E' \models_a P$.

(d) *If* $E \vdash G$ *then* $E, E' \vdash G$.

PROOF. We prove each claim individually.

(a) Follows directly from the definition of well-formed environment.

(b) Straightforward induction on the structure of M, appealing to the monotonicity of inference in extended Datalog and the order naturality of $\Rightarrow$ (see Remark 13).

(c) By induction on the structure of P, appealing to (a) and (b) when necessary. Three cases are interesting:

$(P \mid Q)$ Assume $E \models_a P \mid Q$ and $dom(E) \cap dom(E') = \emptyset$.
By the typing rule, $E, env(Q) \models_a P$,
and $E, env(P) \models_a Q$.
By the induction hypothesis, $E, env(Q), E' \models_a P$,
and $E, env(P), E' \models_a Q$.
By permutation, $E, E', env(Q) \models_a P$,
and $E, E', env(P) \models_a Q$.
By the typing rule, $E, E' \models_a P \mid Q$.

$(\mathsf{learn}\, M \Rightarrow N.P)$ Assume $E \models_a \mathsf{learn}\, M \Rightarrow N.P$,
and $dom(E) \cap dom(E') = \emptyset$.
By the typing rule, $E \models_a M : \mathsf{Un}$ and $E \models_a N : \mathsf{Un}$, and
$clauses(E) \vDash a\ says\ M \Rightarrow N$,
and $E \models_a P$.
By (b), $E, E' \models_a M : \mathsf{Un}$ and $E, E' \models_a N : \mathsf{Un}$.
By monotonicity of inference for clauses,
$clauses(E, E') \vDash a\ says\ M \Rightarrow N$.
By the induction hypothesis, $E, E' \models_a P$.
By the typing rule, $E, E' \models_a \mathsf{learn}\, M \Rightarrow N.P$.

$(\mathsf{expect}\ \mathbb{C})$ Assume $E \models_a \mathsf{expect}\ \mathbb{C}$,
and $dom(E) \cap dom(E') = \emptyset$.
By the typing rule, $E, \mathbb{C} \vdash \diamond$ and $clauses(E) \Vdash \mathbb{C}$.
By (a), $E, \mathbb{C}, E' \vdash \diamond$.
By permutation, $E, E', \mathbb{C} \vdash \diamond$.
By Monotonicity, $clauses(E, E') \Vdash \mathbb{C}$.
By the typing rule, $E, E' \models_a \mathsf{expect}\ \mathbb{C}$.

(d) Straightforward induction on the structure of G, appealing to (a) and (c) when necessary. □

Proposition 34 (Substitutivity of Typing). *Let $E, x{:}T, E'$ be an environment such that $E, x{:}T, E' \vdash \diamond$, and $\{x := M\}$ an arbitrary substitution. Then,*

(a) If $E, x{:}T, E' \vDash \vec{\mathbb{C}}$ and $E \models_A M : T$ then $E, E'\{x := M\} \vDash \vec{\mathbb{C}}\{x := M\}$.

(b) If $E \models_A M : T$ then $E, E'\{x := M\} \vdash \diamond$.

(c) If $E, x{:}T, E' \models_A N : U$ and $E \models_A M : T$ then $E, E'\{x := M\} \models_A N\{x := M\} : U\{x := M\}$.

(d) If $E, x{:}T, E' \models_a P$ and $E \models_a M : T$ then $E, E'\{x := M\} \models_a P\{x := M\}$.

PROOF. We prove each claim individually.

(a) First note that by the definition of clauses, $E, x{:}M, E'\{x := M\} = E, E'\{x := M\}$. Then the result follows from the Substitutivity of Inference for Clauses.

(b) Proof by induction on the derivation of $E, x{:}T, E' \vdash \diamond$.

$(\cdot \vdash \diamond)$ Trivial.

$(E, x{:}T, E'', y{:}U \vdash \diamond)$ By hypothesis, $E, x{:}T, E'' \vdash \diamond$,
and $fn(U) \subseteq dom(E, x{:}T, E'')$,
and $y \notin dom(E, x{:}T, E'')$.
By the induction hypothesis, $E, E''\{x := M\} \vdash \diamond$.
It is easy to show that substitution commutes with $fn(\cdot)$ and $dom(\cdot)$,
therefore $fn(U\{x := M\}) \subseteq dom(E, E''\{x := M\})$,
and $y \notin dom(E, E''\{x := M\})$.
By the rule, $E, E''\{x := M\}, y{:}U\{x := M\} \vdash \diamond$.

