Flow-Limited Authorization: Technical Report

Owen Arden Jed Liu Andrew C. Myers
Department of Computer Science
Cornell University
{owen,liujed,andru}@cs.cornell.edu

Abstract

Because information flow control mechanisms often rely on an underlying authorization mechanism, their security guarantees can be subverted by weaknesses in authorization. Conversely, the security of authorization can be subverted by information flows that leak information or that influence how authority is delegated between principals. We argue that interactions between information flow and authorization create security vulnerabilities that have not been fully identified or addressed in prior work. We explore how the security of decentralized information flow control (DIFC) is affected by three aspects of its underlying authorization mechanism: first, delegation of authority between principals; second, revocation of previously delegated authority; third, information flows created by the authorization mechanisms themselves. It is no surprise that revocation poses challenges, but we show that even delegation is problematic because it enables unauthorized downgrading. Our solution is a new security model, the *Flow-Limited Authorization Model* (FLAM), which offers a new, integrated approach to authorization and information flow control. FLAM ensures *robust authorization*, a novel security condition for authorization queries that ensures attackers cannot influence authorization decisions or learn confidential trust relationships. We discuss our prototype implementation and its algorithm for proof search.

1 Introduction

Authorization mechanisms are essential to enforcing security. However, authorization alone is not enough. First, authorization can be subverted and exploited by an adversary that can influence the delegation of authority among principals. Second, queries performed as part of an authorization check can leak confidential information. These weaknesses are examples of insecure information flows.

Conversely, information flow control is an appealing approach to building secure systems. It enables the expression of high-level information security policies describing the end-to-end behavior of the system. These policies are inherently compositional. Further, they can be formally characterized in terms of semantic security conditions such as noninterference [1], permitting rigorous proofs that enforcement mechanisms enforce policies as intended.

While control of information flow is crucial to security, it too is not enough. In particular, real systems need to be able to control the release of confidential information, but also to release that information under suitable conditions. Controlled release of information, such as through *downgrading* of information flow labels, is a violation of noninterference.

Decentralized information flow control (DIFC) [2] introduced the idea that information flow control mechanisms could control the use of downgrading mechanisms through an authorization mechanism. In a DIFC system, information flow labels are therefore expressed using the vocabulary of the authorization mechanism. For example, the original Decentralized Label Model (DLM) [2] expresses labels in terms of

principals, and delegations between principals (expressing the trust between those principals) affect which information flows are permitted. Subsequent DIFC systems use labels expressed in terms of *tags* combined with capabilities [3–6], or tags combined with principals [7].

Building on an underlying authorization mechanism adds power and expressiveness to DIFC. However, prior work has not fully explored the interactions between information flow and authorization, especially in systems in which trust can change. We refer to the collection of all delegations among principals as the system's *trust configuration*. In real systems, it is important that this trust configuration be able to change by adding or removing delegations, but we show that these changes can lead to security vulnerabilities:

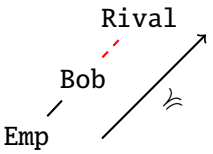
- Delegations of authority can enable information relabeling equivalent to unauthorized downgrading.
- Relabeling information limits a principal's ability to revoke access to that information.
- Changes to the trust configuration may leak information from the agent performing the change.
- Dynamic authorization queries may leak information from the querying computation.

All but the most limited existing DIFC and decentralized authorization models are susceptible to at least some of these security vulnerabilities, including several systems [8–14] designed to handle changes in trust securely. To address these vulnerabilities, we introduce a new DIFC approach that we call *flow-limited authorization*, embodied in the Flow-Limited Authorization Model (FLAM).

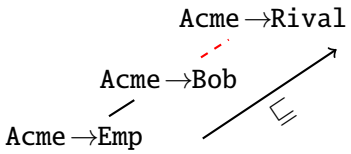
Rather than taking the trust configuration as a constant, FLAM explicitly models how information flows both through updates to the trust configuration and through the authorization mechanism itself. Our approach avoids previous restrictions [12] on trust relationships and enables fully decentralized trust in the sense that each principal's view of the trust configuration is represented and all principals' policies are enforced simultaneously.

This work makes several contributions:

- We identify new security vulnerabilities that arise from the interactions of authorization and information flow, and that are not handled satisfactorily by previous mechanisms and models. (Section 2).
- We define a new *principal algebra* that unifies principals with information flow labels, providing a clean, abstract vocabulary for exploring interactions between authorization and information flow. (Section 3).
- We provide a novel logic for making both authorization and information flow decisions securely while avoiding the newly identified vulnerabilities (Section 4)
- To characterize the interaction of authorization and information flow, we introduce *robust authorization*, a security condition that applies when delegations and revocations change the meaning of information flow policies (Section 5).
- We give a proof search algorithm for securely and efficiently processing FLAM queries (Section 6) in accordance with the logic.
- We also show that FLAM can be embedded in a programming language that supports trust management (Section 7), as well as apply our FLAM implementation to the ARBAC97 [15] role-based access control model.



(a) Trust ($\succcurlyeq$) in $\mathcal{H}$



(b) Flow ordering ($\sqsubseteq$) of policies in $\mathcal{H}$

Figure 1: Delegation loophole: By delegating to `Rival`, `Bob` effectively declassifies a label owned by `Acme`. Dashed lines indicate relationships influenced by `Bob`.

2 Motivating examples

Delegation and revocation of trust are important features of DIFC, but previous approaches fall short with respect to both their expressiveness and the security guarantees they offer. We now demonstrate three classes of vulnerabilities that arise in DIFC systems.

We motivate flow-limited authorization primarily in the context of the DLM. However, the vulnerabilities discussed in this section are generally applicable to other DIFC and authorization models, as we discuss in Section 8.

2.1 Delegation loopholes

Delegation of trust allows principals in a system to specify other principals that may act on their behalf. In addition to representing trust between two entities, delegation may encode membership in groups or roles represented by principals. Most previous models treat delegations as universally agreed-upon, but in a decentralized system, different principals can have different opinions about delegations. Most previous information flow models, including the DLM, ignore the implications of allowing the trust configuration to be controlled by partially trusted principals. As we show, a partially trusted principal can choose to delegate to an untrusted principal, and thereby achieve the effect of downgrading information even when it has not been granted the authority to downgrade. We call this use of delegation to achieve downgrading the *delegation loophole*. Some previous work [10, 12, 14] does observe this connection between delegation and downgrading, but does not eliminate the influence attackers may exert on which flows are authorized.

To see how the delegation loophole works, consider the following example of an insider attack. Bob, who works for Acme, has been enticed to disclose valuable trade secrets to Rival, one of Acme's competitors. Acme's policies[1] are written in terms of principals using the DLM [2]. In the DLM, principals like `Emp` can express membership of a group by delegating to other principals using the acts-for relation $\succcurlyeq$, where we read the expression $p \succcurlyeq q$ as "p acts for q". Thus, the DLM trust configuration consists of a set of delegations of the form $p \succcurlyeq q$, called the *principal hierarchy*. In several DIFC systems [2, 7, 16], confidentiality and integrity policies have an associated *owner* principal, expressing the authority necessary to enforce or downgrade the label. In the DLM, Acme's trade secrets might be protected under the label component `Acme` $\rightarrow$ `Emp`, which is owned by `Acme`. The label ensures that trade secrets should be readable only by employees of Acme, to whom the principal `Emp` delegates. These would include `Bob` since `Bob` $\succcurlyeq$ `Emp`.

The idea of the DLM is that only Acme itself should be able to release data labeled `Acme` $\rightarrow$ `Emp`, because `Acme` is the owner of the policy. However, if an employee like `Bob` is able to control his own delegations, he can effectively release information to a third party. For instance, Figure 1 shows how Bob

[1] We use the word "policy" here to mean a component of an information flow label governing the use of the labeled data, rather than a global system property such as noninterference.

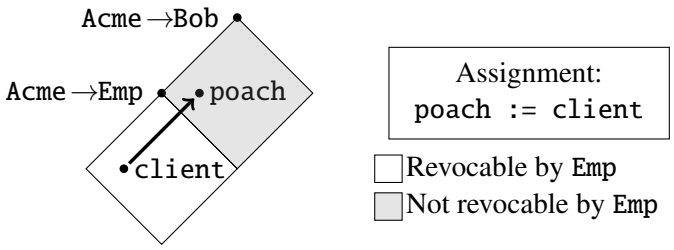


Figure 2: Poaching attack: If information at Acme →Emp is relabeled to a more restrictive policy, Emp can no longer revoke access.

might abuse his access to Acme's trade secrets. Figure 1a shows a trust configuration $\mathcal{H}$ comprising two delegations: Bob $\succcurlyeq$ Emp, and Rival $\succcurlyeq$ Bob. An edge indicates that the higher principal is trusted to act on behalf of the lower principal; the dashed line indicates that Bob has delegated trust to Rival.

Figure 1b shows the restrictiveness of information flow policies in $\mathcal{H}$. An edge indicates that the higher policy is at least as restrictive as the lower policy, or in other words, information with the lower policy may be *relabeled* to the higher policy. Bob's delegation causes Acme to believe it is safe to relabel information from the policy Acme $\rightarrow$ Emp to the policy Acme $\rightarrow$ Rival since Bob is trusted by Emp and Rival is trusted by Bob. This influence of Bob causes Acme's own system to disclose sensitive data to Rival.

Although the DLM allows the trust configuration to evolve by adding or removing delegations, it ignores the possibility that changes to the trust configuration may create insecure information flows, However, recent systems built on the DLM, such as SIF [17] and Fabric [18], give principals the power to control their delegations dynamically. These systems have therefore opened up the delegation loophole.

Surprisingly, though Bob uses delegation to cause the disclosure, the real weakness lies in how information is relabeled. Relabeling information upward in the lattice of information flow labels has heretofore been considered a safe operation requiring no privilege. This example shows that when such relabeling is justified based on a principal hierarchy, it is actually a kind of downgrading operation that must be controlled.

2.2 Poaching attacks

The presence of revocation in a DIFC system raises two challenging questions: when should revocation take effect, and what are the consequences for information flow? Answers to the first question are complicated in distributed environments where revocation messages may not be immediately disseminated. Programs with an inconsistent view of current trust relationships may make insecure authorizations.

Existing DIFC systems have particularly unsatisfying semantics with regard to the consequences of revocation. The root of the problem is that current DIFC systems permit information to flow between different policies without regard to a principal's ability to revoke access in the future. This makes it difficult to reason about what information a principal retains access to after a revocation.

Suppose Acme protects its client list with the policy Acme $\rightarrow$ Emp so that only employees may read it. Figure 2 illustrates how Bob can use his access to Emp data to "poach" Acme's client list, storing it with a more restrictive policy, to which he retains access in the event of a revocation. The white region beneath Acme $\rightarrow$ Emp represents the part of the lattice of information flow policies in which information can be relabeled to Acme $\rightarrow$ Emp. The shaded region represents information with policies that relabel to Acme $\rightarrow$ Bob, but not to Acme $\rightarrow$ Emp. The assignment poach:=client assigns the contents of variable client protected by a policy in the white region to variable poach protected by a policy in the shaded region. Since Emp delegates to Bob, policies in the white region may be relabeled to Acme $\rightarrow$ Bob.

However, relabeling information from `Acme` $\rightarrow$ `Emp` to `Acme` $\rightarrow$ `Bob` has consequences. Whereas `Emp` may revoke `Bob`'s access to information in the white region by revoking its delegation to `Bob`, it cannot revoke `Bob`'s access to information in the shaded region. Therefore, if Bob can influence what information is relabeled, he can prevent `Emp` from ever revoking access (for instance, if Bob is fired).

Like the delegation loophole, poaching attacks demonstrate that relabeling is a kind of policy downgrade exploitable by an insider. However, the two vulnerabilities differ. Delegation *enables future relabelings* to occur; therefore, to eliminate loopholes, relabelings must only be based on trusted delegations. On the other hand, relabeling *prevents future revocations* from occurring; therefore, to prevent poaching, the decision to relabel a policy should be trusted by the policy owner.

2.3 Leaking information via authorization

DIFC uses authorization decisions to decide which information flows are permitted. However, the authorization process has the potential to leak confidential information in two distinct ways.

The first source of leakage is a side channel in the authorization process. No single entity in a distributed system has a complete view of the current system state, which includes the trust configuration. Consequently, to make authorization decisions in a decentralized way, entities must *query* the current trust configuration, leading to communication. This communication may leak information to untrusted agents about what the querying process is doing or about the data it is using. For example, suppose a certain query is made only if a secret value is true; in other words, it occurs in a secret *context*. In this case, it would be insecure to query an entity not trusted to learn the secret information.

This side channel is an instance of a *read channel* [19], in which accesses to data leak information about the accessor. Read channels arising from authorization queries have been largely ignored in the DIFC literature, perhaps because the implementation platform was originally assumed to be trusted. In a fully distributed system, however, different parts of the computing infrastructure, including the implementation of the trust configuration, will in general be provided by differently trusted principals. The Fabric system therefore adds *access labels* [20] to control information flows via read channels. However, Fabric does not consider read channels arising from authorization requests.

The second source of information leakage via authorization arises if a public decision is based on the result of an authorization query whose answer depends on a secret trust relationship. Several distributed authorization systems [8, 9, 11, 13, 21–23] protect sensitive credentials with access policies, but do not constrain how credentials are used after granting access, resulting in possible leaks. These systems do not guard against authorization side channels.

A central challenge of distributed, decentralized authorization is that an entity's limited view of the trust configuration constrains its ability to securely process authorization queries. Any general approach must provide a way to bootstrap knowledge of the distributed trust configuration from local knowledge while avoiding communication that could leak information. To bootstrap this knowledge securely, we need a tighter coupling between authorization and information flow control than has been previously recognized.

2.4 Vulnerabilities in other DIFC systems

Almost all previous DIFC systems have some degree of vulnerability to the attacks described, which abuse the way authorization controls information flow. Clearly, systems based on the DLM, such as Jif [24] and Fabric [18], have these weaknesses. Capability-based DIFC systems such as Asbestos [3], Histar [4], Flume [5], and Laminar [6] also exhibit delegation loopholes and poaching attacks since processes may

transfer capabilities and relabel information. Aeolus [7] has some characteristics of capability-based systems, but maintains a trust configuration like Fabric. It too is vulnerable to these attacks.

3 Unifying principals and policies

Our goal is a simple model that supports reasoning about authorization, about information flow, and about their interactions, and that guides the construction of secure distributed systems. Our model, which we call the Flow-Limited Authorization Model (FLAM), addresses all the security issues discussed in Section 2. FLAM is both an authorization logic and an information flow model. It is an authorization logic (like [25–27]) since it derives judgments about trust. It is an information flow model (like [2, 12, 28]) since it derives judgments about secure information flow. FLAM integrates reasoning about trust and reasoning about information flow; this integration is central to preventing the security vulnerabilities identified in Section 2. We are unaware of prior models that support this kind of combined reasoning.

For simplicity, FLAM completely unifies principals, roles, privileges, and information flow labels, a perhaps surprising feature that distinguishes FLAM from previous models for either authorization or information flow[2]. In FLAM, principals are *both* authorization entities *and* information flow policies enforcing confidentiality and integrity. In subsequent discussion, we sometimes use *label* (or *policy*) to talk about a *principal* used to specify permitted information flow, but these concepts are interchangeable in FLAM. As we show, unifying principals with information flow labels enables a simpler, algebraic presentation of the relationships between information flow policies and the principals they concern.

This section provides the formal basis for unifying authority and decentralized information flow policies. Although the algebraic definitions given in this section may appear complex at first, we show in Section 4 that they enable a concise logic, collected in Figures 5 and 6. Authorization decisions derived from this logic are protected from the problems discussed in Section 2.

3.1 Authority projections

All entities in a system are represented as principals that may delegate to each other. FLAM provides a particularly rich set of principals. We construct this set of principals by defining operations on principals that combine or attenuate principals in different ways.

