2019 IEEE European Symposium on Security and Privacy (EuroS&P)

Programming with Flow-Limited Authorization: Coarser is Better

Mathias V. Pedersen
Department of Computer Science
Aarhus University
mvp@cs.au.dk

Stephen Chong
Harvard School of Engineering and Applied Sciences
Cambridge, Massachusetts
chong@seas.harvard.edu

***Abstract*—Applications that handle sensitive information need to express and reason about the trust relationships between security principals. Such reasoning is difficult because the trust relationships are dynamic, and must thus be reasoned about at runtime when discovering and reasoning about trust relationships might inadvertently reveal confidential information and be subject to manipulation by untrusted principals.**

The Flow-Limited Authorization Model (FLAM) by Arden *et al.* exactly meets these needs. However, previous attempts to use FLAM in a programming language have not reaped the full benefits of the model.

We present Flamio, an instantiation of FLAM in a language with coarse-grained dynamic information-flow control (IFC) which naturally lends itself to dynamic enforcement techniques. In our implementation of Flamio, the FLAM proof search rules for deriving trust relationships are implemented as regular Flamio computations: the IFC requirements for FLAM proof rules are a natural fit with coarse-grained information-flow mechanisms. Flamio even supports remote procedure calls, and thus seamlessly supports FLAM's distributed proof search.

We have implemented Flamio as a Haskell library, and proved that a calculus based on Flamio enforces a noninterference-based security guarantee. We have implemented several case studies, demonstrating the expressiveness and usefulness of Flamio in distributed settings, as well as our novel approach to control label creep during proof search.

I. Introduction

Most modern systems require some form of authorization to control access to data. Such authorization mechanisms tend to be complex and hard to get right, even though the correctness of such components is vital for the security of the system and its users [2]. The behavior of such systems is often dynamic, with access control continually changing on a per-user basis. Thus, the authorization mechanisms used in these systems also need to be dynamic and able to securely control changes to user privileges [2] at runtime. Furthermore, the mere existence of a trust relationship between two principals may leak confidential information to an attacker if the trust relationship was established based on the result of a confidential computation. Dually, if attackers can provide trust relationships that influence otherwise high-integrity computations, they may be able to influence access control decisions inappropriately. These issues are especially ubiquitous in distributed systems, where nodes may not agree on the trust relationship among principals.

The Flow-Limited Authorization Model (FLAM) [1] is an expressive security model designed for rigorous reasoning about dynamic changes to authorization policies in a distributed setting, where nodes can forward trust checking requests to other nodes in the system. FLAM also guarantees that no confidential information is leaked to an attacker through the trust checking mechanism and that security principals cannot inappropriately influence the trust relationships. This makes FLAM ideal for highly dynamic security policies involving many principals with intricate trust relationships. However, currently, no programming language that builds on top of FLAM reaps the full benefit of the authorization logic.

In this paper we introduce Flamio, which takes a coarse-grained approach to information-flow control (IFC). Fine-grained IFC (as seen in information-flow aware languages like FlowCaml [3] and Jif [4]) labels individual values with security labels. By contrast, coarse-grained IFC does not label individual values, but instead labels the *computational context* in which the program is running with a single label. Coarse-grained IFC lends itself naturally to dynamic enforcement techniques as demonstrated by its success in both operating systems [5]–[8] and programming languages [9]–[12].

We show that FLAM's rules for proving trust relationships have a straightforward encoding in the coarse-grained IFC setting of Flamio, which is inspired by the work on Labeled IO (LIO) [9]. As evidence of the straightforward encoding, we implement Flamio as a Haskell library [13] that encapsulates FLAM's proof search for trust relationships in an information-flow aware computational context, which ensures that the proof search itself does not leak confidential information to, and cannot be inappropriately influenced by, attackers. Leveraging FLAM's decentralized authorization model, Flamio supports distributed proof search of trust relationships, where nodes can forward trust checking to other nodes in the system. To demonstrate the usefulness of distributed proof search of trust relationships in a setting with dynamic security policies we present three case studies involving distributed computations with confidential information and mutually distrusting principals that must cooperate to perform their tasks.

The case studies also demonstrate a novel technique for mitigating the problem of *label creep* during proof search.

Label creep refers to the label of the computational context *creeping* up the information-flow lattice as the program executes. Previous work [9] mitigates label creep using the toLabeled construct. However, we cannot use this technique to mitigate label creep during search for proofs of trust relationships. We present a technique that gives the programmer fine-grained control over the way proofs of trust are derived, and how the proofs can affect the label on the computational context, providing the programmer with the ability to control label creep during proof search.