$(E, x{:}T, E'', \mathbb{C})$ By hypothesis, $E, x{:}T, E'' \vdash \diamond$,
and $fn(\mathbb{C}) \subseteq dom(E, x{:}T, E'')$.
By the induction hypothesis, $E, E''\{x := M\} \vdash \diamond$.
By def., $fn(\mathbb{C}\{x := M\}) \subseteq dom(E, E''\{x := M\})$.
By the rule, $E, E'\{x := M\}, \mathbb{C}\{x := M\} \vdash \diamond$.

(c) Proof by induction on the derivation of $E, x{:}T, E' \models_A N : U$:

$(E, x{:}T, E' \models_A \eta : U$ **where** $E, x{:}T, E'(\eta) = U)$ There are two subcases:
If $(\eta = x)$:
By inspection of the rules, either $U = T$ or $U = \mathsf{Un}$.
If $U = \mathsf{Un}$, see the following case, for now assume $U = T$.
By (b), $E, E'\{x := M\} \vdash \diamond$.
By definition of wfe, $x \notin fn(T)$,
so $U = T = T\{x := M\} = U\{x := M\}$.
By definition, $\eta\{x := M\} = M$.
By Weakening, $E, E'\{x := M\} \models_A M : T$,
so, $E, E'\{x := M\} \models_A \eta\{x := M\} : U\{x := M\}$.
If $(\eta \neq x)$:
By (b), $E, E'\{x := \eta\} \vdash \diamond$.
By def. of subst., $\eta\{x := M\} = \eta$.
By def. of subst., $E, E'\{x := M\}(n) = U\{x := M\}$.
By the type rule, $E, E'\{x := M\} \models_A \eta : U\{x := M\}$,
so, $E, E'\{x := M\} \models_A \eta\{x := M\} : U\{x := M\}$.

$(E, x{:}T, E' \models_A \eta : \mathsf{Un}$ **where** $E, x{:}T, E'(\eta) = U)$ By (b), $E, E'\{x := M\} \vdash \diamond$.
By Weakening, $E, E'\{x := M\} \models_A M : T$.
From inspection of the typing rules, we can see that $E, E'\{x := M\} \models_A M : \mathsf{Un}$.
By def. of subst., $\eta\{x := M\} = M$ or η,
in either case, $E, E'\{x := M\} \models_A \eta\{x := M\} : \mathsf{Un}$.

$(E, x{:}T, E' \models_A \mathsf{del} : \mathsf{Un})$
Trivial.

$(E, x{:}T, E' \models_A \mathbf{0} : \mathsf{Un})$
Trivial.

$(E, x{:}T, E' \models_A \mathbf{1} : \mathsf{Un})$
Trivial.

$(E, x{:}T, E' \models_A B \wedge C : \mathsf{Un})$
Straightforward induction.

$(E, x{:}T, E' \models_A B \mid C : \mathsf{Un})$
Straightforward induction.

($E,x\!:\!T,E' \vdash_{\overline{A}} N(\vec{N}) : \mathsf{Un}$)
where $E,x\!:\!T,E' \models A$ *says* $\mathbb{C}\{\vec{y} := \mathsf{tag}\,A(\vec{N})\}$
By hypothesis,
$E,x\!:\!T,E' \vdash_{\overline{A}} N : \mathsf{Label}(\vec{y}\!:\!\vec{U})\mathbb{C}$,
and $(\forall i)$ $E,x\!:\!T,E' \vdash_{\overline{A}} N_i : U_i$,
and $E,x\!:\!T,E' \models A$ *says* $\mathbb{C}\{\vec{y} := \mathsf{tag}\,A(\vec{N})\}$.
By the induction hypothesis,
$E,E' \vdash_{\overline{A}} N\{x := M\} : \mathsf{Label}(\vec{y}\!:\!\vec{U})\mathbb{C}\{x := M\}$,
and $(\forall i)$ $E,E' \vdash_{\overline{A}} N_i\{x := M\} : U_i\{x := M\}$.
By (a), $E,E'\{x := M\} \models$
A *says* $(\vec{\mathbb{C}}\{x := M\})\{\vec{y} := \mathsf{tag}\,A(\vec{N})\{x := M\}\}$.
By the rule, $E,E'\{x := M\} \vdash_{\overline{A}} N(\vec{N})\{x := M\} : \mathsf{Un}$.