Let $\mathcal{N}$ be the set of all primitive principals, which are essentially uninterpreted names. Starting from primitive principals, we can construct more complex compound principals. For any two principals p and q, we represent the conjunction of their authority, the authority of *both* p and q, as the compound principal $p \wedge q$. Likewise, the authority of *either* p or q is written $p \vee q$. These conjunction and disjunction operators, as in Boolean algebra, define a lattice[3] over principals. If a principal q trusts principal p, then we say p *acts for* q and write $p \succcurlyeq q$. If q represents the privilege or permission to perform an action, the statement $p \succcurlyeq q$ means p has the right to perform that action. Lattice properties imply $p \wedge q \succcurlyeq p \succcurlyeq p \vee q$ for any p and q.

Conjunction and disjunction are already familiar from previous logics for authentication and authorization, and the acts-for relation of FLAM is related to the speaks-for relation of authentication logics [25, 27], but Section 4.4 draws a distinction between the speaks-for relation for FLAM and the acts-for relation.

[2] Some prior work has unified *roles* with information flow labels, while distinguishing principals from roles [12, 14, 29].

[3] Authorization logics typically treat $\top$ as the *least* trusted principal and use the symbol $\wedge$ to represent conjunctive principals, which denote lattice meets. DIFC models often use $\top$ to represent the *most* trusted principal, yet retain the $\wedge$ notation for conjunctive principals even though they correspond to lattice joins. We find treating conjunctions of authority as "higher" to be intuitive, and adopt the DIFC approach.

In many DIFC models, the flows-to ordering $\sqsubseteq$ between information flow policies derives from an ordering on principals that is similar to $\succcurlyeq$. Rather than defining a separate space of information flow policies, we characterize confidentiality and integrity as a limited form of *authority*. For a principal p, let $p^{\rightarrow}$ represent its *read authority*, and $p^{\leftarrow}$ represent its *write authority*. Separating these components of p's authority allows us to think of information flow policies as delegations by one or both of these attenuated principals. For instance, delegating authority $p^{\rightarrow}$ to q grants q read-only access to p's data.

FLAM generalizes this idea of attenuating a principal's authority by defining operations called *authority projections*, which allow new attenuated principals to be constructed from existing principals. In FLAM, we represent p's read authority ($p^{\rightarrow}$) and its write authority ($p^{\leftarrow}$) as projections.

Definition 1 (Authority projections)**.** An *authority projection*, π, is an operation on principals such that for any principal p, p^{π} is a principal, and

1. $p \succcurlyeq p^{\pi}$
2. $p \succcurlyeq q \implies p^{\pi} \succcurlyeq q^{\pi}$
3. $p^{\pi} \wedge q^{\pi} = (p \wedge q)^{\pi}$
4. $p^{\pi} \vee q^{\pi} = (p \vee q)^{\pi}$
5. $(p^{\pi})^{\pi} = p^{\pi}$

These five properties capture the essence of limited authority derived from a principal's general authority, without requiring separate classes of entities such as *roles* [30], *subprincipals*, or *groups* [26]. Naturally, the originating principal acts for the derived authority (1), and projection preserves the properties of the authorization lattice (2, 3, 4). Finally, projections are idempotent (5).

FLAM defines two classes of authority projections, *basis projections* and *ownership projections*. Basis projections define the different kinds of authority a principal may possess, i.e., confidentiality and integrity, whereas ownership projections (discussed in Section 3.3) attenuate a principal's authority relative to other principals.

For the purpose of this paper, all authority is representable as a combination of confidentiality and integrity authority. In other words, the conjunctive principal $p^{\rightarrow} \wedge p^{\leftarrow}$ has authority equivalent to p, meaning that confidentiality and integrity projections form a kind of *basis* for authority.

Definition 2 (Confidentiality and integrity basis)**.** Let $\rightarrow$ and $\leftarrow$ be authority projections such that, for all principals

1. $p = p^{\rightarrow} \wedge p^{\leftarrow}$
2. $(p^{\leftarrow})^{\rightarrow} = (p^{\rightarrow})^{\leftarrow} = \bot$
3. $p^{\rightarrow} \vee q^{\leftarrow} = \bot$

We represent all authority as a combination of confidentiality and integrity authority (1), so any principal that acts for both projections of a principal also acts for the principal. Additionally, composing (2) or taking the meet (3) of confidentiality and integrity projections yields $\bot$. In this paper, we focus on information flow policies for confidentiality and integrity, but we expect it is possible to extend FLAM with additional projections that represent other aspects of security. For instance, [31] adds *availability* policies, and [32] includes *reference authority* and *persistence* policies. We leave representing such policies as basis projections to future work.

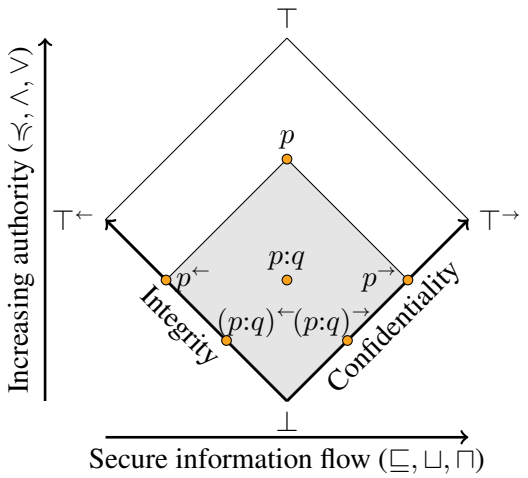


Figure 3: The FLAM lattices for trust and information flow

Using the above operations, we can extend the set of primitive principals to create a richer set of principals ordered by $\succcurlyeq$. Let $\mathcal{P}_0$ be the closure of $\mathcal{N}$ under the operations $\wedge$ and $\vee$, and the projections $\leftarrow$ and $\rightarrow$. We can construct a lattice from the preorder $\succcurlyeq$ in the usual way, by defining an equivalence relation $a \equiv_{\succcurlyeq} b \iff (a \succcurlyeq b \text{ and } b \succcurlyeq a)$ and grouping equivalent principals into a single lattice element representing an equivalence class. Then $\mathcal{P}_0$ induces a lattice $(\mathcal{P}_0, \succcurlyeq)$ where we define $\top$ and $\bot$ as distinguished principals with highest and lowest authority, respectively. Joins in $\mathcal{P}_0$ are the conjunctions of principals ($\wedge$), and meets are disjunctions ($\vee$).

3.2 The information flow ordering

The value of authority projections is that they allow secure information flow to be represented as authority relationships in a simple and natural way. In fact, there is no explicit need for a separate lattice of information flow policies; we could express information flow entirely by authority relationships. It is often convenient, however, to have notation for the authority ordering on principals as well as the information flow ordering on principals. Below, we define an information flow lattice whose ordering and operations are syntactic sugar for authority relationships and operations in the authority lattice.

For principals p and q, we say *p flows to q*, written $p \sqsubseteq q$, if p acts for q's integrity (q trusts information from p) and q acts for p's confidentiality (p trusts q to protect p's secrets). In the definition below, these relationships are represented simultaneously by conjunctions of authority projections.

Definition 3 (Secure information flow as authorization)**.**

$$p \sqsubseteq q \overset{\triangle}{\iff} q^{\rightarrow} \wedge p^{\leftarrow} \succcurlyeq p^{\rightarrow} \wedge q^{\leftarrow}$$
$$p \sqcup q \overset{\triangle}{\iff} (p \wedge q)^{\rightarrow} \wedge (p \vee q)^{\leftarrow}$$
$$p \sqcap q \overset{\triangle}{\iff} (p \vee q)^{\rightarrow} \wedge (p \wedge q)^{\leftarrow}$$

The flows-to relation $\sqsubseteq$ is a preorder, so we can lift it to a partial order just as we did for acts-for, with equivalences defined by $a \equiv_{\sqsubseteq} b \iff (a \sqsubseteq b \text{ and } b \sqsubseteq a)$. The relation $\sqsubseteq$ induces an *information flow* lattice $(\mathcal{P}_0, \sqsubseteq)$. In this lattice, we represent joins by $\sqcup$ and meets by $\sqcap$. The top element of $(\mathcal{P}_0, \sqsubseteq)$ is the

policy that most restricts use of the information, *secret* and *untrusted*: $\top^{\rightarrow} \wedge \bot^{\leftarrow}$. The bottom element is the least restrictive policy, *public* and *trusted*: $\bot^{\rightarrow} \wedge \top^{\leftarrow}$. We often omit projections of the $\bot$ principal to obtain the more concise (but equivalent) principal representation; e.g., $p^{\rightarrow}$ instead of $p^{\rightarrow} \wedge \bot^{\leftarrow}$ and $p^{\leftarrow}$ instead of $\bot^{\rightarrow} \wedge p^{\leftarrow}$.

By the definitions above, the equivalence classes of $\succcurlyeq$ and $\sqsubseteq$ are identical, and there is a one-to-one correspondence between the elements of $(\mathcal{P}_0, \succcurlyeq)$ and $(\mathcal{P}_0, \sqsubseteq)$, even though the two orderings are "at right angles" to each other. Figure 3 illustrates this correspondence by aligning both lattices on the same set of elements. Secure information flow is from left to right, toward increasing confidentiality and decreasing integrity. The trust ordering is bottom to top, toward increasing authority. This correspondence allows us to easily translate relationships from one ordering to another when convenient.

3.3 Owned principals

To give FLAM the expressive power of some previous authorization systems, such as *role-based access control* (RBAC) [30] and the DLM [2], we introduce another way to construct principals. In RBAC, principals are assigned *roles* which they may select when performing sensitive tasks, and access control policies are specified in terms of roles that are permitted access. It is tempting to use delegation to express authorization concepts such as roles and *groups* [26]. However, this approach fails to adequately control modification of role membership. For instance, if Acme uses the principal `Emp` to represent a role by delegating to all Acme employees, then Bob can effectively add employees via delegation. What Acme requires is a way to refer to principals like Bob while retaining control over their trust relationships. Then a principal like `Emp` can delegate to such a principal without risking subversion of its authorization mechanism.

From the perspective of information flow control, the principals from the set $\mathcal{P}_0$ can represent both authority and information flow policies, but the information flow policies expressible with these principals are rather limited—they are not *decentralized* in the sense of the DLM [2]. The key aspect of decentralized policies is that policy *owners* retain control over decisions to release information.

In FLAM, we express ownership as a special class of authority projections called *ownership projections*. The *owned principal* `Acme:Bob` represents[4] Bob as a principal whose trust relationships Acme retains control of. Intuitively, `Acme:Bob` delegates trust to the same principals as Bob, but only if Acme allows the delegation. Acme may also create new delegations of trust from `Acme:Bob` even though Acme doesn't act for Bob. Owned principals are similar in spirit to *roles* [30], *groups*, and *subprincipals* [26], but are first-class principals that may delegate and be delegated to.

Owned principals are useful for representing decentralized information flow policies. For instance, the principal $(p{:}q)^{\rightarrow}$ is a confidentiality projection of the ownership projection $p{:}q$. This principal represents a confidentiality policy owned by p that specifies q as a reader, and is similar to the DLM policy $p \rightarrow q$. In the DLM, $p \rightarrow q \sqsubseteq r \rightarrow s$ if and only if $r \succcurlyeq p$ and $s \succcurlyeq q$. FLAM permits finer-grained delegations of trust, so the relationship $(p{:}q)^{\rightarrow} \sqsubseteq (r{:}s)^{\rightarrow}$ holds, for example, if $r{:}s \succcurlyeq p{:}q$ but also if $r \succcurlyeq p$ and $s^{\rightarrow} \succcurlyeq q^{\rightarrow}$.

Definition 4 formalizes the properties of ownership that unify decentralized policies with principal authority.

Definition 4 (Ownership projection)**.** For each principal p let $:p$ be a distinguished authority projection, an *ownership projection*. We say $p{:}q$ is an *owned principal* and p is the *owner* of $p{:}q$. Owned principals satisfy the following properties:

1. $p \succcurlyeq r$ and $q \succcurlyeq s \implies p{:}q \succcurlyeq r{:}s$

[4]For better readability and to resemble DLM notation, we abuse the syntax of authority projections and write $p{:}q$ instead of $p^{:q}$.

2. $p \succcurlyeq r$ and $q \succcurlyeq r{:}s \implies p{:}q \succcurlyeq r{:}s$
3. $p{:}p = p$
4. $p{:}\bot = \bot$
5. $p{:}r \wedge p{:}s = p{:}(r \wedge s)$
6. $p{:}r \vee p{:}s = p{:}(r \vee s)$
7. $p{:}q^{\pi} = (p{:}q)^{\pi}$ for $\pi \in \{\leftarrow, \rightarrow\}$
8. $p^{\pi}{:}q = (p{:}q)^{\pi}$ for $\pi \in \{\leftarrow, \rightarrow\}$

The principal $p{:}q$ is a principal that represents q but that p, the owner, retains control over. Specifically, since $:q$ is an authority projection, p acts for $p{:}q$. Principal $p{:}q$ reflects the delegations of both p and q, so owned principals are similar to disjunctive principals, but are not commutative: $p{:}q \neq q{:}p$. Property (1) permits a delegation between unowned principals ($q \succcurlyeq s$) to induce one between corresponding owned principals ($p{:}q \succcurlyeq r{:}s$), but only if the owners also have an acts-for relationship ($p \succcurlyeq r$). This condition on owners is central to the idea of ownership since it prevents a delegation to an owned principal $p{:}q$ from implying a delegation to the corresponding unowned principal q. Similarly, property (2) ensures a delegation from an owned principal $r{:}s$ to an unowned principal q induces a similar delegation to a corresponding owned principal $p{:}q$, but only if the owners have an acts-for relationship ($p \succcurlyeq r$).

An ownership projection $:p$ is the identity when applied to the principal p that defines it (3), and applying the bottom ownership projection $:\bot$ always yields $\bot$ (4). Finally, conjunction and disjunction distribute through ownership (5, 6), and confidentiality and integrity projections are associative with and commute with ownership projections (7, 8).

Using ownership projections, we can further extend our set of principals. Let $\mathcal{O} = \{:p \mid p \in \mathcal{P}_0\}$ be a set of ownership projections. Then let $\mathcal{P}$ be the closure of $\mathcal{P}_0$ under the projections in $\mathcal{O}$. Like $\mathcal{P}_0$, the equivalence classes of $\mathcal{P}$ form lattices $(\mathcal{P}, \succcurlyeq)$ and $(\mathcal{P}, \sqsubseteq)$, whose elements have a one-to-one correspondence. Figure 3 relates an owned principal, $p{:}q$ and its projections, to the other elements of these lattices. For the remainder of this paper, principals are implicitly members of the set $\mathcal{P}$ unless otherwise specified.

3.4 FLAM normal form

Constructing efficient algorithms for manipulating elements of an algebraic system such as FLAM is much easier when the elements have a normal form. A normal form for FLAM principals can be obtained from the equational rules and lattice properties already stated. Using these rules, any FLAM principal can be factored into the join of a confidentiality projection and an integrity projection $p^{\rightarrow} \wedge q^{\leftarrow}$, where p and q are each a join of meets of owned or primitive principals.

Definition 5. A FLAM principal p is in *normal form* if it is accepted by the following grammar where $n \in \mathcal{N}$.

$$\begin{array}{ll} p ::= J^{\rightarrow} \wedge J^{\leftarrow} & M ::= L \mid L \vee M \\ J ::= M \mid M \wedge J & L ::= n \mid L{:}L \end{array}$$

Our prototype implementation, discussed in Section 6, includes an algorithm for converting FLAM principals to normal form. This algorithm is relatively straightforward: it applies lattice properties and equational rules of authority projections as rewrite rules to reduce principals to normal form. We have formalized and proved this algorithm correct in Coq, but omit discussion of it here for the sake of brevity.