We also present a calculus for Flamio, which formally proves that Flamio enforces a noninterference-based [14] security guarantee that attackers cannot leak or corrupt information, despite the incorporation of FLAM as an expressive, dynamic mechanism to state and reason about trust relationships.

This paper makes the following contributions.

- We show how the FLAM principal lattice integrates cleanly into a language with coarse-grained dynamic information-flow control and distributed trust checking.
- We present a formal model of Flamio and prove that the language guarantees noninterference.
- We present an implementation of Flamio as a Haskell library, along with an efficient implementation of the FLAM authorization logic to decide trust relationships.
- We describe a novel approach to avoid the problem of *label creep* during proof search.
- We present several examples of distributed information-flow problems all of which can easily be modeled using Flamio.

The rest of the paper is structured as follows. Section II introduces the necessary concepts from FLAM and LIO, and Section III demonstrates how Flamio combines FLAM and LIO through several examples. Section IV describes the formal calculus for Flamio and how the FLAM judgment for deciding trust relations is modeled in a coarse-grained setting. Section V defines the attacker model we consider in this work and presents the formal security guarantees offered by Flamio. Section VI discusses the implementation in Haskell and presents three case studies demonstrating the use of Flamio. Section VII presents related work and Section VIII concludes.

II. Background on FLAM and LIO

Before discussing how FLAM and LIO fit together, we briefly introduce each separately. We first highlight the essential parts of FLAM necessary for this work; more details can be found in Arden *et al.* [1]. Similarly, we highlight the essential parts of LIO, and additional details can be found in Stefan *et al.* [9].

A. The FLAM principal lattice

Figure 1 describes the syntax of FLAM principals. The grammar is parametric in a set $\mathcal{N}$ of names representing principals like Alice and Bob. Given a principal p, FLAM can talk about

$$\begin{aligned} &n \in \mathcal{N} \\ &p ::= \bot \mid \top \mid n \mid p \wedge p \mid p \vee p \mid p^{\rightarrow} \mid p^{\leftarrow} \mid p : p \end{aligned}$$

Fig. 1: Syntax of FLAM

the confidentiality and integrity of p using *basis projections* $p^{\rightarrow}$ and $p^{\leftarrow}$ respectively. The principal $p^{\rightarrow}$ represents the authority to learn anything that p can learn, and $p^{\leftarrow}$ represents the authority to modify anything that p can modify. Given principals p and q, FLAM can also represent the authority of *both* p and q as $p \wedge q$ or the authority of *either* p or q as $p \vee q$. This forms a lattice $(\mathcal{P}, \succcurlyeq)$ with a partial order $\succcurlyeq$ (pronounced *acts for*), that represents trust: if $p \succcurlyeq q$ principal p is allowed to act on behalf of q, i.e., q trusts p to act on its behalf. Principals $\bot$ and $\top$ represent the least and most trusted principals respectively, and operations $\wedge$ and $\vee$ are the lattice join and meet operations respectively. The principals in $\mathcal{P}$ are given by the grammar in Figure 1.[1] Besides basis projections $\rightarrow$ and $\leftarrow$, FLAM also defines *ownership projections* $o\!:\!p$ representing the same authority as the principal p, but where the owner o controls which principals $o\!:\!p$ trusts. For example, the delegation Acme: Bob $\succcurlyeq$ Acme implies that Bob can act for Acme, but it is not the case that Alice can act for Acme: Bob, even if Alice acts for Bob. We will use ownership extensively in the example presented in Section VI-B.

a) An information-flow ordering: An important distinction between FLAM and other authorization models is that FLAM unifies trust and information-flow into a single concept. That is, in FLAM, principals denote both entities with security concerns and information-flow labels that can be used to restrict the propagation of information in a system. The acts-for ordering describes trust relationships between principals, and is used to define an information-flow ordering that describes the permitted propagation of information. Specifically, FLAM defines the operations

$$p \sqsubseteq q \doteq q^{\rightarrow} \wedge p^{\leftarrow} \succcurlyeq p^{\rightarrow} \wedge q^{\leftarrow} \tag{1}$$

$$p \sqcup q \doteq (p \wedge q)^{\rightarrow} \wedge (p \vee q)^{\leftarrow} \tag{2}$$

$$p \sqcap q \doteq (p \vee q)^{\rightarrow} \wedge (p \wedge q)^{\leftarrow} \tag{3}$$