($E,x\!:\!T,E' \vdash_{\overline{A}} N(\vec{N}) : \mathsf{Un}$)
where $E,x\!:\!T,E' \models \mathbf{0}$ *says* $\mathbf{1} \Rightarrow A$
By hypotheisis,
$E,x\!:\!T,E' \vdash_{\overline{A}} N : \mathsf{Un}$,
and $(\forall i)$ $E,x\!:\!T,E' \vdash_{\overline{A}} N_i : U_i$,
and $E,x\!:\!T,E' \models \mathbf{0}$ *says* $\mathbf{1} \Rightarrow A$.
By the induction hypothesis,
$E,E'\{x := M\} \vdash_{\overline{A}} N : \mathsf{Un}$,
and $(\forall i)$ $E,E'\{x := M\} \vdash_{\overline{A}} N_i\{x := M\} : U_i\{x := M\}$.
By (a), $E,E'\{x := M\} \models \mathbf{0}$ *says* $\mathbf{1} \Rightarrow A$.
By the rule, $E,E'\{x := M\} \vdash_{\overline{A}} N(\vec{N})\{x := M\} : \mathsf{Un}$.

($E,x\!:\!T,E' \vdash_{\overline{A}} \mathsf{sig}\,B(N) : U$)
By hypothesis, $E,x\!:\!T,E' \vdash_{\overline{B}} N : U$.
By the induction hypothesis,
$E,E'\{x := M\} \vdash_{\overline{B}} N\{x := M\} : U\{x := M\}$.
By the typing rule,
$E,E'\{x := M\} \vdash_{\overline{A}} \mathsf{sig}\,B(N\{x := M\}) : U\{x := M\}$.
By def. of subst.,
$\mathsf{sig}\,B(N)\{x := M\} = \mathsf{sig}\,B(N\{x := M\})$, so,
$E,E'\{x := M\} \vdash_{\overline{A}} \mathsf{sig}\,B(N)\{x := M\} : U\{x := M\}$.

($E,x\!:\!T,E' \vdash_{\overline{A}} \mathsf{tag}\,B(N) : U$)
By hypothesis, $E,x\!:\!T,E' \vdash_{\overline{A|B}} N : U$.
By the induction hypothesis,
$E,E'\{x := M\} \vdash_{\overline{A|B}} N\{x := M\} : U\{x := M\}$.
By the typing rule,
$E,E'\{x := M\} \vdash_{\overline{A}} \mathsf{tag}\,B(N\{x := M\}) : U\{x := M\}$.
By def. of subst.,
$\mathsf{tag}\,B(N)\{x := M\} = \mathsf{tag}\,B(N\{x := M\})$, so,
$E,E'\{x := M\} \vdash_{\overline{A}} \mathsf{tag}\,B(N)\{x := M\} : U\{x := M\}$.

($E,x\!:\!M,E' \vdash_{\overline{A}} N.\mathsf{val} : U$)
By hypothesis, $E,x\!:\!T,E' \vdash_{\overline{A}} N : U$,
and $E,x\!:\!T,E' \models A$ *says* $N.\mathsf{src} \Rightarrow A$.
By the induction hypothesis,
$E,E'\{x := M\} \vdash_{\overline{A}} N\{x := M\} : U\{x := M\}$.
By (a),
$E,E'\{x := M\} \models (A$ *says* $N.\mathsf{src} \Rightarrow A)\{x := M\}$.
By def. of subst., this reduces to
$E,E'\{x := M\} \models A$ *says* $N\{x := M\}.\mathsf{src} \Rightarrow A$.
By the typing rule,
$E,E'\{x := M\} \vdash_{\overline{A}} N\{x := M\}.\mathsf{val} : U\{x := M\}$.
By def. of subst.,
$E,E'\{x := M\} \vdash_{\overline{A}} N.\mathsf{val}\{x := M\} : U\{x := M\}$.

($E,x\!:\!T,E' \vdash_{\overline{A}} N.\mathsf{src} : \mathsf{Un}$)
Similar to previous case.

(d) Straightforward induction on the derivation of $E,x\!:\!T,E' \vdash_{\overline{a}} P$. All cases are easy, appealing to (a), (b), (c) and Monotonicity of Inference for Clauses. □

Lemma 35. *If $E \models \mathbb{C}$ and $E,\mathbb{C} \vdash_{\overline{a}} P$ then $E \vdash_{\overline{a}} P$.*

PROOF. By induction on the derivation of $E,\mathbb{C} \vdash_{\overline{a}} P$, appealing to transitivity of inference. □

Lemma 36. *If $E \vdash G$ and $G \equiv H$ then $E \vdash H$.*

PROOF. Straightforward induction on the derivation of $G \equiv H$. □

Lemma 37 (Properties of src). *We note that src has the following properties:*

(a) $M.\mathsf{src}(A) \Rightarrow M.\mathsf{src}(B)$ iff $A \Rightarrow B$.

(b) $A \Rightarrow M.\mathsf{src}(A)$.