4 Secure reasoning with dynamic trust

In this section, we present the FLAM system model and a set of inference rules for deriving authorization decisions from the distributed system state. Unlike most previous models, FLAM does not presume universally agreed-upon trust relationships. Instead, principals may regard a trust relationship (i.e., delegation) to be untrustworthy, or may wish to prevent others from learning of its existence. Furthermore, principals do not have a global view of the system state and must communicate with other principals to discover new relationships. These attributes make FLAM an appropriate model for authorization in distributed systems.

4.1 System model and trust configuration

Our goal is to model the security of a distributed system comprising various host nodes that keep track of different parts of the system's trust configuration. In FLAM, these nodes, like all other entities in the system, are represented as principals. Thus, a host node is a primitive principal in $\mathcal{N}$; we use n and c to denote such principals. We treat the trust configuration $\mathcal{H}$ as a distributed data structure, wherein each fragment $\mathcal{H}(n)$ is the *delegation set* stored at node n. Each delegation $(p \succcurlyeq q, \ell)$ has an associated *delegation label* ℓ expressing the confidentiality and integrity of the delegation.

Definition 6 (FLAM trust configurations)**.** A *trust configuration* $\mathcal{H}$ is a map from principals $n \in \mathcal{N}$ to delegation sets. A *delegation set* is a set of tuples of the form $(p \succcurlyeq q, \ell)$ where p, q, ℓ are principals in $\mathcal{P}$.

For example, a delegation $(p \succcurlyeq q, n^{\leftarrow})$ might be hosted by principal n; in other words, $(p \succcurlyeq q, n^{\leftarrow}) \in \mathcal{H}(n)$. The delegation label $n^{\leftarrow}$ means that the delegation is public (since $(n^{\leftarrow})^{\rightarrow} = \bot$) and has the integrity of n. We make no well-formedness assumptions about $\mathcal{H}$; for instance, a malicious node n might store the delegation $(n \succcurlyeq \top, \top^{\leftarrow})$.

This abstraction allows us to reason about information flow in the trust configuration without exposing the details of the underlying distributed data structure. For instance, $\mathcal{H}(n)$ might represent a remote call interface for requesting derived delegations from n, or it might represent delegations stored or replicated at n that can be fetched on demand.

4.2 Flow-limited judgments

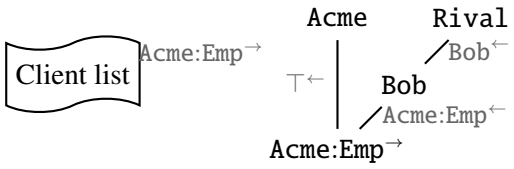
Authorization queries are submitted to principals that process them by using local data, by obtaining remote data via communication with other principals, or by a combination of both. The answers to queries are used to determine the relationships that currently exist between principals in the given trust configuration $\mathcal{H}$.

Queries take the form of judgments; positive query results carry proofs (or derivations) of these judgments. Derivation rules specify how to obtain proofs given a set of delegations. One approach would be to represent judgments with the form $D \vdash p \succcurlyeq q$, meaning that the relationship $p \succcurlyeq q$ holds assuming the delegations in D.

However, constructing a proof in a distributed system creates information flows. Consequently, this form of judgment has two fundamental problems. First, it fails to characterize the confidentiality and integrity of the conclusion $p \succcurlyeq q$. Second, the conclusion is the result of a distributed computation over the hosts that collectively store the trust configuration $\mathcal{H}$, so communicating with these hosts to obtain the delegations in D could leak confidential information about the query or permit poaching attacks by the query's issuer.

FLAM solves both problems by parameterizing authorization queries with policies that restrict the flow of information as the query is answered. The resulting *flow-limited judgments* have the following form:

$$\mathcal{H}; c; \mathit{pc}; \ell \vdash p \succcurlyeq q$$



$$\boxed{\mathcal{H}; c; \texttt{Acme:Emp}^{\leftarrow}; \texttt{Acme:Emp}^{\leftarrow} \not\Vdash \texttt{Acme:Emp}^{\rightarrow} \sqsubseteq \texttt{Rival}^{\rightarrow}} \quad (1)$$

$$\boxed{\mathcal{H}; c; \texttt{Bob}^{\leftarrow}; \texttt{Acme:Emp}^{\leftarrow} \not\Vdash \texttt{Acme:Emp}^{\rightarrow} \sqsubseteq \texttt{Bob}^{\rightarrow}} \quad (2)$$

Figure 4: Section 2 attacks prevented. The boxed judgments do not hold robustly with the illustrated delegations. Judgment (1) does not hold since Bob's delegation to Rival cannot be used to robustly relabel Acme's policies, closing the delegation loophole. In (2), the query label $\texttt{Bob}^{\leftarrow}$ has insufficient integrity to relabel Acme's policies, preventing Bob from poaching the client list.

Here, $\mathcal{H}$ is the trust configuration and $c \in \mathcal{N}$ is the *current host* performing the derivation. The policy ℓ is the *derivation label*, which is an upper bound in $(\mathcal{P}, \sqsubseteq)$ for all delegation labels of delegations used in the derivation. The label *pc* is the *query label*, which is an upper bound in $(\mathcal{P}, \sqsubseteq)$ on the confidentiality and integrity of the query. For remotely issued queries, the integrity of the originating host must flow to the query label, and the query label must flow to the confidentiality of any host that is contacted during the derivation.

Flow-limited judgments are constructed by inspecting the delegations in $\mathcal{H}$. Accesses to local delegations, i.e. $\mathcal{H}(c)$, are not externally observable, but principals may also communicate with any host $n \in \mathrm{dom}(\mathcal{H})$ to obtain judgments derived from remote delegations. We abbreviate judgments that hold in any trust configuration, or *statically*, as $\vdash p \succcurlyeq q$. For instance, $\vdash p \wedge q \succcurlyeq q$ holds statically.

As with the trust configuration $\mathcal{H}$, we make no well-formedness assumptions about the query label or derivation label specified in authorization queries. However, to protect their own security, we assume that honest hosts specify a query label for top-level queries that characterizes the confidentiality and integrity of the issuing context; hence the name *pc* for the *program counter* label, as in Jif [24]. Likewise, we assume honest hosts will treat query results in accordance with the derivation label. In our technical report [33], we describe a programming language whose type system verifies these assumptions.

4.3 Robust derivations

Tracking information flow through judgments is only the first step—we still need to eliminate delegation loopholes and poaching attacks.

Consider the example of Section 2.1. We can model this scenario with the delegation set shown in Figure 4. Acme grants Bob read-only access with the delegation $(\texttt{Bob} \succcurlyeq \texttt{Acme:Emp}^{\rightarrow}, \texttt{Acme:Emp}^{\leftarrow})$. As before, Bob delegates to Acme's competitor Rival.

Delegation loopholes arise when attackers influence the derivation of sensitive queries—when derivations are not *robust*. In the example, we can close the loophole by eliminating the influence of attackers like Bob on the derivation of queries about who acts for Acme's principals. If Bob's delegation cannot be used in the proof of a query like $\texttt{Rival}^{\rightarrow} \succcurlyeq \texttt{Acme:Emp}^{\rightarrow}$, then the proof is *robust*, and Bob cannot influence whether $\texttt{Acme:Emp}^{\rightarrow}$ can flow to $\texttt{Rival}^{\rightarrow}$.

FLAM's derivation labels allow Acme to constrain Bob's influence on the derivation. Consider the

following judgment, which holds in our example trust configuration.

$$\mathcal{H}; c; pc; \texttt{Acme:Emp}^{\leftarrow} \vdash \texttt{Bob} \succcurlyeq \texttt{Acme:Emp}^{\rightarrow}$$

It has integrity $\texttt{Acme:Emp}^{\leftarrow}$, so any derivation of this judgment can only depend on delegations that have Acme:Emp's integrity or greater in the authority ordering ($\succcurlyeq$). In contrast, there is no robust proof of the following judgment since using Bob's delegation would result in a proof with lower integrity than $\texttt{Acme:Emp}^{\leftarrow}$.

$$\mathcal{H}; c; pc; \texttt{Acme:Emp}^{\leftarrow} \not\vdash \texttt{Rival} \succcurlyeq \texttt{Acme:Emp}$$

Poaching attacks arise when attackers influence the *decision* to relabel information—that is, when they influence the context of a query. The query label represents the information flow context of such a query, so by restricting this label, FLAM prevents attackers from poaching information.

For instance, Figure 4 shows Acme's client list labeled with confidentiality $\texttt{Acme:Emp}^{\rightarrow}$. Suppose Bob wants to copy this list to a file with confidentiality $\texttt{Bob}^{\rightarrow}$ so he can maintain access if he is fired. To do so, Acme's system requires that the following judgment holds.

$$\mathcal{H}; c; \texttt{Acme:Emp}^{\leftarrow}; \texttt{Acme:Emp}^{\leftarrow} \vdash \texttt{Bob} \succcurlyeq \texttt{Acme:Emp}$$

This judgment is immune to poaching attacks since neither the result nor the query itself is influenced by Bob. Bob cannot independently issue such a query since his influence would taint the query label, shown below.

$$\mathcal{H}; c; \texttt{Acme:Emp}^{\leftarrow} \vee \texttt{Bob}^{\leftarrow}; \texttt{Acme:Emp}^{\leftarrow} \vdash \texttt{Bob} \succcurlyeq \texttt{Acme:Emp}$$

This query has insufficient authority to robustly relabel $\texttt{Acme:Emp}^{\rightarrow}$ to $\texttt{Bob}^{\rightarrow}$. This prevents Bob from poaching Acme's client list, giving Acme control of what information is released to Bob.

One might wonder why Acme requires $\texttt{Bob} \succcurlyeq \texttt{Acme:Emp}$ to hold instead of $\texttt{Bob}^{\rightarrow} \succcurlyeq \texttt{Acme:Emp}^{\rightarrow}$. The answer illustrates a fundamental difference between information flow control and access control. Specifically, Acme wants to know whether it is safe to enforce information labeled $\texttt{Acme:Emp}^{\rightarrow}$ with the policy $\texttt{Bob}^{\rightarrow}$. This is a distinct goal from access control since Acme not only cares about the access to the client list, but also the propagation of that data. Even though Bob cannot influence whether $\texttt{Acme:Emp}^{\rightarrow} \sqsubseteq \texttt{Bob}^{\rightarrow}$, he *does* control what $\texttt{Bob}^{\rightarrow}$ flows to. Thus, Acme wants to ensure that Bob has sufficient integrity to enforce the confidentiality of the client list. Since he does not, Acme should deny any request to relabel $\texttt{Acme:Emp}^{\rightarrow}$ to $\texttt{Bob}^{\rightarrow}$.

4.4 Speaking for other principals

Prior work on *robust downgrading* [18,34,35] of information flow policies places constraints on the influence an attacker may have on declassification and endorsement. Specifically, a principal should not be able to leak information by influencing downgrading decisions. Here, we seek similar constraints, but on information flow authorizations in general, whether they represent a downgrade or not.

In FLAM, the *voice* of a principal q, written $\nabla(q)$, defines the minimum integrity required to influence the flow of information labeled q.

Definition 7 (Principal voice)**.** For a principal in normal form $p^{\rightarrow} \wedge q^{\leftarrow}$, the *voice* of $p^{\rightarrow} \wedge q^{\leftarrow}$ is defined as

$$\nabla(p^{\rightarrow} \wedge q^{\leftarrow}) \triangleq p^{\leftarrow} \wedge q^{\leftarrow}$$

$$[\textsc{Bot}] \quad C \vdash p \succcurlyeq \bot \qquad [\textsc{Top}] \quad C \vdash \top \succcurlyeq p \qquad [\textsc{Refl}] \quad C \vdash p \succcurlyeq p \qquad [\textsc{Proj}] \quad \frac{C \vdash p \succcurlyeq q}{C \vdash p^{\pi} \succcurlyeq q^{\pi}}$$

$$[\textsc{ProjR}]\ C \vdash p \succcurlyeq p^{\pi} \qquad [\textsc{Own1}]\ \frac{C \vdash o \succcurlyeq o' \quad C \vdash p \succcurlyeq p'}{C \vdash o{:}p \succcurlyeq o'{:}p'} \qquad [\textsc{Own2}]\ \frac{C \vdash o \succcurlyeq o'}{C \vdash p \succcurlyeq o'{:}p'} \qquad [\textsc{ConjL}]\ \frac{C \vdash p_k \succcurlyeq p \quad k \in \{1,2\}}{C \vdash p_1 \wedge p_2 \succcurlyeq p}$$

$$[\textsc{ConjR}]\ \frac{C \vdash p \succcurlyeq p_1 \quad C \vdash p \succcurlyeq p_2}{C \vdash p \succcurlyeq p_1 \wedge p_2} \qquad [\textsc{DisjL}]\ \frac{C \vdash p_1 \succcurlyeq p \quad C \vdash p_2 \succcurlyeq p}{C \vdash p_1 \vee p_2 \succcurlyeq p} \qquad [\textsc{DisjR}]\ \frac{C \vdash p \succcurlyeq p_k \quad k \in \{1,2\}}{C \vdash p \succcurlyeq p_1 \vee p_2}$$

$$[\textsc{Trans}]\ \frac{C \vdash p \succcurlyeq q \qquad C \vdash q \succcurlyeq r}{C \vdash p \succcurlyeq r} \qquad [\textsc{Del}]\ \frac{(p \succcurlyeq q, \ell) \in \mathcal{H}(c)}{\mathcal{H}; c; \mathit{pc}; \ell \vdash p \succcurlyeq q} \qquad [\textsc{Fwd}]\ \frac{\mathcal{H}; c; \mathit{pc}; \ell \Vdash n \succcurlyeq \mathit{pc}^{\rightarrow} \wedge \ell \quad \mathcal{H}; n; \mathit{pc} \sqcup \ell \sqcup c^{\leftarrow}; \ell \sqcap c^{\rightarrow} \vdash p \succcurlyeq q}{\mathcal{H}; c; \mathit{pc}; \ell \vdash p \succcurlyeq q}$$

$$[\textsc{Weaken}]\ \frac{\mathcal{H}; c; \mathit{pc}'; \ell' \vdash p \succcurlyeq q \quad \mathcal{H}; c; \mathit{pc} \sqcup \ell'; \ell \Vdash \mathit{pc} \sqsubseteq \mathit{pc}' \quad \mathcal{H}; c; \mathit{pc} \sqcup \ell'; \ell \Vdash \ell' \sqsubseteq \ell}{\mathcal{H} \cup \mathcal{H}'; c; \mathit{pc}; \ell \vdash p \succcurlyeq q}$$

Figure 5: Inference rules for flow-limited judgments. For brevity, C denotes the context $\mathcal{H}; c; \mathit{pc}; \ell$. The union of trust configurations is defined pointwise: $(\mathcal{H} \cup \mathcal{H}')(n) = \mathcal{H}(n) \cup \mathcal{H}'(n)$.

As its name suggests, the voice of a principal is related to the *speaks-for* relation [25, 27] found in authorization logics. In these models, if Bob speaks for Alice (sometimes written Bob $\Rightarrow$ Alice) and Bob says some proposition P is true, then Alice also says P is true. Flow-limited judgments permit a refinement of speaks-for since we can reason directly about the influence of principals on authorization decisions. In FLAM, a principal's voice is the integrity needed to speak on its behalf, so Bob speaks for Alice if $\texttt{Bob} \succcurlyeq \nabla(\texttt{Alice})$.

This version of speaks-for differs from that in other authorization logics. First, it derives from the integrity of principals and the acts-for relationships between them. Second, the speaks-for relation is transitive, but not reflexive. Notice that $\texttt{Acme}^{\rightarrow}$ does not speak for itself.

As in [27], FLAM's speaks-for relation distinguishes the concepts of *speaking for* and *acting for* a principal. Previous DIFC models [2] have considered these concepts to be similar, but they are distinct in FLAM to support reasoning separately about the confidentiality and integrity of principals. For instance, the principal $\texttt{Acme}^{\leftarrow}$ speaks for both Acme and $\texttt{Acme}^{\rightarrow}$, but acts for neither.