PROOF. Both claims follow directly from the definition of src, noting that $\Vdash A \Rightarrow (A\,\mathit{for}\,B)$ and $\Vdash A \Rightarrow (A\,|\,B)$ are tautologies in the axiomatization of entailment. □

Lemma 38. *If $E \vdash_{\overline{A}} M : T$ then $E \vdash_{\overline{M.\mathsf{src}(A)}} M.\mathsf{val} : T$.*

PROOF. By case analysis of the structure of M. All but the following two cases are immediate.

Case ($\mathsf{tag}\,B(N)$) Assume $E \vdash_{\overline{A}} \mathsf{tag}\,B(N) : T$.
By definition, $\mathsf{tag}\,B(N).\mathsf{val} = N.\mathsf{val}$,
and $\mathsf{tag}\,B(N).\mathsf{src}(A) = A\,|\,(N.\mathsf{src}(B))$.
By the typing rule, $E \vdash_{\overline{A|B}} N : T$.
By Lemma 37 and Weakening, $E \vdash_{\overline{A|(N.\mathsf{src}(B))}} N : T$.
By Lemma 37, $N.\mathsf{src} \Rightarrow N.\mathsf{src}(B)$.
By def. of $|$, $N.\mathsf{src} \Rightarrow A\,|\,(N.\mathsf{src}(B))$.
Finally, by the typing rule, $E \vdash_{\overline{A|(N.\mathsf{src}(B))}} N.\mathsf{val} : T$.

Case ($\mathsf{sig}\,B(N)$) Assume $E \vdash_{\overline{A}} \mathsf{sig}\,B(N) : T$.
By definition, $\mathsf{sig}\,B(N).\mathsf{val} = N.\mathsf{val}$,
and $\mathsf{sig}\,B(N).\mathsf{src}(A) = A\,\mathit{for}\,(N.\mathsf{src}(B))$.
By the typing rule, $E \vdash_{\overline{B}} N : T$.
By Lemma 37 and Weakening, $E \vdash_{\overline{A\,\mathit{for}(N.\mathsf{src}(B))}} N : T$.
By def. of *for*, $N.\mathsf{src}(B) \Rightarrow A\,\mathit{for}\,(N.\mathsf{src}(B))$, so
$A\,\mathit{for}\,(N.\mathsf{src}(B))$ *says* $N.\mathsf{src}(B) \Rightarrow A\,\mathit{for}\,(N.\mathsf{src}(B))$.
Finally, by the typing rule, $E \vdash_{\overline{A\,\mathit{for}(N.\mathsf{src}(B))}} N.\mathsf{val} : T$. □

Corollary 39. *If $E \vdash_{\overline{A}} M : T$ then $E \vdash_{\overline{M.\mathsf{src}}} M.\mathsf{val} : T$.*

PROOF. Follows from Lemmas 37, 38 and Weakening, noting that $M.\mathsf{src}$ is defined as $M.\mathsf{src}(\mathbf{0})$. □

Proposition 40 (Type Preservation). *If $G \rightarrow H$ and $E \vdash G$ then $E \vdash H$.*

PROOF. By induction on the derivation of $G \rightarrow H$.

Case $(a\,[\mathsf{new}\,b\ \mathsf{with}\,P] \rightarrow \mathsf{new}\,b.(b\,[P] \mid b\langle a \Rightarrow b\rangle))$

Assume $E \vdash a\,[\mathsf{new}\,b\ \mathsf{with}\,P]$.
By the typing rule, $E(a) = \mathsf{Un}$ and $E \vdash_a \mathsf{new}\,b\ \mathsf{with}\,P$.
By the typing rule, $E, b{:}\mathsf{Un}, b\ \mathit{says}\ a \Rightarrow b \vdash_{\bar b} P$.
By the typing rule, $E, b{:}\mathsf{Un}, b\ \mathit{says}\ a \Rightarrow b \vdash b\,[P]$.
By the typing rule, $E, b{:}\mathsf{Un} \vdash b\langle a \Rightarrow b\rangle$.
By Weakening, $E, b{:}\mathsf{Un}, \mathit{env}(b\,[P]) \vdash b\langle a \Rightarrow b\rangle$.
Noting that $\mathit{env}(b\langle a \Rightarrow b\rangle) = b\ \mathit{says}\ a \Rightarrow b$,
by the typing rule $E, b{:}\mathsf{Un} \vdash b\,[P] \mid b\langle a \Rightarrow b\rangle$.
By the typing rule, $E \vdash \mathsf{new}\,b.(b\,[P] \mid b\langle a \Rightarrow b\rangle)$.

Case $(a\,[M\,!N] \mid b\,[M'?x.P] \rightarrow b\,[P\{x := \mathsf{sig}\,a(N)\}])$

Assume $E \vdash a\,[M\,!N] \mid b\,[M'?x.P]$.
By hypothesis, $M.\mathsf{val} \simeq M'.\mathsf{val} \simeq n$, for some n.
By the typing rule, $E, \mathit{env}(b\,[M'?x.P]) \vdash a\,[M\,!N]$
and $E, \mathit{env}(a\,[M\,!N]) \vdash b\,[M'?x.P]$
By definition, $\mathit{env}(b\,[M'?x.P]) = \cdot$
and $\mathit{env}(a\,[M\,!N]) = \cdot$,
so, $E \vdash a\,[M\,!N]$
and $E \vdash b\,[M'?x.P]$.
By the typing rule for output, $E \vdash_a M\,!N$.
By the typing rule for output, $E \vdash_a M : \mathsf{Un}$
and $E \vdash_a N : \mathsf{Un}$.
By the typing rule for input, $E \vdash_{\bar b} M' : \mathsf{Un}$
and $E, x{:}\mathsf{Un} \vdash_{\bar b} P$.
By the typing rule for sig, $E \vdash_{\bar b} \mathsf{sig}\,a(N) : \mathsf{Un}$.
By Substitution, $E \vdash_{\bar b} P\{x := \mathsf{sig}\,a(N)\}$.
Finally, by the typing rule for cfg, $E \vdash b\,[P\{x := \mathsf{sig}\,a(N)\}]$.

Case $(a\,[\mathsf{match}\,M\ \mathsf{as}\,L(\vec{x}).P] \rightarrow a\,[P\{\vec{x} := \mathsf{tag}\,B(\vec{N})\}])$

Assume $E \vdash a\,[\mathsf{match}\,M\ \mathsf{as}\,L(\vec{x}).P]$.
By hypothesis, $M.\mathsf{val} \simeq L'(\vec{N})$, $M.\mathsf{src} \simeq B$,
$L.\mathsf{val} \simeq L'.\mathsf{val}$, and $|\vec{x}| = |\vec{N}|$.
By the typing rule, $E \vdash_a \mathsf{match}\,M\ \mathsf{as}\,L(\vec{x}).P$.
There are two subcases.
If $(E \vdash_a L : \mathsf{Label}(\vec{x}{:}\vec{T})\vec{\mathbb{C}})$**:**
By the typing rule, $E \vdash_a M : \mathsf{Un}$,
and $E, \vec{x}{:}\vec{T}, B\ \mathit{says}\ \vec{\mathbb{C}} \vdash_a P$.
By Corollary 39, $E \vdash_B L(\vec{N}) : \mathsf{Un}$.
By the typing rule, $(\forall i)\ E \vdash_B N_i : \mathsf{Un}$,
and $E \vDash B\ \mathit{says}\ \vec{\mathbb{C}}\{\vec{x} := \mathsf{tag}\,B(\vec{N})\}$.
By Weakening, noting that $\vDash B \Rightarrow (a \mid B)$,
$(\forall i)\ E \vdash_{a|B} N_i : \mathsf{Un}$.
By the typing rule, $(\forall i)\ E \vdash_a \mathsf{tag}\,B(N_i) : \mathsf{Un}$.
By Substitution,
$E, B\ \mathit{says}\ \vec{\mathbb{C}}\{\vec{x} := \mathsf{tag}\,B(\vec{N})\} \vdash_a P\{\vec{x} := \mathsf{tag}\,B(\vec{N})\}$.
By Lemma 35, $E \vdash_a P\{\vec{x} := \mathsf{tag}\,B(\vec{N})\}$.
Finally, by the typing rule, $E \vdash a\,[P\{\vec{x} := \mathsf{tag}\,B(\vec{N})\}]$.
If $(E \vdash_a L : \mathsf{Un})$**:**
By the typing rule, $E \vdash_a M : \mathsf{Un}$,
and $E, \vec{x}{:}\mathsf{Un} \vdash_a P$.
By Lemma 38, $E \vdash_B L(\vec{N}) : \mathsf{Un}$.
By the typing rule, $(\forall i)\ E \vdash_B N_i : \mathsf{Un}$.
By Weakening, noting that $\vDash B \Rightarrow (a \mid B)$,
$(\forall i)\ E \vdash_{a|B} N_i : \mathsf{Un}$.
By the typing rule, $(\forall i)\ E \vdash_a \mathsf{tag}\,B(N_i) : \mathsf{Un}$.
By Substitution, $E \vdash_a P\{\vec{x} := \mathsf{tag}\,B(\vec{N})\}$.
Finally, by the typing rule, $E \vdash a\,[P\{\vec{x} := \mathsf{tag}\,B(\vec{N})\}]$.