To provide end-to-end information flow security, FLAM distinguishes *robust judgments* that hold with sufficient integrity to speak on behalf of the principals involved. Robust judgments in FLAM are identified by the symbol $\Vdash$. FLAM's inference rules, discussed below, use robust judgments to ensure that all derivations exhibit robust information flow.

4.5 Rules for flow-limited reasoning

Figure 5 gives inference rules for deriving flow-limited judgments. Most rules are straightforward, encoding properties of conjunctions (rules ConjL, ConjR), disjunctions (rules DisjL, DisjR), authority projections

$$[\text{R-STATIC}]\quad \frac{\vdash p \succcurlyeq q}{C \Vdash p \succcurlyeq q} \qquad [\text{R-LIFT}]\quad \frac{\begin{array}{c}\mathcal{H}; c; pc; \ell \wedge \nabla(q) \vdash p \succcurlyeq q \\ \mathcal{H}; c; pc; \ell \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \\ \mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(q)\end{array}}{\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q}$$

$$[\text{R-LIFTPC}]\ \frac{\mathcal{H}; c; pc; \ell \wedge \nabla(q) \vdash pc \succcurlyeq \nabla(q)}{\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(q)} \qquad [\text{R-CONJR}]\ \frac{C \Vdash p \succcurlyeq p_1 \quad C \Vdash p \succcurlyeq p_2}{C \Vdash p \succcurlyeq p_1 \wedge p_2} \qquad [\text{R-DISJL}]\ \frac{C \Vdash p_1 \succcurlyeq p \quad C \Vdash p_2 \succcurlyeq p}{C \Vdash p_1 \vee p_2 \succcurlyeq p}$$

$$[\text{R-TRANS}]\ \frac{\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q \qquad \mathcal{H}; c; pc; \ell \Vdash q \succcurlyeq r \quad \mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(r^{\rightarrow})}{\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq r} \qquad [\text{R-FWD}]\ \frac{\mathcal{H}; c; pc; \ell \Vdash n \succcurlyeq pc^{\rightarrow} \wedge \ell \wedge \nabla(q) \quad \mathcal{H}; n; pc \sqcup \ell \sqcup c^{\leftarrow}; \ell \sqcap c^{\rightarrow} \Vdash p \succcurlyeq q}{\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q}$$

$$[\text{R-WEAKEN}]\ \frac{\begin{array}{c}\mathcal{H}; c; pc'; \ell' \Vdash p \succcurlyeq q \\ \mathcal{H}; c; pc \sqcup \ell'; \ell \Vdash pc \sqsubseteq pc' \\ \mathcal{H}; c; pc \sqcup \ell'; \ell \Vdash \ell' \sqsubseteq \ell\end{array}}{\mathcal{H} \cup \mathcal{H}'; c; pc; \ell \Vdash p \succcurlyeq q}$$

Figure 6: Inference rules for robust judgments.

(rules PROJ and PROJR), ownership projections (rules OWN1, OWN2), and lattices in general (rules BOT, TOP, REFL, TRANS). The DEL rule allows the use of a local delegation if its label matches the derivation label of the context.

The WEAKEN rule allows judgment contexts to be weakened. If $p \succcurlyeq q$ is derivable with trust configuration $\mathcal{H}$ and bounds $pc'; \ell'$, then it is still derivable after adding delegations[5] to $\mathcal{H}$ or increasing the restrictiveness of the bounds ($pc \sqsubseteq pc'$ and $\ell' \sqsubseteq \ell$).

Like any other relabelings that use dynamic trust relationships, attackers might try to abuse these relabelings of pc and ℓ'. For example, Bob could try use WEAKEN to hide his influence on a judgment by boosting its derivation label from $\texttt{Acme:Emp}^{\leftarrow} \sqcup \texttt{Bob}^{\leftarrow}$ to $\texttt{Acme:Emp}^{\leftarrow}$, or he could try to reduce a judgment's confidentiality by downgrading its derivation label from $\texttt{Acme:Emp}^{\rightarrow} \sqcup \texttt{Bob}^{\rightarrow}$ to $\texttt{Bob}^{\rightarrow}$. The rule prevents this by requiring the relabelings to be robust. Because these robustness proofs are only attempted after the relabeled judgment is proved, their query labels ($pc \sqcup \ell'$) are tainted with the derivation label ℓ' of the relabeled judgment.

The FWD rule is used to derive acts-for judgments via remote hosts. The first premise ensures that c can prove the remote host n is trusted to protect both the query's confidentiality and its derivation label. In the second premise, n derives the desired relationship with a query label that is tainted both with c's integrity and with the derivation label of the first premise. To ensure c can see the result, the derivation label is attenuated by c's confidentiality. If these premises hold, then n can release the result to c, and c can trust it at label ℓ, therefore c can conclude that the relationship holds.

The rules for reasoning about robust judgments are shown in Figure 6. The first three rules specify how robust judgments derive from non-robust judgments. Rule R-STATIC permits static judgments to be treated as robust judgments in any context, whereas rule R-LIFT derives robust judgments from dynamic judgments. The first premise of R-LIFT ensures the judgment holds with the voice $\nabla(q)$ of the delegating

[5]The union of two trust configurations is defined to take their pointwise union: $(\mathcal{H} \cup \mathcal{H}')(n) = \mathcal{H}(n) \cup \mathcal{H}'(n)$

principal. The second premise ensures that principals that speak for p's confidentiality also speak for q's confidentiality[6]. The third premise ensures that the query's context is sufficiently trusted to influence this authorization decision. Rule R-LIFTPC handles judgments regarding the query label as a special case. Rules R-CONJR, R-DISJL, R-WEAKEN, and R-TRANS are similar to their non-robust counterparts but possess robust premises. Rule R-TRANS adds a query label restriction to TRANS to ensure that the query's context speaks for r. Likewise, R-FWD adds the restriction that remote principals must speak for the principal that the judgment concerns.

The need for both robust and non-robust inference rules may not be immediately apparent. FLAM constrains the flow of information during authorization by selectively prohibiting derivations that would result in information leakage. However, reasoning exclusively with robust judgments is too restrictive since it would eliminate many valid trust configurations and prevent many access control use-cases. For access control decisions (made via non-robust queries), the robust judgments in FWD and WEAKEN ensure the integrity and confidentiality of authorization decisions. For information flow control decisions (made via robust judgments), the non-robust judgments in R-STATIC, R-LIFT, and R-LIFTPC provide a bootstrapping mechanism for trust relationships that preserves information security.

5 Robust authorization

To demonstrate that the inference rules presented in the previous section prevent the various attacks described in Section 2, we show that the rules ensure a novel security condition that we call *robust authorization*. This security condition characterizes how both delegations and revocations may affect authorization decisions in a particular information-flow context.

Theorem 1 (Robust authorization)**.** *If $\mathcal{H}; c; pc; \ell \vdash p \succcurlyeq q$, let $D \subseteq \mathcal{H}$ be the delegations used in the derivation. For each $(p' \succcurlyeq q', \ell') \in D(n)$, define $n_0 \ldots n_k$ as the sequence of nodes in the derivation between n and c, where $n_0 = n$ and $n_k = c$, and let $N = \bigvee_{i<k} n_i$. Then the following statements hold:*

$$\mathcal{H}; c; pc; \ell \Vdash \ell' \vee N \sqsubseteq \ell \tag{1}$$

$$\mathcal{H}; c; pc; \ell \Vdash N \succcurlyeq pc^{\rightarrow} \wedge \ell^{\leftarrow} \tag{2}$$

$$k > 0 \Rightarrow \mathcal{H}; c; pc; \ell \Vdash c \succcurlyeq (\ell' \vee N)^{\rightarrow} \tag{3}$$

The guarantees robust authorization bestows on authorization queries are quite strong. Remote principals cannot exceed their authority to influence the derivation, despite having the power to create arbitrary delegations and participate in the derivation itself. In particular, the authorization mechanism preserves the end-to-end security of each delegation's information flow policy ℓ' (1) while preserving the confidentiality $pc^{\rightarrow}$ of the query and the integrity $\ell^{\leftarrow}$ of the result (2), and without leaking confidential information to c (3). Conclusion (3) only applies to distributed derivations (where $k > 0$) since we permit a node to use a local delegation without requiring proof that it acts for the delegation label.

FLAM derivations therefore never require unsafe communication: every remote node that participates in a derivation must robustly act for the confidentiality $pc^{\rightarrow}$ of the query and integrity $\ell^{\leftarrow}$ of the result. Results are received by c only if c is permitted to learn (implicitly) that c acts for $(\ell' \vee N)^{\rightarrow}$. Because FLAM makes no assumptions about the relationship between n and ℓ', the disjunction N limits the authority of ℓ' to be no greater than the nodes in the derivation, ensuring that malicious delegations do not influence the derivation

[6]The analogous premise for integrity is redundant since acting and speaking for integrity are equivalent: $\vdash p \succcurlyeq \nabla(q^{\leftarrow}) \iff \vdash p \succcurlyeq q^{\leftarrow}$

beyond the authority of these nodes. From the perspective of confidentiality, the disjunction also ignores information flows in which the claimed confidentiality of the delegation label exceeds the confidentiality authority of nodes providing the delegation; ignoring such flows makes sense because confidentiality is enforced by the providers, not by the recipient c.

Robust authorization is a proof-theoretic property since it defines security in terms of the relationship between FLAM judgments and delegation labels. However, it bears some resemblance to semantic security properties like noninterference. Adding or removing delegations with more confidentiality or less integrity than ℓ cannot affect the output of queries bounded by ℓ. However, since the judgments derivable in a particular context *define* which flows are interfering and which are not, there is some subtlety in the statement that certain delegations cannot affect these derivations. For example, the delegation $(\texttt{Bob} \succcurlyeq \texttt{Acme}, \texttt{Bob}^{\leftarrow})$ should be cause for concern: it asserts that `Acme` delegates to `Bob`, but with the integrity of `Bob`. Thus the delegation should not be sufficient to prove that $\mathcal{H}; c; \mathit{pc}; \texttt{Acme}^{\leftarrow} \vdash \texttt{Acme}^{\rightarrow} \sqsubseteq \texttt{Bob}^{\rightarrow}$. Theorem 1 states that such delegations do not affect *any* judgments with the bound $\mathit{pc}; \texttt{Acme}^{\leftarrow}$. In this paper, we do not make any formal connections between robust authorization and noninterference, but characterizing semantic guarantees of FLAM is an interesting future research direction.

FLAM ensures robust judgments cannot be leveraged to perform poaching attacks or other non-robust policy downgrades. The following lemma states that if a query holds with robust authority, then the query label speaks for any principal whose dynamic delegations are used in the derivation.

Lemma 1 (Principal factorization)**.** *If $\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq q$, then there exist principals q_s and q_d where $q \equiv_{\succcurlyeq} q_s \wedge q_d$ such that $\vdash p \succcurlyeq q_s$, $\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq q_d$, and*

$$\mathcal{H}; c; \mathit{pc}; \ell \Vdash \mathit{pc} \succcurlyeq \nabla(q_d)$$

In other words, queries with untrusted query labels can only derive robust judgments that hold statically, preserving each principal's control over the revocability of its information flow policies.

The fact that we can always split robust acts-for judgments into static and dynamic components means that we can derive a more traditional transitivity rule for robust judgments:

$$[\text{R-TRANS*}] \qquad \frac{\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq q \quad \mathcal{H}; c; \mathit{pc}; \ell \Vdash q \succcurlyeq r}{\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq r}$$

The main insight regarding the admissibility of R-TRANS* involves principal factorization. By Lemma 1, for any robust judgment $\mathcal{H}; c; \mathit{pc}; \ell \Vdash q \succcurlyeq r$, we can factor r into $r_s \wedge r_d$ such that $\mathcal{H}; c; \mathit{pc}; \ell \Vdash \mathit{pc} \succcurlyeq \nabla(r_d)$. Therefore, any judgment $\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq q$ in the same context can be used to derive $\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq r_d$ by R-TRANS. This relationship, combined with an additional result regarding static judgments, gives us the above rule.

Theorem 1 and Lemma 1 prove that attackers cannot use delegation and revocation to interfere with authorization queries, eliminating the delegation loophole (Section 2.1) and poaching attacks (Section 2.2). New delegations cannot cause unsafe communication to occur or cause existing delegations to be disclosed (Section 2.3) unless the new delegations are sufficiently trusted. Furthermore, this result serves as a useful guide to developers of DIFC systems and languages: supporting delegation and revocation while enforcing information flow policies requires *all* relabeling of policies to be robust—otherwise, changes in the trust configuration could be exploited to create new flows.

We formalized FLAM principals and our inference rules for deriving flow-limited judgments in Coq, and used this formalization to prove Theorem 1 and Lemma 1. We make one primary assumption, that

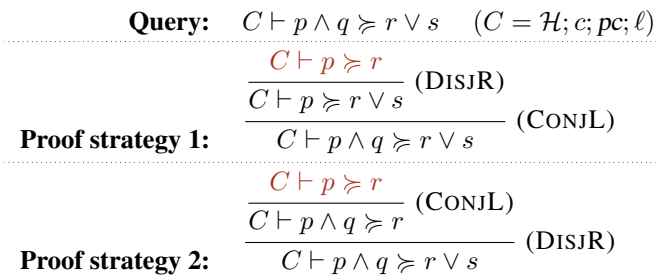


Figure 7: Redundant work in the basic search algorithm. If the query is not provable, an exhaustive proof search must be made before a negative result can be returned. Here, both CONJL and DISJR apply, so the search will try both proof strategies shown. Without caching, redundant proof searches would be made for the two identical premises shown in red.

principals that statically act for each other are equivalent. We believe this assumption can be avoided with some refactoring, which we leave as future work.

6 FLAM prototype

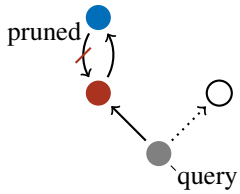
Secure authorization has been a relatively active area of research for over a decade [8, 9, 11–14, 21–23], so it might seem that the strong formal security guarantees offered by FLAM would be difficult to achieve in practice.

We have demonstrated that FLAM can be used to provide robust authorization in realistic authorization mechanisms by developing a prototype implementation and using it to implement ARBAC97 [15], an expressive role-based access control model. Our version of ARBAC97 uses owned principals to represent roles and extends the strong security guarantees of FLAM to role-based access control; for example, untrusted users cannot use authorization queries to infer the secret membership of roles. Our prototype currently only uses rules R-LIFT and R-LIFTPC for reasoning about robust judgments, but these were sufficient for our purposes.

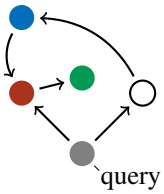
6.1 Efficient flow-limited query processing

Our FLAM prototype answers acts-for queries through a proof search; the relationship being queried is said to hold exactly when a proof of the relationship can be found. For simplicity, we assume that the trust configuration does not change during the proof search; in practice, query isolation can be provided by existing mechanisms for distributed transactions (e.g., [18]). The basic proof-search algorithm is a simple depth-first search with cycle detection. It returns two types of results: `PROVED` (which comes with a proof) and `FAILED`.

This algorithm alone performs poorly, however, owing to much duplicated work. Queries with `FAILED` results are particularly expensive, since they require a full exhaustive proof search. For example, in Figure 7, if the query $C \vdash p \wedge q \succcurlyeq r \vee s$ is unprovable, the algorithm must explore all possible proof strategies, including using CONJL and DISJR, as shown. Both of these strategies have the unprovable subquery $C \vdash p \succcurlyeq r$, shown in red. Without caching, redundant proof searches would be made for these identical



(a) A pruned proof strategy



(b) A successful proof strategy

Figure 8: Proof diagrams showing two strategies for proving a query. Nodes represent premises. Edges represent proof dependencies; unexplored edges are dotted. In strategy (a), the proof search for the blue node is pruned because its proof depends on the red node, which would introduce a cycle in the proof diagram. Strategy (b) results in a successful proof: the proof forms a DAG, wherein all leaf nodes are axioms.

subqueries. Furthermore, caching only positive results would not significantly improve the performance of unprovable queries.