Case $(a\,[\mathsf{learn}\,M \Rightarrow N.P] \mid a\langle \vec{s}\rangle \rightarrow a\,[P] \mid a\langle \vec{s}, M \Rightarrow N\rangle)$

Assume $E \vdash a\,[\mathsf{learn}\,M \Rightarrow N.P] \mid a\langle \vec{s}\rangle$
where $\vec{s} = M_1 \Rightarrow N_1 \ldots M_n \Rightarrow N_n$.
By definition, $\mathit{env}(a\,[\mathsf{learn}\,M \Rightarrow N.P]) = \cdot$.
By definition, $\mathit{env}(a\langle \vec{s}\rangle) = a\ \mathit{says}\ \vec{s}$.
By the typing rule, $E, a\ \mathit{says}\ \vec{s} \vdash a\,[\mathsf{learn}\,M \Rightarrow N.P]$
and $E \vdash a\langle \vec{s}\rangle$.
By the typing rule, $E, a\ \mathit{says}\ \vec{s} \vdash_a \mathsf{learn}\,M \Rightarrow N.P$.
By the typing rule, $E, a\ \mathit{says}\ \vec{s} \vdash_a M : \mathsf{Un}$,
and $E, a\ \mathit{says}\ \vec{s} \vdash_a N : \mathsf{Un}$,
and $E, a\ \mathit{says}\ \vec{s} \vdash_a P$,
and $E, a\ \mathit{says}\ \vec{s} \vDash a\ \mathit{says}\ M \Rightarrow N$.
By the typing rule, $(\forall i)\ E \vdash_a M_i : \mathsf{Un}$ and $E \vdash_a N_i : \mathsf{Un}$.
By the typing rule $E \vdash a\langle \vec{s}, M \Rightarrow N\rangle$.
By Weakening, $E, \mathit{env}(a\,[P]) \vdash a\langle \vec{s}, M \Rightarrow N\rangle$.
By definition,
$\mathit{env}(a\langle \vec{s}, M \Rightarrow N\rangle) = a\ \mathit{says}\ \vec{s}, a\ \mathit{says}\ M \Rightarrow N$.
By Weakening, $E, a\ \mathit{says}\ \vec{s}, a\ \mathit{says}\ M \Rightarrow N \vdash_a P$.
By the typing rule, $E \vdash a\,[P] \mid a\langle \vec{s}, M \Rightarrow N\rangle$.

Case $(a\,[\mathsf{check}\,M \Rightarrow N\ \mathsf{then}\,P\ \mathsf{else}\,Q] \mid a\langle \vec{s}\rangle \rightarrow a\,[P] \mid a\langle \vec{s}\rangle)$

Assume $E \vdash a\,[\mathsf{check}\,M \Rightarrow N\ \mathsf{then}\,P\ \mathsf{else}\,Q] \mid a\langle \vec{s}\rangle$.
By the typing rule, $E, \mathit{env}(a\langle \vec{s}\rangle) \vdash a\,[\mathsf{check}\,M \Rightarrow N\ \mathsf{then}\,P\ \mathsf{else}\,Q]$
and $E, \mathit{env}(a\,[\mathsf{check}\,M \Rightarrow N\ \mathsf{then}\,P\ \mathsf{else}\,Q]) \vdash a\langle \vec{s}\rangle$.
By definition, $\mathit{env}(a\langle \vec{s}\rangle) = a\ \mathit{says}\ \vec{s}$,
so $E, a\ \mathit{says}\ \vec{s} \vdash a\,[\mathsf{check}\,M \Rightarrow N\ \mathsf{then}\,P\ \mathsf{else}\,Q]$
and $\mathit{env}(\mathsf{check}\,M \Rightarrow N\ \mathsf{then}\,P\ \mathsf{else}\,Q) = \cdot$,
so $E \vdash a\langle \vec{s}\rangle$.
By the typing rule, $E, a\ \mathit{says}\ \vec{s} \vdash_a \mathsf{check}\,M \Rightarrow N\ \mathsf{then}\,P\ \mathsf{else}\,Q$.
By the typing rule, $E, a\ \mathit{says}\ \vec{s} \vdash_a M : \mathsf{Un}$
and $E, a\ \mathit{says}\ \vec{s} \vdash_a N : \mathsf{Un}$
and $E, a\ \mathit{says}\ \vec{s}, a\ \mathit{says}\ M \Rightarrow N \vdash_a P$
and $E, a\ \mathit{says}\ \vec{s} \vdash_a Q$.
By hypothesis, $\vec{s} \Vdash M \Rightarrow N$.
By definition, $a\ \mathit{says}\ \vec{s} \Vdash a\ \mathit{says}\ M \Rightarrow N$.
By monotonicity, $\mathit{clauses}(E, a\ \mathit{says}\ \vec{s}) \Vdash M \Rightarrow N$.
By Lemma 35, $E, a\ \mathit{says}\ \vec{s} \vdash_a P$.
By the typing rule, $E, a\ \mathit{says}\ \vec{s} \vdash a\,[P]$.
By Weakening, $E, \mathit{env}(P) \vdash a\langle \vec{s}\rangle$.
Finally, By the typing rule, $E \vdash a\,[P] \mid a\langle \vec{s}\rangle$.

Case $(a\,[\mathsf{check}\,M \Rightarrow N\ \mathsf{then}\,P\ \mathsf{else}\,Q] \mid a\langle \vec{s}\rangle \rightarrow a\,[Q] \mid a\langle \vec{s}\rangle)$

Assume $E \vdash a\,[\mathsf{check}\,M \Rightarrow N\ \mathsf{then}\,P\ \mathsf{else}\,Q] \mid a\langle \vec{s}\rangle$.
By the typing rule, $E, \mathit{env}(a\langle \vec{s}\rangle) \vdash a\,[\mathsf{check}\,M \Rightarrow$

N then P else Q]
and $E, env(a$[check $M \Rightarrow N$ then P else Q]$) \vdash a$<$\vec{s}$>.
By definition, $env(a$<$\vec{s}$>$) = a\ says\ \vec{s}$,
so $E, a\ says\ \vec{s} \vdash a$[check $M \Rightarrow N$ then P else Q],
and $env(a$[check $M \Rightarrow N$ then P else Q]$) = \cdot$,
so $E \vdash a$<$\vec{s}$>.
By the typing rule, $E, a\ says\ \vec{s} \models_a$ check $M \Rightarrow N$ then P else Q.
By the typing rule, $E, a\ says\ \vec{s} \models_a M$: Un
and $E, a\ says\ \vec{s} \models_a N$: Un
and $E, a\ says\ \vec{s}, a\ says\ \vec{s} \models_a P$
and $E, a\ says\ \vec{s} \models_a Q$.
By Weakening, $E, env(Q) \vdash a$<$\vec{s}$>.
Finally By the typing rule, $E \vdash a$[Q] | a<$\vec{s}$>.

Case (a[0] $\to$ 0) Immediate from typing rule.

Case (a[P | Q] $\to a$[P] | a[Q]) Direct from typing rules.

Case (a[$\mu Z.P$] $\to a$[$P\{Z := \mu Z.P\}$]) Follows from typing rules and Substitution.

Case (a[new $n.P$] $\to$ new $n.a$[P]) Direct from typing rules.

Case (a[new b with P] $\to$ new b.(b[P] | b<$a \Rightarrow b$>))
Direct from typing rules.

Case ($G \to H$)
Assume $E \vdash G$.
By hypothesis, $G \equiv G' \to H' \equiv H$.
Finally by Lemma 36, $E \vdash H$.

Case ($G \mid H \to G' \mid H$)
Assume $E \vdash G \mid H$.
By hypothesis, $G \to G'$.
By the typing rule, $E, env(H) \vdash G$,
and $E, env(G) \vdash H$.
By induction hypothesis, $E, env(H) \vdash G'$.
By the typing rule, $E \vdash G' \mid H$.

Case (new $n{:}T.G \to$ new $G{:}T.'$)
Assume $E \vdash$ new $n{:}T.G$.
By hypothesis, $G \to G'$.
By the typing rule, $E, n{:}T \vdash G$.
By induction hypothesis, $E, n{:}T \vdash G'$.
By the typing rule, $E \vdash$ new $n{:}T.G'$.

Case (new $b.G \to$ new $b.G'$)
Similar to previous case. □

Lemma 41 (Initial Opponent Term Typability). *Let M be a term that does not contain any subterms of the form* sig $B(N)$. *Further suppose that $fn(M) \subseteq dom(E)$. If A is a E-opponent principal then $E \models_A M$:* Un.

PROOF. By definition of E-opponent principal, assume $E \models \mathbf{0}\ says\ \mathbf{1} \Rightarrow A$. By induction on the structure of M.

Cases (n), (del), (**1**), (**0**) Immediate from typing rules.

Cases ($A \wedge B$), ($A \mid B$) By typing rule and induction hypothesis.

Case ($L(\vec{N})$) By (2nd form of) typing rule and induction hypothesis,

Case (sig $B(M)$) Not present, by hypothesis.