Naively caching intermediate negative results can lead to incompleteness due to searches that are *pruned* to avoid infinite recursion and circular reasoning. Figure 8 illustrates this using proof diagrams. Nodes represent premises to be proved, and edges represent their dependencies. Unexplored edges are dotted. In the first proof strategy (Figure 8a), the proof of the blue node is pruned to avoid circular reasoning with the red node. While it would be sound to cache a `FAILED` result for the blue node, doing so would be incomplete. When the proof search later attempts the second proof strategy (Figure 8b), it finds a successful proof for the red node via the green node. With a cached `FAILED` result for the blue node, the proof of the white node would simply use the cached result, failing to notice that because the circularity with the red node has been resolved, the blue node can now be proved.

To prevent this incompleteness, our implementation of FLAM uses an intermediate caching strategy for pruned results. Instead of caching `FAILED` for pruned subqueries, we introduce an additional result type, `PRUNED`. When a cycle is detected during proof search, the current subproof is abandoned and the subquery is added to a cache of pruned queries. Each `PRUNED` cache entry contains a *progress condition*, a boolean formula that expresses the conditions under which further progress can be made on the proof of the subquery. In Figure 8, the first proof strategy would result in a `PRUNED` cache entry for the blue subquery, with the progress condition $Q =$ ●, indicating that further progress can be made on the proof of the blue node exactly when the red node can be proved. Another progress condition might have the form $Q_1 \vee (Q_2 \wedge Q_3)$, meaning that progress can be made if Q_1 is proved or if both Q_2 and Q_3 are proved.

This cache is used by the proof search to improve performance when resolving shared subqueries. The cache has three components: an *acts-for cache* for proofs of `PROVED` subqueries, a *failed cache* for `FAILED` subqueries, and a *pruned-search cache* for `PRUNED` subqueries and their progress conditions. Figure 9 gives the algorithm for updating the cache with a new result for a subquery $query$. At the core of this algorithm is the rewriting of progress conditions in the pruned-search cache. If the new result is `PROVED`, the progress conditions are rewritten to substitute instances of $query$ with `True` (line 7), to indicate that the $query$ condition is satisfied. If this satisfies the progress condition of a pruned search q, then q should be provable, and is removed from the cache (lines 8–9); a `PROVED` entry is not added for q yet because we do not yet have a proof. If the new result is `PRUNED`, then instances of $query$ are substituted with $query$'s progress condition (line 15). Finally, if the new result is `FAILED`, then instances of $query$ are substituted with `False` (line 21), to indicate that the $query$ condition is not satisfiable. If the progress condition of a pruned search q becomes unsatisfiable, then q is also unprovable, and the cache is updated with a `FAILED` result for q (lines 27–28).

Given a query, for each applicable FLAM inference rule, the algorithm searches for a proof of each

```
1: function UPDATE(cache, query, type, data)
2:     (proved, pruned, failed) ← cache
3:     if type = PROVED then
4:         proved ← proved[query ↦ data]
5:         remove query from pruned
6:         for [q ↦ Q] in pruned do
7:             Q′ ← Q{query/True}
8:             if Q′ ⊨ True then
9:                 remove q from pruned
10:            else
11:                pruned ← pruned[q ↦ Q′]
12:    else if type = PRUNED then
13:        pruned ← pruned[query ↦ data]
14:        for [q ↦ Q] in pruned do
15:            pruned ← pruned[q ↦ Q{query/data}]
16:    else if type = FAILED then
17:        add q to failed
18:        remove q from pruned
19:        new ← ∅
20:        for [q ↦ Q] in pruned do
21:            Q′ ← Q{query/False}
22:            if Q′ ⊨ False then
23:                add q to new
24:            else
25:                pruned ← pruned[q ↦ Q′]
26:        cache ← (proved, pruned, failed)
27:        for q in new do
28:            cache ← UPDATE(cache, q, FAILED, ⊥)
29:        return cache
30:    return (proved, pruned, failed)
```

Figure 9: Algorithm for managing entries of the proof search cache. For $type$ equal to PROVED or PRUNED, $data$ is either a proof of $query$ or a progress condition, respectively.

premise. If a proof is found for all premises, then the search is successful, and the proof is returned. If any of the premises' proof searches were pruned, then the query may or may not be provable, so the query is added to the pruned cache with the conjunction of the progress conditions of the pruned searches. Finally, if any premise's proof search fails, or if the conjunction of the progress conditions is unsatisfiable, then the query is unprovable via the chosen rule. If no other FLAM rules apply, then the query is false. Appendix D gives the complete search algorithm.

6.2 Example: ARBAC97 Access Control

To demonstrate the expressiveness of FLAM and the functionality of our implementation, we have adapted the ARBAC97 role-based access control model for role management [15] using our FLAM implementation. The implementation required only 242 lines of code, showing that FLAM is already quite expressive. Using FLAM means our implementation of ARBAC97 also enjoys stronger security properties; in particular, robust

$assignUser(a, u, r, \mathsf{pc}, \ell)\{$
  if $\exists(ar, cr, mn, mx) \in can\_assign$
  such that
    $\mathcal{H}; c; \mathsf{pc}; \ell \Vdash a \succcurlyeq ar$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash u \succcurlyeq cr$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash r \succcurlyeq mn$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash mx \succcurlyeq r$
  then
    let $\ell' = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge r)^{\leftarrow}$
    $\mathcal{H} := \mathcal{H} \cup [c \mapsto (u \succcurlyeq r, \ell')]$
$\}$

(a) Authorize a's assignment of user u to role r. If the FLAM judgments hold, a delegation $u \succcurlyeq r$ is created with the integrity of ar and r.

$revokeUser(a, u, r, \mathsf{pc}, \ell)\{$
  if $\exists(ar, mn, mx, \mathsf{pc}, \ell) \in can\_revoke$
  such that
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash a \succcurlyeq ar$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash r \succcurlyeq mn$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash mx \succcurlyeq r$
  then
    let $\ell' = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge r)^{\leftarrow}$
    $\mathcal{H} := \bigcup_{c \in \mathrm{dom}(\mathcal{H})} [c \mapsto \mathsf{rev}(\mathcal{H}, c, u \succcurlyeq r, \ell')]$
$\}$

(b) Authorize a's revocation of u's membership in role r. If the FLAM judgments hold, all delegations $(u \succcurlyeq r, \ell'')$ where $\ell' \sqsubseteq \ell''$ are revoked.

$$\mathsf{rev}(\mathcal{H}, c, p \succcurlyeq q, \ell) \triangleq \mathcal{H}(c) - \{(p \succcurlyeq q, \ell') \in \mathcal{H}(c) \mid \mathcal{H}; c; \mathsf{pc}; \ell \Vdash \ell \sqsubseteq \ell'\}$$

(c) Revocation operation. Returns the delegation set for host c with all delegations $(p \succcurlyeq q, \ell')$ where $\ell \sqsubseteq \ell'$ removed.

Figure 10: User–role assignment. The FLAM judgments ensure a is a member of the administrative role ar, that u meets criteria cr (in Figure 10a), and that r is in the range $[mn, mx]$. Each judgment requires the integrity of ar to ensure only administrators influence role management.

authorization means that attackers can neither influence the membership of roles nor learn anything about confidential role assignments.

ARBAC97 controls trust management operations using three separate relations: user–role assignment (UA), for assigning users to roles; permission–role assignment (PA), for specifying the permissions granted to roles; and role–role assignment (RH), for defining role hierarchies. ARBAC authorizes a user's modifications to these relations by ensuring an administrator is a member of the appropriate administrative role and that modifications meet specified conditions.

The key difficulty in representing ARBAC's role-management authorization policies is in the separation between management authority and role membership. FLAM simplifies the ARBAC model since administrative roles, roles, users, and permissions may all be represented as principals. This allows the unification of the three relations UA, PA, and RH into a single trust configuration $\mathcal{H}$. Our version of ARBAC97, adapted from the formalization presented in [36], leverages FLAM's information flow tracking and expressive principal algebra to preserve the separation of management authority and role membership in $\mathcal{H}$.

In ARBAC, the authorization criteria for making modifications to the trust configuration is defined by additional relations[7]. The relations *can_assign* and *can_revoke* encode policies for user–role assignment.

[7]For simplicity, we treat these relations as public and trusted, and thus do not track information flows on them.

Entries of *can_assign* are tuples of principals (ar, c, mn, mx), where ar represents an *administrative role*, cr represents some *minimal criteria*[8] that users must meet to be assigned the role, and $[mn, mx]$ represents a range that bounds the role assignments ar is permitted to make. Entries of can_revoke are tuples of principals (ar, mn, mx) which are similar to those of *can_assign*, but have no minimal criteria.

FLAM strengthens the guarantees of ARBAC97 by tracking information flow on modifications to trust configuration and ensuring robust authorization. Figures 10a and 10b illustrate our encoding of user–role assignment authorization. Each method includes a parameter *pc* that represents the information flow context of the caller, and a label ℓ for specifying the confidentiality and integrity of the role assignment. In Figure 10a, the assignment of user u to role r by administrator a is authorized if there is an entry in the can_assign relation such that the subsequent FLAM judgments hold robustly with the integrity of ar. The first judgment ensures a is a member of the ar role. The second judgment ensures that the user acts for a principal representing some minimal criteria. Finally, the third and fourth judgments ensure r is within the range $[mn, mx]$.

When the relevant FLAM judgments hold, delegation or revocation is performed with the integrity of both ar and r. This indicates that the above methods *endorse* the delegation or revocation. As shown below, we use these high-integrity delegations to keep role membership separate from role management.

ARBAC is a centralized access control model: there is a single hierarchy of administrative roles. In addition to providing stronger security guarantees, our FLAM adaptation extends ARBAC to decentralized settings. Administrative domains may differ on the roles assigned to a particular user. Let AR be a set of administrative roles. We use a principal ad to represent an *administrative domain*, defined as the disjunction of a set of administrative roles:

$$ad \triangleq \bigvee_{ar \in AR} ar$$

Then for a particular administrative domain ad, we can determine if user u is a member of role r with the following FLAM query:

$$\mathcal{H}; c; \mathit{pc}; \ell \wedge ad^{\leftarrow} \Vdash u \succcurlyeq r$$

By requiring the integrity of ad, we ensure that only delegations created by some administrative role are considered. Since the judgment is robust, the delegation must also have the integrity of r, meaning that ar can only influence delegations via *assignUser* which constrains the roles ar may assign and the users it may assign them to.

The remaining methods for permission–role management and role–role management share many similarities with the above methods for user–role management. The definitions of these methods are found in Appendix C.

Our implementation suggests a general approach for extending robust authorization to traditional access control models. Translating the *authority* implied by the ARBAC97 roles to FLAM trust relationships allow FLAM queries to securely implement ARBAC authorization requests without creating authorization side channels. Coupling this translation with specialized role management code yields a more secure access control system. This exercise demonstrates the expressiveness of FLAM policies as well as the effectiveness of the implemented algorithm; we expect other access control systems could be enhanced in a similar way.

[8] In [15], criteria are more general, allowing cr to specify both roles that a user *must* have, as well as roles a user *must not* have. By separating positive and negative criteria we can represent the general case in FLAM, but for simplicity of exposition we omit negative criteria.

7 Robust authorization with Fλ

FLAM provides a secure model for authorization with information flow control. To explore how FLAM can be used to build secure systems, we have defined a DIFC programming language based on FLAM that supports robust trust management and authorization. Fλ ("FLAMbda") is a security typed language whose policies are FLAM principals. Fλ programs may delegate or revoke trust, and may issue authorization queries to determine whether a relationship holds. To ensure a consistent view of the trust configuration, all authorization queries occur in transactions, and updates to the trust configuration are applied atomically upon transaction commit. The syntax (Figure 11) is based on λ_{DSec} [37], a DIFC language with dynamic labels and dependent security types. We have redefined the syntax and semantics of λ_{DSec} labels to support a unified representation of policies as principals with attenuated authority, and added flow-limited authorization, policy downgrading, dynamic delegation and revocation, and distributed state.

Like other statically-typed DIFC languages, Fλ aims to enforce information-flow security via type-checking. In addition to type safety, Fλ's subtyping rule ensures that a supertype's policy is at least as restrictive as the subtype's policy. Unlike other DIFC languages, Fλ also ensures that all relabelings are robust, ensuring by Theorem 1 that attackers cannot leak information by exploiting delegation and revocation.

Although Theorem 1 provides a strong guide for language designers, a choice remains in how the static type system should interact with the dynamic trust configuration. In Fλ, program use *explicit downgrading* to relabel policies: dynamic trust relationships may only be used to prove flows annotated with the `relabel` keyword. All other flows must be proven secure for any trust configuration. Although some languages require explicit declassification or endorsement for some relabelings, existing DIFC languages (e.g., [12, 14, 18, 24, 35, 38]) do not regard relabeling as a downgrading operation. In these languages, any relationship that holds in a particular context may be used to prove that a flow is valid.

In light of the attacks discussed in Section 2, we find this implicit use of dynamic trust information unsatisfying. First, it violates the principle of least privilege [39] since programs may wield their full authority whether or not they require it. Implicit downgrading could lead to unintentional errors when a program *can* declassify information (i.e., in a high-integrity context), but the developer does not intend to. Consequently, high-integrity code would become more dangerous and would require more careful auditing.

Another weakness of implicit downgrading relates to revocation. As with unintentional declassification or endorsement, allowing policies to be relabeled implicitly makes it more difficult to reason about what a principal retains access to after a revocation. Here also, additional auditing may be required to ensure information was not relabeled to other policies prior to a revocation.

The Fλ type system makes explicit any relabelings that are based on dynamic information using the `relabel` keyword. This approach ensures the intentions of the developer are clear. Implicit relabelings are permitted, but only if they can be proven safe in an empty trust configuration. Therefore, these relabelings are safe in any trust configuration.

7.1 Operational semantics of Fλ

The evaluation rules for the Fλ authorization operations are presented in Figure 12. The complete Fλ operational semantics is presented in Appendix A. The state $\sigma = \langle \sigma_{\mathcal{H}}, \sigma_M \rangle$ of a Fλ program includes the trust configuration $\sigma_{\mathcal{H}}$ and σ_M, a map from typed locations $m^{\tau@a}$ to values.

The premises of E-IfTrue and E-IfFalse are FLAM authorization queries parameterized by the dynamic principals in the program. Depending on the result of the query, execution proceeds to e_1 or e_2. In an implementation of Fλ, these rules would define the interface with a FLAM library such as our prototype in section 6. E-Trust creates a new delegation at host r with the specified information policy ℓ. E-Revoke

$$
\begin{array}{rcl}
D & ::= & D \mid (p \succcurlyeq p, \ell)\,,\ D \mid \epsilon \\
\beta & ::= & \texttt{int} \mid \texttt{prin} \mid \texttt{unit} \mid \tau @ a\ \texttt{ref} \\
& \mid & (x{:}\tau) \xrightarrow{D\,;\,pc} \tau \mid (x{:}\tau) * \tau \\
\tau & ::= & \beta_\ell \\
v & ::= & n \mid p\ \text{closed} \mid () \mid m^{\tau @ a} \\
& \mid & \lambda(x{:}\tau)[D\,;pc].\, e \mid (x{=}v,\ v{:}\tau) \\
e & ::= & x \mid v \mid e\ e \mid\ !e \mid e := e \\
& \mid & \texttt{ref}^{\tau @ a}\ e \mid \texttt{let}\ (x,y){=}e\ \texttt{in}\ e \\
& \mid & \texttt{if}\ p \succcurlyeq p\ \texttt{with}\ \ell\ \texttt{at}\ \ell\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & p\ \texttt{trust}\ p\ \texttt{with}\ \ell\ \texttt{at}\ \ell \mid \texttt{relabel}\ e\ \ell \\
& \mid & p\ \texttt{revoke}\ p\ \texttt{with}\ \ell\ \texttt{at}\ \ell
\end{array}
$$

For $x \in \mathcal{V}$ (variables), $n \in \mathbb{Z}$, $m \in \mathcal{M}$ (locations), and $p, \ell, a, pc \in \mathcal{P}$ with primitive principals $k \in \mathcal{N} \cup \mathcal{V}$.