Case (tag $B(M)$) By typing rule and induction hypothesis, noting that the axioms entail $E \models \mathbf{0}\ says\ \mathbf{1} \Rightarrow (A \mid B)$.

Cases (M.val), (M.src) By induction hypothesis.

Lemma 42 (Initial Opponent Process Typability). *Let P be a process that does not contain any subterms of the form* sig $B(N)$. *Further suppose that $fn(P) \subseteq dom(E)$. If a is a E-opponent principal then $E \models_a P$.*

PROOF. By definition of E-opponent principal, assume $E \models \mathbf{0}\ says\ \mathbf{1} \Rightarrow a$. By induction on the structure of P.

Case (0) Immediate from typing rule.

Case ($P \mid Q$) By induction hypothesis.

Case ($\mu Z.P$) By induction hypothesis.

Case (Z) Immediate from typing rule.

Case (new $n{:}T.P$) By induction hypothesis, appealing to monotonicity of inference.

Case (new b with P) By hypothesis, $b \notin E$,
and by def. of w.f.e, $E, b{:}$Un$, b\ says\ a \Rightarrow b \vdash \diamond$.
By transitivity, $E, b{:}$Un$, b\ says\ a \Rightarrow b \models \mathbf{0}\ says\ \mathbf{1} \Rightarrow b$.
By induction hypothesis, $E, b{:}$Un$, b\ says\ a \Rightarrow b \models_b P$.
Finally, by the typing rule, $E \models_a$ new b with P.

Case ($M!N$) By typing rule, and Lemma 41.

Case ($M?x.P$) By typing rule, Lemma 41 and induction hypothesis.

Case (match M as $L(\vec{N}).P$) By typing rule, Lemma 41 and induction hypothesis.

Case (learn $M \Rightarrow N.P$) By typing rule, induction hypothesis and Lemma 41, noting that $(\forall\phi)\mathbf{1}\ says\ \phi$.

Case (check $M \Rightarrow N$ then P else Q) By typing rule, Lemma 41 and induction hypothesis.

Case ($\mathbb{C}$) Immediate from typing rule.

Case (expect $\mathbb{C}$) Immediate from (the 2nd form of the) typing rule. □

Proposition 43 (Initial Opponent Typability). *Let H be an initial E-opponent configuration. Further suppose that $fn(H) \subseteq dom(E)$. Then $E \vdash H$.*

PROOF. By induction on the structure of H.

Case (0) Immediate from typing rule.

Case ($G \mid H$) By induction hypothesis.

Case $(\mathsf{new}\, n{:}T.G)$ By induction hypothesis, noting that, by definition, G is an initial $(E, n{:}T)$-opponent.

Case $(\mathsf{new}\, b.G)$ By induction hypothesis, noting that, by definition, G is an initial $(E, b{:}\mathsf{Un})$-opponent.

Case $(a[P])$ By Lemma 42.

Case $(a\langle\vec{s}\rangle)$ Immediate from the typing rule, noting that by hypothesis $E \models \mathbf{0}\ \mathit{says}\ \mathbf{1} \Rightarrow a$, and therefore $(\forall i)\ E \models a\ \mathit{says}\ M_i \Rightarrow N_i$ where $M_i \Rightarrow N_i \in \vec{s}$.

Theorem 44 (Robust Safety). *If $E \vdash G$ then G is robustly E-safe.*

Proof. Assume that $E \vdash G$, and let H be an initial E-opponent such that $G \,|\, H \rightarrow^* G' \,|\, a[\mathsf{expect}\ \mathbb{C}]$. We show that $E, \mathit{env}(G') \models \mathbb{C}$.

Let E' map every identifier in $\mathit{fn}(H) \setminus \mathit{dom}(E)$ to Un. Further, for every atomic principal $b \in \mathit{dom}(E')$ ensure that $E' \models \mathbf{0}\ \mathit{says}\ \mathbf{1} \Rightarrow b$.
By Opponent Typability, $E, E' \vdash H$.
By the typing rule, $E, E' \vdash G \,|\, H$.
By Type Preservation, $E, E' \vdash G' \,|\, a[\mathsf{expect}\ \mathbb{C}]$.
By the typing rule, $E, E', \mathit{env}(a[\mathsf{expect}\ \mathbb{C}]) \vdash G'$,
and $E, E', \mathit{env}(G') \vdash a[\mathsf{expect}\ \mathbb{C}]$.
By the typing rule, $E, E', \mathit{env}(G') \vdash_a \mathsf{expect}\ \mathbb{C}$.
By the typing rule, $E, E', \mathit{env}(G') \models \mathbb{C}$. □