Figure 11: Syntax of Fλ.

performs a revocation visible at policy ℓ. Any delegations with labels at least as restrictive as ℓ are removed (robustly), but less restrictive delegations are unaffected, ensuring the revocation doesn't leak information or influence trusted operations. Relabeling of information, in rule E-RELABEL, is a purely static operation with no run-time effect.

Memory locations $m^{\tau @ a}$ may be stored at remote hosts. The access policy [20] a represents a bound on the confidentiality authority of that host's principal. Any principal r such that $\vdash r \succcurlyeq a$ may be a host for memory or delegations with access policy a. Because this judgment holds with any trust configuration and delegation set, revocations can never cause the host of a delegation or memory location to become invalid. Note that via R-STATIC, such judgments are also robust.

Accesses to resources stored at such a host are observable by that host's principal. Evaluation steps that generate a potentially observable event α are marked by $\xrightarrow{\alpha}$. The function $obs(\alpha)$, defined below, denotes the principal with the least authority capable of observing an event. For $\alpha = \cdot$ (no event), no event is observable so $obs(\cdot) = \top^{\rightarrow}$.

Definition 8 (Event observability)**.**

$$
obs(\alpha) = \begin{cases} pc^{\rightarrow} & \alpha = pc(\cdots) \\ \top^{\rightarrow} & \alpha = \cdot \end{cases}
$$

Though Fλ is design to model distributed computation, in this paper we are concerned with semantic security guarantees for sequential programs. Thus, for simplicity, we do not define any mechanisms for concurrency control.

7.2 Fλ type system

Figure 13 shows the typing rules for the trust-related operations of Fλ. The typing judgment for Fλ has the general form $\Gamma\,;D\,;\ pc \vdash e : \tau$, where Γ is a type environment mapping variables to types, D is the static view of the current delegation set, and *pc* is the information flow policy on the program counter.

Authorization queries are integrated into the typing context via rule T-IF. The second premise of T-IF ensures the FLAM query label is at least as restrictive as the *pc* label and the labels on the query parameters. Validated queries are appended to the static trust configuration, allowing them to be reused statically without communication. These entries are used to perform relabelings via T-RELABEL, which ensures that all such relabelings are robust. The complete set of typing rules are presented in Appendix B.

$\boxed{\langle e,\, \sigma\rangle \xrightarrow{\alpha} \langle e',\, \sigma'\rangle}$

[E-IFTRUE]
$$\frac{\sigma_{\mathcal{H}}; pc; \ell \vdash p \succcurlyeq q \qquad \sigma' = \langle \sigma_{\mathcal{H}}; \sigma_M \rangle}{\langle \texttt{if}\ p{\succcurlyeq}q\ \texttt{with}\ \ell\ \texttt{at}\ pc\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, \sigma\rangle \xrightarrow{pc(p,q,\ell)} \langle e_1,\, \sigma'\rangle}$$

[E-IFFALSE]
$$\frac{\sigma_{\mathcal{H}}; \ell; pc^{\rightarrow} \not\vdash p \succcurlyeq q}{\langle \texttt{if}\ p{\succcurlyeq}q\ \texttt{with}\ \ell\ \texttt{at}\ pc\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, \sigma\rangle \xrightarrow{pc(p,q,\ell)} \langle e_2,\, \sigma\rangle}$$

[E-TRUST]
$$\frac{\mathcal{H}' = \sigma_{\mathcal{H}}[r \mapsto \sigma_{\mathcal{H}}(r), (p{\succcurlyeq}q, \ell)]}{\langle p\ \texttt{trust}\ q\ \texttt{with}\ \ell\ \texttt{at}\ r,\, \sigma\rangle \xrightarrow{r(+,p,q,\ell)} \langle (),\, \langle \mathcal{H}'; \sigma_M\rangle\rangle}$$

[E-REVOKE]
$$\frac{\mathcal{H}' = [r \mapsto \mathsf{rev}(\sigma_{\mathcal{H}}, r, p{\succcurlyeq}q, \ell) \mid \sigma_{\mathcal{H}}; pc; \ell \Vdash r \succcurlyeq pc^{\rightarrow}]}{\langle p\ \texttt{revoke}\ q\ \texttt{with}\ \ell\ \texttt{at}\ pc,\, \sigma\rangle \xrightarrow{pc(-,p,q,\ell)} \langle (),\, \langle \mathcal{H}'; \sigma_M\rangle\rangle}$$

[E-RELABEL]
$$\langle \texttt{relabel}\ v\ \ell,\, \sigma\rangle \xrightarrow{\cdot} \langle v,\, \sigma\rangle$$

$$\mathsf{rev}(\mathcal{H}, r, p{\succcurlyeq}q, \ell) \triangleq \mathcal{H}(r) - \{(p{\succcurlyeq}q, \ell') \in \mathcal{H}(r) \mid \mathcal{H}; c; pc; \ell \Vdash \ell \sqsubseteq \ell'\}$$

Figure 12: Semantic rules for Fλ authorization operations.

The typing rule and updates to the trust configuration are parameterized by . In rule T-IF, these policies limit the direct delegations used to make an authorization judgment. Rules T-TRUST and T-REVOKE ensure updates to the trust configuration respect these policies.

Notably absent from the type system of Fλ is an explicit notion of downgrading authority. Many DIFC systems (e.g., [3–7, 18, 35]) constrain where policy downgrades may occur in the system by designating processes or classes with downgrading authority. As in Rx [12] and RTI [14], Fλ declassifications and endorsements are expressed by updating trust relationships. The integrity policies on delegations and revocations ensure that decisions to relabel are only influenced by trusted principals. Since delegation may change the interpretation of confidentiality and integrity policies simultaneously, the delineation between downgrading a confidentiality policy and downgrading an integrity policy is less natural. Therefore, Fλ does not explicitly distinguish between declassification and endorsement. Instead, policy downgrades are performed by delegating and relabeling expressions.

We assume that the labels of memory locations and delegations are enforceable by the hosts of each resource regardless of any revocations that may occur. This ensures revocations cannot cause the location of a resource to become insecure. We formalize these assumptions with the following definition.

Definition 9 (Well-formed program elements)**.** Memory M is *well-formed* if $\forall m^{\beta_\ell @ a} \in \mathit{dom}(M),\ \Vdash \beta \sqsubseteq a$ and $\vdash M(m^{\beta_\ell @ a}) : \beta_\ell$. A configuration $\langle e,\, \sigma\rangle$ is well-formed if $FV(e) = \emptyset$, $loc(e) \subseteq \mathit{dom}(\sigma_M)$, and σ_M is well-formed.

7.3 Example: implementing screening policies

Information screening policies, historically called a "Chinese wall," are used to prevent conflicts of interest. These policies are familiar in access control settings, but rely on properties previous DIFC systems cannot

$\boxed{\Gamma\,;D\,;\,pc \vdash e : \tau}$

[T-TRUST]
$$\frac{\Gamma\,;D\,;\,pc \vdash \ell_i : \mathtt{prin}_{\ell_i'}\ \ i \in \{1,2,3,4\} \qquad \Vdash \bigsqcup_i \ell_i' \sqcup pc \sqsubseteq \ell_3 \qquad \Vdash \ell_4 \succcurlyeq \ell_3}{\Gamma\,;D\,;\,pc \vdash \ell_1\ \mathtt{trust}\ \ell_2\ \mathtt{with}\ \ell_3\ \mathtt{at}\ \ell_4 : \mathtt{unit}_\bot}$$

[T-REVOKE]
$$\frac{\Gamma\,;D\,;\,pc \vdash \ell_i : \mathtt{prin}_{\ell_i'}\ \ i \in \{1,2,3,4\} \qquad \Vdash \bigsqcup_i \ell' \sqcup pc \sqsubseteq \ell_3 \qquad \Vdash \ell_4 \succcurlyeq \ell_3}{\Gamma\,;D\,;\,pc \vdash \ell_1\ \mathtt{revoke}\ \ell_2\ \mathtt{with}\ \ell_3\ \mathtt{at}\ \ell_4 : \mathtt{unit}_\bot}$$

[T-RELABEL]
$$\frac{\Gamma\,;D\,;\,pc \vdash e : \beta_\ell \qquad [c \mapsto D]; pc; \ell' \Vdash \ell \sqsubseteq \ell'}{\Gamma\,;D\,;\,pc \vdash \mathtt{relabel}\ e\ \ell' : \beta_{\ell'}}$$

[T-IF]
$$\frac{\begin{array}{c}\Gamma\,;D\,;\,pc \vdash \ell_i : \mathtt{prin}_{\ell_i'}\ \ i \in \{1,2,3,4\} \\ \Vdash \bigsqcup_i \ell_i' \sqcup pc \sqsubseteq \ell_4 \qquad \Gamma\,;D, (\ell_1 \succcurlyeq \ell_2, \ell_3)\,;\,\ell_3 \sqcup \ell_4 \vdash e_1 : \tau \qquad \Gamma\,;D\,;\,\ell_3 \sqcup \ell_4 \vdash e_2 : \tau\end{array}}{\Gamma\,;D\,;\,pc \vdash \begin{array}{l}\mathtt{if}\ \ell_1 \succcurlyeq \ell_2\ \mathtt{with}\ \ell_3\ \mathtt{at}\ \ell_4 \\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\end{array} : \bigsqcup_i \ell_i' \sqcup \ell_3 \sqcup \tau}$$

Figure 13: Typing rules for Fλ authorization operations.

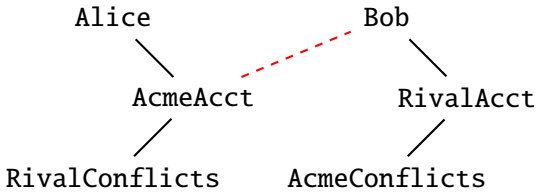


Figure 14: Information screening: `Bob` should not be allowed to act for `AcmeAcct` (dashed line) since it is a conflict of interest.

provide. Integrating such policies into DIFC systems is tempting since the inherent compositionality of information flow enable decentralized screening policies managed by mutually distrustful principals.

For example, a law firm, Loblaw LLC, has both Acme and Rival as clients. Loblaw uses screening policies to eliminate conflicts of interest by preventing lawyers from accessing both Acme and Rival data. Each screening policy is represented by two principals: an "account" principal representing the group with access to client's account data, and a "conflicts" principal, representing the client's competitors. Membership in these groups is represented by trust delegation: `Alice` is a member of the `AcmeAcct` group if `Alice` $\succcurlyeq$ `AcmeAcct`. Figure 14 shows the trust configuration of these screening policies. Alice has access to the Acme account, while Bob has access to the Rival account. Since Acme and Rival are competitors, the `RivalAcct` principal is a member of Rival's conflict group `AcmeConflicts`. Since Bob is a member of the `RivalAcct` group, the screening policy should prevent Bob from being added to `AcmeAcct`.

Figure 15 shows implementations of two core operations of the screening policy pattern. Each screening policy has an owner `o` who manages the screening policy. In our example, owner `o` would be Loblaw LLC. The owned principal `o : g` represents the group with access to the protected information, e.g., `AcmeAccts`. Principal `o : x` represents the conflict group, e.g. `AcmeConflicts`. The principal `o : p` is the ownership projection that represents the member to add. Additionally, the confidentiality of `o : p`'s membership in `o : g` or `o : x` is represented by $\mathtt{s}^{\rightarrow}$. It is easy to check that this Fλ program type-checks. Therefore, only principals with integrity `o` : $\mathtt{g}^{\leftarrow}$ may influence the membership of `o : g` or `o : x`, and only principals trusted

```
λ(o:prin_{s→∧o:g←}, p:prin_{s→∧o:g←}, g:prin_{s→∧o:g←},
  s:prin_{s→∧o:g←}, x:prin_{s→∧o:g←})
[s→ ∧ o : g←].
 if o:p ≽ o:x with s→ ∧ o : g← at s→ ∧ o : g← then ()
 else o:g trust o:p with s→ ∧ o : g← at s→ ∧ o : g←
```

(a) `addmember`: Add $o : p$ to group $o : g$ unless $o : p \succcurlyeq o : x$.

```
λ(o:prin_{s→∧o:g←}, p:prin_{s→∧o:g←}, g:prin_{s→∧o:g←},
  s:prin_{s→∧o:g←}, x:prin_{s→∧o:g←})
[s→ ∧ o : g←].
 if o:p ≽ o:g with s→ ∧ o : g← at s→ ∧ o : g← then ()
 else o:x trust o:p with s→ ∧ o : g← at s→ ∧ o : g←
```

(b) `addconflict`: Add $o : p$ to conflicts $o : x$ unless $o : p \succcurlyeq o : g$.

Figure 15: Primitives for the information screening policy pattern

with confidentiality $\mathsf{s}^{\rightarrow}$ can learn the members of these groups.

8 Related work

The connection between delegation and policy downgrades, here called the delegation loophole, is identified in [10] and further developed in [12]. These papers also discuss secret trust relationships, and thus have similar threat models to FLAM. We are not aware of previous work addressing poaching attacks.

Broberg et al. [40] identify classes of flows which specific information flow models may consider secure or insecure. Delegation loopholes are an example of a *time-transitive flow* in their terminology. FLAM considers these flows insecure since they permit attackers to influence how information is relabeled. FLAM also considers poaching attacks to be unsafe since attackers may obtain information not directly released to them, which undermines the effectiveness of revocation. These flows are not completely characterized by the classes presented in [40], but share some characteristics with the *direct-release* class of flows.

FLAM's bounded derivation rules place information flow constraints on which delegations may be used to derive judgments. This differs from previous approaches (e.g., Rx roles [12] and Flume capability groups [5]), which give a single information flow bound for all trust relationships of a principal. As recognized by Bandhakavi et al. [14], a single bound is too restrictive since it must also protect delegations made by other principals. So, when the bound of principal p is more restrictive than the bound of principal q, either q cannot delegate to p or p's bound must be downgraded (as in [12]), even though p might not trust q. RTI [14], like FLAM, overcomes these restrictions by tracking information flow at the level of delegations and ignoring relationships that exceed information flow bounds. However, since relabeling is not robust in RTI, it remains vulnerable to the delegation loophole and poaching attacks. FLAM's flows-to relation is more consistent with decentralized information flow control principles: the authorization of a flow depends only on those principals who speak for the policies in question.

Label algebras [41] abstract the structure and semantics of the security policies of several DIFC systems. It might appear that a clever encoding of FLAM contexts (i.e., $pc; \ell$) as label algebra *authorities* might serve to represent FLAM as a label algebra. However, such an encoding would be too abstract to represent conditions such as robust authority or even robust downgrading, so delegation loopholes and poaching attacks cannot be addressed within this framework. For instance, the noninterference lemma given in [41] for an

example language admits non-robust declassification, even without changes to the trust configuration.

Many models and mechanisms have been suggested for expressive, decentralized authorization and trust management [26, 42–49]. Few consider the information security of the authorization policies or the authorization process. For instance, Birgisson et al. [49] note that, under certain conditions, an attacker could use malicious credentials to probe for private information such as group membership. Such an attack is possible in many frameworks. DCC [50] has been used to model both information flow control and authorization logic [27], but not both simultaneously.

Several authorization systems use access control policies to protect sensitive credentials. In trust negotiation [8,22,23], principals iteratively exchange credentials protected by access control policies, withholding sensitive credentials until sufficient trust has been established. Minami and Kotz [11, 13] encrypt authorization proofs based on access control policies to protect the confidentiality and integrity of authorization results, though they ignore side-channels. Because access control policies are not compositional, they are insufficient for controlling the propagation of sensitive credentials: the rules for disclosure may vary arbitrarily between principals. FLAM unifies principals and information flow control policies, which are inherently compositional, and enforces end-to-end security of trust relationships.

Garg and Pfenning [51] present a constructive authorization logic that ensures assertions made by untrusted principals cannot influence the truth of statements made by other principals, similar to the way low-integrity delegations in FLAM cannot lead to unsafe relabelings.

Becker [52] discusses *probing attacks* that reveal secret portions of authorization policies to an attacker; Bryans et al. [53] compare noninterference and opacity as security conditions for confidential policies. FLAM ensures queries of the trust configuration satisfy robust authorization, so probing attacks cannot reveal confidential information. Opacity is a *possibilistic* notion of security, meaning that an authorization decision may depend on secret information, provided that the same result could derive from public information. Possibilistic security conditions are often inadequate in settings with attackers that have (or can acquire) additional knowledge, perhaps through additional queries.

Some type systems proposed for information flow control encode authorized policy downgrades directly in data types (e.g., [54, 55]) or with respect to privileges granted to code (e.g. [56]). This removes some of the need for an underlying authorization mechanism, permitting developers to model trust relations using the type system or structure of the program. Such type systems are in a sense too low-level to be directly vulnerable to delegation loopholes or poaching attacks, but the authorization mechanisms they encode may still be vulnerable. FLAM can provide guidance for a way to obtain robust authorization in these systems.

9 Conclusions

We have shown that in a decentralized, distributed setting, mechanisms for both DIFC and authorization currently exhibit security vulnerabilities. The core problem is that neither security mechanism tracks how information flows through the authorization process itself. Consequently, both mechanisms introduce side channels, and DIFC systems are subject to newly identified delegation loopholes and poaching attacks. When the trust configuration is dynamic and can be affected by partially trusted principals, additional controls are needed to make relabeling secure.

We introduced flow-limited authorization in FLAM as a simple, coherent, and powerful way to address a set of fundamental, interconnected security issues. FLAM unifies principals with information flow policies through a novel principal algebra. It supports integrated reasoning about both authorization and information flow control so that delegations are trusted only when appropriate and kept secret when necessary; further, authorization side channels are explicitly controlled. A key insight is that relabeling information flow poli-

cies is really a downgrading operation that can be made secure by preventing untrusted principals from influencing relabeling decisions.

We have formalized FLAM in Coq and proved strong results: FLAM provides *robust authorization*, a new security condition that bounds an attacker's influence on authorization decisions and eliminates side-channels, even when the attacker is able to modify the trust configuration and make arbitrary queries.

We have implemented the FLAM principal normalization algorithm and system of inference rules (with the exception of some robustness rules). Our prototype efficiently answers FLAM queries using a specialized caching protocol.

FLAM not only prevents security vulnerabilities, but also extends decentralized information flow control to trust management systems with expressive security models like role-based access control. We expect many common access control patterns have interesting new DIFC analogues when expressed using FLAM.

Acknowledgments

We thank Michael Clarkson, Fred Schneider, Ross Tate, and especially Aslan Askarov and Mike George for helpful discussions on a variety of topics spanning trust and information flow, authorization logic, and proof search algorithms. Abhishek Anand and Andrew Hirsch provided insight for formalizing FLAM in Coq. For their insightful comments on our submission, we thank Eleanor Birrell, Steve Chong, Andrew Hirsch, Chin Isradisaikul, Elisavet Kozyri, Tom Magrino, Laure Thompson, Bart van Delft, Danfeng Zhang, Yizhou Zhang, and our anonymous reviewers. This work was supported by an NDSEG Fellowship, by grant N00014-13-1-0089 from the Office of Naval Resesearch, by MURI grant FA9550-12-1-0400, and by a grant from the National Science Foundation (CCF-0964409). This paper does not necessarily reflect the views of any of these sponsors.

References

[1] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. on Security and Privacy*, Apr. 1982, pp. 11–20.

[2] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 4, pp. 410–442, Oct. 2000.

[3] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the Asbestos operating system," in *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, Oct. 2005.

[4] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proc. 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 263–278.

[5] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard OS abstractions," in *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[6] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: Practical fine-grained decentralized information flow control," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2009.

[7] W. Cheng, D. R. K. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov, "Abstractions for usable information flow control in Aeolus," in *Proc. 2012 USENIX Annual Technical Conference*, Jun. 2012.

[8] W. H. Winsborough, K. E. Seamons, and V. E. Jones, "Automated trust negotiation," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 1, Jan. 2000, pp. 88–102.

[9] W. H. Winsborough and N. Li, "Safety in automated trust negotiation," in *Proc. IEEE Symp. on Security and Privacy*, May 2004, pp. 147–160.

[10] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic, "Dynamic updating of information-flow policies," in *Proc. Foundations of Computer Security Workshop*, 2005.

[11] K. Minami and D. Kotz, "Secure context-sensitive authorization," *Journal of Pervasive and Mobile Computing*, vol. 1, no. 1, pp. 123–156, March 2005.

[12] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic, "Managing policy updates in security-typed languages," in *Proc. 19th IEEE Computer Security Foundations Workshop*, Jul. 2006, pp. 202–216.

[13] K. Minami and D. Kotz, "Scalability in a secure distributed proof system," in *Proc. 4th International Conference on Pervasive Computing*, ser. Lecture Notes in Computer Science, vol. 3968. Dublin, Ireland: Springer-Verlag, May 2006, pp. 220–237.

[14] S. Bandhakavi, W. Winsborough, and M. Winslett, "A trust management approach for flexible policy management in security-typed languages," in *Computer Security Foundations Symposium, 2008*, 2008, pp. 33–47.

[15] R. Sandhu, V. Bhamidipati, and Q. Munawer, "The ARBAC97 model for role-based administration of roles," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 1, pp. 105–135, 1999.

[16] H. Chen and S. Chong, "Owned policies for information security," in *Proc. 17th IEEE Computer Security Foundations Workshop*, Jun. 2004.

[17] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing confidentiality and integrity in web applications," in *Proc. 16th USENIX Security Symp.*, Aug. 2007.

[18] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, "Fabric: A platform for secure distributed computation and storage," in *Proc. 22nd ACM Symp. on Operating System Principles (SOSP)*, Oct. 2009, pp. 321–334.

[19] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM Trans. on Computer Systems*, vol. 20, no. 3, pp. 283–328, Aug. 2002.

[20] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers, "Sharing mobile code securely with information flow control," in *Proc. IEEE Symp. on Security and Privacy*, May 2012, pp. 191–205.

[21] W. H. Winsborough and N. Li, "Towards practical automated trust negotiation," in *Proc. 3rd Policies for Distributed Systems and Networks*. IEEE, 2002, pp. 92–103.

[22] M. Winslett, C. C. Zhang, and P. A. Bonatti, "Peeraccess: A logic for distributed authorization," in *Proc. 19th ACM Conf. on Computer and Communications Security (CCS)*. ACM, 2005, pp. 168–179.

[23] C. C. Zhang and M. Winslett, "Distributed authorization by multiparty trust negotiation," in *ESORICS 2008*. Springer, 2008, pp. 282–299.

[24] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 1999, pp. 228–241.

[25] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," in *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, Oct. 1991, pp. 165–182, *Operating System Review*, 253(5).

[26] F. B. Schneider, K. Walsh, and E. G. Sirer, "Nexus Authorization Logic (NAL): Design rationale and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 1, pp. 8:1–8:28, Jun. 2011.

[27] M. Abadi, "Access control in a core calculus of dependency," in *Proc. 11th ACM SIGPLAN Int'l Conf. on Functional Programming*. New York, NY, USA: ACM, 2006, pp. 263–273.

[28] N. Broberg and D. Sands, "Flow locks: Towards a core calculus for dynamic flow policies," in *Programming Languages and Systems*, Mar. 2006, pp. 180–196.

[29] R. S. Sandhu, "Role hierarchies and constraints for lattice-based access controls," in *Proc. 4th European Symp. on Research in Computer Security (ESORICS)*, Sep. 1996.

[30] D. Ferraiolo and R. Kuhn, "Role-based access controls," in *15th National Computer Security Conference*, 1992.

[31] L. Zheng and A. C. Myers, "End-to-end availability policies and noninterference," in *Proc. 18th IEEE Computer Security Foundations Workshop*, Jun. 2005, pp. 272–286.

[32] J. Liu and A. C. Myers, "Defining and enforcing referential security," in *Proc. 3rd Conf. on Principles of Security and Trust*, Apr. 2014, pp. 199–219.

[33] O. Arden, J. Liu, and A. C. Myers, "Flow-limited authorization: Technical report," May 2015.

[34] S. Chong and A. C. Myers, "Decentralized robustness," in *Proc. 19th IEEE Computer Security Foundations Workshop*, Jul. 2006, pp. 242–253.

[35] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif 3.0: Java information flow," Jul. 2006, software release, http://www.cs.cornell.edu/jif.

[36] M. V. Tripunitara and N. Li, "A theory for comparing the expressive power of access control models," *Journal of Computer Security*, vol. 15, no. 2, pp. 231–272, 2007.

[37] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *International Journal of Information Security*, vol. 6, no. 2–3, Mar. 2007.

[38] N. Broberg, B. van Delft, and D. Sands, "Paragon for practical programming with information-flow control," in *Programming Languages and Systems: 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*. Springer, 2013, pp. 217–232.

[39] J. H. Saltzer, "Protection and the control of information sharing in Multics," *Comm. of the ACM*, vol. 17, no. 7, pp. 388–402, Jul. 1974.

[40] N. Broberg, B. van Delft, and D. Sands, "The anatomy and facets of dynamic policies," in *IEEE Symp. on Computer Security Foundations*. IEEE, 2015.

[41] B. Montagu, B. C. Pierce, and R. Pollack, "A theory of information-flow labels," in *Proc. 26th IEEE Symp. on Computer Security Foundations*, Jun. 2013, pp. 3–17.

[42] C. Ellison, "SPKI requirements," Internet RFC-2692, Sep. 1999.

[43] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "SPKI certificate theory," Internet RFC-2693, Sep. 1999.

[44] M. Y. Becker, C. Fournet, and A. D. Gordon, "SecPAL: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, vol. 18, no. 4, pp. 619–665, 2010.

[45] Y. Gurevich and I. Neeman, "DKAL: Distributed-knowledge authorization language," in *IEEE Symp. on Computer Security Foundations*. IEEE, 2008, pp. 149–162.

[46] M. Abadi, M. Burrows, B. W. Lampson, and G. D. Plotkin, "A calculus for access control in distributed systems," *ACM Trans. on Programming Languages and Systems*, vol. 15, no. 4, pp. 706–734, 1993.

[47] N. Li, B. N. Grosof, and J. Feigenbaum, "Delegation logic: A logic-based approach to distributed authorization," *ACM Transactions on Information and System Security (TISSEC)*, vol. 6, no. 1, pp. 128–171, 2003.

[48] N. Li, J. C. Mitchell, and W. H. Winsborough, "Design of a role-based trust-management framework," in *Proc. IEEE Symp. on Security and Privacy*, 2002, pp. 114–130.

[49] A. Birgisson, J. G. Politz, Úlfar Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[50] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, "A core calculus of dependency," in *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 1999, pp. 147–160.

[51] D. Garg and F. Pfenning, "Non-interference in constructive authorization logic," in *Proc. 19th IEEE Computer Security Foundations Workshop*, 2006.

[52] M. Y. Becker, "Information flow in trust management systems," *Journal of Computer Security*, vol. 20, no. 6, pp. 677–708, 2012.

[53] J. W. Bryans, M. Koutny, L. Mazaré, and P. Y. Ryan, "Opacity generalised to transition systems," *International Journal of Information Security*, vol. 7, no. 6, pp. 421–435, 2008.

[54] N. Broberg and D. Sands, "Paralocks—role-based information flow control and beyond," in *Proc. 37th ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2010.

[55] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *Proc. IEEE Symp. on Security and Privacy*, 2011, pp. 165–179.

[56] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell, "Disjunction category labels," in *Proceedings of the 16th Nordic conference on Information Security Technology for Applications*, 2011, pp. 223–239.

A Operational semantics of Fλ

$\boxed{\langle e,\, \sigma\rangle \xrightarrow{\alpha} \langle e',\, \sigma'\rangle}$

[E-IfTrue]
$$\frac{\sigma_{\mathcal{H}};\, \ell;\, pc^{\rightarrow} \vdash p \succcurlyeq q}{\langle \texttt{if}\ p{\succcurlyeq}q\ \texttt{with}\ \ell\ \texttt{at}\ pc\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, \sigma\rangle \xrightarrow{pc(p,q,\ell)} \langle e_1,\, \sigma'\rangle}$$

[E-IfFalse]
$$\frac{\sigma_{\mathcal{H}};\, \ell;\, pc^{\rightarrow} \nvdash p \succcurlyeq q}{\langle \texttt{if}\ p{\succcurlyeq}q\ \texttt{with}\ \ell\ \texttt{at}\ pc\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, \sigma\rangle \xrightarrow{pc(p,q,\ell)} \langle e_2,\, \sigma\rangle}$$

[E-Trust]
$$\frac{\mathcal{H}' = \sigma_{\mathcal{H}}[r \mapsto \sigma_{\mathcal{H}}(r), (p{\succcurlyeq}q, \ell)]}{\langle p\ \texttt{trust}\ q\ \texttt{with}\ \ell\ \texttt{at}\ r,\, \sigma\rangle \xrightarrow{r(+,p,q,\ell)} \langle (),\, \langle \mathcal{H}';\, \sigma_M\rangle\rangle}$$

[E-Revoke]
$$\frac{\mathcal{H}' = [r \mapsto \mathsf{rev}(\sigma_{\mathcal{H}}, r, p{\succcurlyeq}q, \ell) \mid \sigma_{\mathcal{H}};\, pc;\, \ell \Vdash r \succcurlyeq pc^{\rightarrow}]}{\langle p\ \texttt{revoke}\ q\ \texttt{with}\ \ell\ \texttt{at}\ pc,\, \sigma\rangle \xrightarrow{pc(-,p,q,\ell)} \langle (),\, \langle \mathcal{H}';\, \sigma_M\rangle\rangle}$$

[E-Relabel] $\langle \texttt{relabel}\ v\ \ell,\, \sigma\rangle \dot{\rightarrow} \langle v,\, \sigma\rangle$

[E-Deref] $\langle !m^{\tau@a},\, \sigma\rangle \xrightarrow{m^{\tau@a}} \langle \sigma_M(m^{\tau@a}),\, \sigma\rangle$

[E-Assign] $\langle m^{\tau@a} := v,\, \sigma\rangle \xrightarrow{(m^{\tau@a}, v)} \langle (),\, \langle \sigma_{\mathcal{H}}, \sigma_M[m^{\tau@a} \mapsto v]\rangle\rangle$

[E-Alloc]
$$\frac{m = \mathit{newloc}(\sigma_M)}{\langle \texttt{ref}^{\tau@a} v,\, \sigma\rangle \xrightarrow{(m^{\tau@a}, v)} \langle m^{\tau@a},\, \langle \sigma_{\mathcal{H}}, \sigma_M[m^{\tau@a} \mapsto v]\rangle\rangle}$$

[E-Apply] $\langle (\lambda(x{:}\tau)[D\,;pc].\, e)\ v,\, \sigma\rangle \dot{\rightarrow} \langle e[v/x],\, \sigma\rangle$

[E-Context]
$$\frac{\langle e,\, \sigma\rangle \xrightarrow{\alpha} \langle e',\, \sigma'\rangle}{\langle E[e],\, \sigma\rangle \xrightarrow{\alpha} \langle E[e'],\, \sigma'\rangle}$$

$$\mathsf{rev}(\mathcal{H}, r, p{\succcurlyeq}q, \ell) \triangleq \mathcal{H}(r) - \{(p{\succcurlyeq}q, \ell') \in \mathcal{H}(r) \mid \mathcal{H};\, pc;\, \ell \Vdash \ell \sqsubseteq \ell'\}$$

B Fλ typing rules

$\boxed{\Gamma\,;\,D\,;\,pc \vdash e : \tau}$

[T-INT] $\Gamma\,;\,D\,;\,pc \vdash n : \mathtt{int}_\bot$ [T-UNIT] $\Gamma\,;\,D\,;\,pc \vdash () : \mathtt{unit}_\bot$ [T-VAR] $\dfrac{x{:}\tau \in \Gamma}{\Gamma\,;\,D\,;\,pc \vdash x : \tau}$

[T-LOC] $\dfrac{FV(\tau@a) = \emptyset}{\Gamma\,;\,D\,;\,pc \vdash m^{\tau@a} : (\tau@a\ \mathtt{ref})_\bot}$ [T-REF] $\dfrac{\Gamma\,;\,D\,;\,pc \vdash e : \tau \qquad \Vdash pc \sqsubseteq \tau \qquad \Vdash \tau \sqsubseteq a}{\Gamma\,;\,D\,;\,pc \vdash \mathtt{ref}^{\tau@a} e : (\tau@a\ \mathtt{ref})_\bot}$

[T-DEREF] $\dfrac{\Gamma\,;\,D\,;\,pc \vdash e : (\tau@a\ \mathtt{ref})_\ell \qquad \Vdash pc \sqcup \ell \sqsubseteq a}{\Gamma\,;\,D\,;\,pc \vdash !e : \tau \sqcup \ell}$

[T-ABS] $\dfrac{\Gamma, x{:}\tau'\,;\,D'\,;\,pc' \vdash e : \tau}{\Gamma\,;\,D\,;\,pc \vdash \lambda(x{:}\tau')[D'\,;\,pc'].\, e : ((x{:}\tau') \xrightarrow{D'\,;\,pc'} \tau)_\bot}$

[T-ASSIGN] $\dfrac{\Gamma\,;\,D\,;\,pc \vdash e_1 : (\tau@a\ \mathtt{ref})_\ell \qquad \Gamma\,;\,D\,;\,pc \vdash e_2 : \tau \qquad \Vdash pc \sqcup \ell \sqsubseteq \tau \qquad \Vdash pc \sqcup \ell \sqsubseteq a}{\Gamma\,;\,D\,;\,pc \vdash e_1 := e_2 : \mathtt{unit}_\bot}$

[T-PRIN1] $\dfrac{\Gamma\,;\,D\,;\,pc \vdash p : \mathtt{prin}_\ell}{\Gamma\,;\,D\,;\,pc \vdash p^\pi : \mathtt{prin}_\ell}$ [T-PRIN2] $\dfrac{\Gamma\,;\,D\,;\,pc \vdash p_1 : \mathtt{prin}_{\ell_1} \qquad \Gamma\,;\,D\,;\,pc \vdash p_2 : \mathtt{prin}_{\ell_2}}{\Gamma\,;\,D\,;\,pc \vdash p_1 \oplus p_2 : \mathtt{prin}_{\ell_1 \sqcup \ell_2}}$

[T-SUB] $\dfrac{\Gamma\,;\,D\,;\,pc \vdash e : \beta_\ell \qquad \Gamma \vdash \beta \leq \beta' \qquad \Vdash \ell \sqsubseteq \ell'}{\Gamma\,;\,D\,;\,pc \vdash e : \beta'_{\ell'}}$

[T-APPLY] $\dfrac{\Gamma\,;\,D\,;\,pc \vdash e_1 : ((x{:}\tau') \xrightarrow{D'\,;\,pc'} \tau)'_\ell \qquad D \vdash D'[\ell/x] \qquad \Vdash pc \sqcup \ell' \sqsubseteq pc' \qquad \Gamma\,;\,D\,;\,pc \vdash \ell : \tau'[\ell/x]}{\Gamma\,;\,D\,;\,pc \vdash e\ \ell : \tau[\ell/x] \sqcup \ell'}$

[T-TRUST] $\dfrac{\Gamma\,;\,D\,;\,pc \vdash \ell_i : \mathtt{prin}_{\ell'_i}\ \ i \in \{1,2,3,4\} \qquad \Vdash \bigsqcup_i \ell'_i \sqcup pc \sqsubseteq \ell_3 \qquad \Vdash \ell_4 \succcurlyeq \ell_3}{\Gamma\,;\,D\,;\,pc \vdash \ell_1\ \mathtt{trust}\ \ell_2\ \mathtt{with}\ \ell_3\ \mathtt{at}\ \ell_4 : \mathtt{unit}_\bot}$

[T-REVOKE] $\dfrac{\Gamma\,;\,D\,;\,pc \vdash \ell_i : \mathtt{prin}_{\ell'_i}\ \ i \in \{1,2,3,4\} \qquad \Vdash \bigsqcup_i \ell' \sqcup pc \sqsubseteq \ell_3 \qquad \Vdash \ell_4 \succcurlyeq \ell_3}{\Gamma\,;\,D\,;\,pc \vdash \ell_1\ \mathtt{revoke}\ \ell_2\ \mathtt{with}\ \ell_3\ \mathtt{at}\ \ell_4 : \mathtt{unit}_\bot}$

[T-RELABEL] $\dfrac{\Gamma\,;\,D\,;\,pc \vdash e : \beta_\ell \qquad [c \mapsto D]; pc; \ell' \Vdash \ell \sqsubseteq \ell'}{\Gamma\,;\,D\,;\,pc \vdash \mathtt{relabel}\ e\ \ell' : \beta_{\ell'}}$

[T-IF] $\dfrac{\begin{array}{c}\Gamma\,;\,D\,;\,pc \vdash \ell_i : \mathtt{prin}_{\ell'_i}\ \ i \in \{1,2,3,4\} \\ \Vdash \bigsqcup_i \ell'_i \sqcup pc \sqsubseteq \ell_4 \qquad \Gamma\,;\,D, (\ell_1 \succcurlyeq \ell_2, \ell_3)\,;\,\ell_3 \sqcup \ell_4 \vdash e_1 : \tau \qquad \Gamma\,;\,D\,;\,\ell_3 \sqcup \ell_4 \vdash e_2 : \tau\end{array}}{\Gamma\,;\,D\,;\,pc \vdash \begin{array}{l}\mathtt{if}\ \ell_1 \succcurlyeq \ell_2\ \mathtt{with}\ \ell_3\ \mathtt{at}\ \ell_4 \\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\end{array} : \bigsqcup_i \ell'_i \sqcup \ell_3 \sqcup \tau}$

The rule APP contains the judgment $D \vdash D'[\ell/x]$ which means that for each entry $(p \succcurlyeq q, \ell) \in D'[\ell/x]$ we have $D; \top^{\rightarrow} \wedge \bot^{\leftarrow}; \ell \vdash p \succcurlyeq q$. This ensures the delegations of D' are derivable from D without relabeling.

C ARBAC methods

User–role assignment.

$assignUser(a, u, r, \mathsf{pc}, \ell)\{$
  if $\exists(ar, cr, mn, mx) \in can\_assign$
  such that
    $\mathcal{H}; c; \mathsf{pc}; \ell \Vdash a \succcurlyeq ar$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash u \succcurlyeq cr$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash r \succcurlyeq mn$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash mx \succcurlyeq r$
  then
    let $\ell' = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge r)^{\leftarrow}$
    $\mathcal{H} := \mathcal{H} \cup [c \mapsto (u \succcurlyeq r, \ell')]$
$\}$

Authorize a's assignment of role r to user u. If the FLAM judgments hold, a delegation $u \succcurlyeq r$ is created with the integrity of ar and r.

$revokeUser(a, u, r, \mathsf{pc}, \ell)\{$
  if $\exists(ar, mn, mx) \in can\_revoke$
  such that
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash a \succcurlyeq ar$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash r \succcurlyeq mn$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash mx \succcurlyeq r$
  then
    let $\ell' = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge r)^{\leftarrow}$
    $\mathcal{H} := \bigcup_{c \in \mathrm{dom}(\mathcal{H})} [c \mapsto \mathsf{rev}(\mathcal{H}, c, u \succcurlyeq r, \ell')]$
$\}$

Authorize a's revocation of u's membership in role r. If the FLAM judgments hold, all delegations $(u \succcurlyeq r, \ell'')$ where $\ell' \sqsubseteq \ell''$ are revoked.

Permission–role assignment.

$assignPermission(a, p, r, \mathsf{pc}, \ell)\{$
  if $\exists(ar, cr, mn, mx) \in can\_assignp$
  such that
    $\mathcal{H}; c; \mathsf{pc}; \ell \Vdash a \succcurlyeq ar$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash p \succcurlyeq cr$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash r \succcurlyeq mn$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash mx \succcurlyeq r$
  then
    let $\ell' = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge p)^{\leftarrow}$
    $\mathcal{H} := \mathcal{H} \cup [c \mapsto (r \succcurlyeq p, \ell')]$
$\}$

Authorize a's grant of permission p to role r. If the FLAM judgments hold, a delegation $r \succcurlyeq p$ is created with the integrity of ar and p.

$revokePermission(a, p, r, \mathsf{pc}, \ell)\{$
  if $\exists(ar, mn, mx) \in can\_revokep$
  such that
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash a \succcurlyeq ar$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash r \succcurlyeq mn$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash mx \succcurlyeq r$
  then
    let $\ell' = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge p)^{\leftarrow}$
    $\mathcal{H} := \bigcup_{c \in \mathrm{dom}(\mathcal{H})} [c \mapsto \mathsf{rev}(\mathcal{H}, p, r \succcurlyeq p, \ell')]$
$\}$

Authorize a's revocation of permission p for role r. If the FLAM judgments hold, all delegations $(r \succcurlyeq p, \ell'')$ where $\ell' \sqsubseteq \ell''$ are revoked.

Role–role assignment.

$addToRange(a, mn, mx, r, \mathsf{pc}, \ell)\{$
  if $\exists(ar, mn, mx) \in can\_modify$
  such that
    $r \neq mn$ and $r \neq mx$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash a \succcurlyeq ar$
  then
    let $\ell_r = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge r)^{\leftarrow}$
    let $\ell_{mn} = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge mn)^{\leftarrow}$
    $\mathcal{H} := \mathcal{H} \cup [o \mapsto (mx \succcurlyeq r, \ell_r), (r \succcurlyeq mn, \ell_{mn})]$
}

Authorize a's addition of r to range $[mn, mx]$. If the FLAM judgments hold, two delegations are created: $mx \succcurlyeq r$ with the integrity of ar and r, and $r \succcurlyeq mn$ with the integrity of ar and mn.

$removeFromRange(a, mn, mx, r, \mathsf{pc}, \ell)\{$
  if $\exists(ar, mn, mx) \in can\_modify$
  such that
    $r \neq mn$ and $r \neq mx$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash a \succcurlyeq ar$
  then
    let $\ell_r = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge r)^{\leftarrow}$
    $\mathcal{H} := \bigcup_{c \in \mathrm{dom}(\mathcal{H})} [c \mapsto \mathsf{rev}(\mathcal{H}, p, mx \succcurlyeq r, \ell_r)]$
    let $\ell_{mn} = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge mn)^{\leftarrow}$
    $\mathcal{H} := \bigcup_{c \in \mathrm{dom}(\mathcal{H})} [c \mapsto \mathsf{rev}(\mathcal{H}, p, r \succcurlyeq mn, \ell_{mn})]$
}

Authorize a's removal of r from range $[mn, mx]$. If the FLAM judgments hold, two revocations occur: $(mx \succcurlyeq r, \ell'_r)$ where $\ell_r \sqsubseteq \ell'_r$ and $(r \succcurlyeq mn, \ell'_{mn})$ where $\ell_{mn} \sqsubseteq \ell'_{mn}$.

$addAsSenior(a, r, s, \mathsf{pc}, \ell)\{$
  if $\exists(ar, mn, mx) \in can\_modify$
  such that
    $\mathcal{H}; c; \mathsf{pc}; \ell \Vdash a \succcurlyeq ar$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash r \succcurlyeq mn$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash mx \succcurlyeq r$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash s \succcurlyeq mn$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash mx \succcurlyeq s$
  then
    let $\ell' = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge s)^{\leftarrow}$
    $\mathcal{H} := \mathcal{H} \cup [c \mapsto (r \succcurlyeq s, \ell')]$
}

Authorize a's addition of r as a senior to s.

$removeAsSenior(a, r, s, \mathsf{pc}, \ell)\{$
  if $\exists(ar, mn, mx) \in can\_modify$
  such that
    $\mathcal{H}; c; \mathsf{pc}; \ell \Vdash a \succcurlyeq ar$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash r \succcurlyeq mn$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash mx \succcurlyeq r$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash s \succcurlyeq mn$
    $\mathcal{H}; c; \mathsf{pc}; \ell \wedge ar^{\leftarrow} \Vdash mx \succcurlyeq s$
  then
    let $\ell' = (\mathsf{pc} \sqcup \ell) \wedge (ar \wedge s)^{\leftarrow}$
    $\mathcal{H} := \bigcup_{c \in \mathrm{dom}(\mathcal{H})} [c \mapsto \mathsf{rev}(\mathcal{H}, p, r \succcurlyeq s, \ell')]$
}

Authorize a's removal of r as a senior to s.

$$\mathsf{rev}(\mathcal{H}, c, p \succcurlyeq q, \ell) \triangleq \mathcal{H}(c) - \{(p \succcurlyeq q, \ell') \in \mathcal{H}(c) \mid \mathcal{H}; c; \mathsf{pc}; \ell \Vdash \ell \sqsubseteq \ell'\}$$

Revocation operation. Returns $\mathcal{H}(c)$ with delegations $(p \succcurlyeq q, \ell')$ where $\ell \sqsubseteq \ell'$ removed.

D Acts-for proof search algorithm

```
function actsForProof(ActsForQuery query, ProofSearchCache cache):
  input: query - the acts-for relationship being queried
         cache - initially empty; caches intermediate results obtained during the proof search
  returns: a ProofSearchResult, containing a result type (PROVED, PRUNED, or FAILED)
           and some optional data (a proof for PROVED results, or a progress condition for PRUNED results)

  // Check the cache.
  if cache has cached result for query: return cached result

  // Cache miss. Put a placeholder result for the query in the cache (to avoid infinite recursion).
  update(cache, query, PRUNED, query)

  // Search for a proof.
  ProofSearchResult result ← findActsForProof(query, cache)
  update(cache, query, result.type, result.data)
  return result

function findActsForProof(ActsForQuery query, ProofSearchCache cache):
  // A boolean formula expressing the conditions for making further progress on this proof, in the
  // event the search is pruned.
  ProgressCondition progressCondition ← False

  for each applicable rule instance r:
    // ⊥ is a special boolean formula that is an identity with respect to both conjunction and
    // disjunction.
    ProgressCondition ruleConditions ← ⊥
    boolean success ← true
    list subproofs ← []

    for each premise p in r:
      ProofSearchResult subqueryResult ← actsForProof(p, cache)
      if subqueryResult.type = PROVED: add subqueryResult.proof to subproofs
      else:
        success ← false
        if subqueryResult.type = PRUNED:
          ruleConditions ← ruleConditions ∧ subqueryResult.progressCondition
        else if subqueryResult.type = FAILED:
          ruleConditions ← ⊥
          break

    if success:
      return new ProofSearchResult(type ← PROVED, data ← new Proof(r, subproofs))

    progressCondition ← progressCondition ∨ ruleConditions

  // No proof found.
  if progressCondition = False:
    return new ProofSearchResult(type ← FAILED)

  return new ProofSearchResult(type ← PRUNED, data ← progressCondition)
```