That is, $p \sqsubseteq q$ (pronounced *p flows to q*) if q acts for the confidentiality of p, and p acts for the integrity of q. So if $p \sqsubseteq q$, information labeled with principal p can safely be relabeled to principal q, as q is at least as confidential, and has no more integrity, than p. The *join* of p and q, written $p \sqcup q$ is defined as the principal with the authority of both p's and q's confidentiality, and the authority of either p's or q's integrity. The *meet* of p and q, written $p \sqcap q$ is defined dually as the confidentiality of either p or q, and the integrity of both p and q. This forms a lattice $(\mathcal{P}, \sqsubseteq)$ with the partial order $\sqsubseteq$, where the bottom element $\bot^{\sqsubseteq} \doteq \bot^{\rightarrow} \wedge \top^{\leftarrow}$ represents the least confidential and most trusted principal, and the top

[1]Formally, the principals in the lattice is the equivalence class of $\mathcal{P}$ modulo the relation $\equiv$ where $a \equiv b \iff a \succcurlyeq b$ and $b \succcurlyeq a$.

element $\top_{\sqsubseteq} \triangleq \top^{\rightarrow} \wedge \bot^{\leftarrow}$ represents the most confidential and least trusted principal.[2]

b) Voice of a principal: Finally, FLAM defines the *voice* of a principal p, denoted $\nabla(p)$, as the minimum integrity needed to influence the flow of information labeled p. Using the voice operator, FLAM defines a *speaks for* relation [15], [16] between principals as "principal p speaks for principal q if p acts for the voice of q". Formally p speaks for q if $p \succcurlyeq \nabla(q)$. Any principal is equivalent to the conjunction of a confidentiality projection and an integrity projection, i.e., $\forall p.\ \exists q, r.\ p \equiv q^{\rightarrow} \wedge r^{\leftarrow}$. The voice of a principal p with normal form $q^{\rightarrow} \wedge r^{\leftarrow}$ is then defined as $\nabla(p) = q^{\leftarrow} \wedge r^{\leftarrow}$ [1].

B. Coarse-grained information flow using LIO.

LIO [9] is a Haskell library for dynamic information-flow control. LIO is parametric in the label model and takes a coarse-grained approach to information-flow using a *floating label model*: instead of attaching a label to each value in the program, the *computational context* is protected with a single label called the *current label*. Throughout the execution of the program the current label will "float up" the information-flow lattice as more confidential (or less trustworthy) information is brought into the computational context. The current label restricts what data can be modified, ensuring that non-confidential side effects do not depend on confidential information, and dually that trusted effects do not depend on untrusted information.

The type of LIO computations gives rise to a monad [17] that encapsulates raising the current label. The monadic structure of LIO makes programming with it convenient in Haskell. Specifically, the operation `return` e embeds a pure expression e into the LIO computational context, and the operation $e_1 \gg\!= e_2$ (pronounced *bind*) chains together monadic LIO operations e_1 and e_2. Throughout the paper, we use Haskell's *do notation* in examples, which can be desugared into the calculus presented in Section IV.

In addition to the current label, LIO also provides a *clearance label*, imposing an upper bound on the current label. The clearance label gives LIO a form of access control by restricting which data can be observed and modified within a computational context. For instance, given two labels ℓ_{clr} and ℓ_{data} such that $\ell_{\mathsf{data}} \not\sqsubseteq \ell_{\mathsf{clr}}$, a computation with clearance label ℓ_{clr} cannot access information labeled with label ℓ_{data} because that would require the current label to float up to ℓ_{data}, but ℓ_{clr} is an upper bound on the current label and $\ell_{\mathsf{data}} \not\sqsubseteq \ell_{\mathsf{clr}}$.

a) Labeled values: In addition to protecting every value in the computational context by a single label ℓ_{cur}, LIO also allows computations to associate explicit labels with particular values. This allows computations to handle data of different labels in the same context. The type of labeled values of type `a` is written in Haskell as `Labeled l a`, where `l` is the type of labels.

[2]Once again, the elements is equivalence classes of $\mathcal{P}$ modulo the relation $\equiv$ defined as $a \equiv b \iff a \sqsubseteq b$ and $b \sqsubseteq a$.

Labeled values are typically used to incorporate sensitive information such as usernames and passwords into the computational context. LIO provides three operations for working with labeled values:

```
label :: Label l => a -> l -> LIO l (Labeled l a)
unlabel :: Label l => Labeled l a -> LIO l a
labelOf :: Label l => Labeled l a -> l
```

Here, `Label l` is a typeclass constraint specifying that the type `l` must be an instance of the `Label` typeclass, meaning that `l` must have operations $\sqcup$, $\sqcap$ and $\sqsubseteq$. The operation `label` takes an expression e and a label ℓ, and labels e with the label ℓ. When the value of the labeled value is needed, the operation `unlabel` must be invoked, which gets back the value and raises the current label ℓ_{cur} to $\ell_{\mathsf{cur}} \sqcup \ell$, while checking that the new current label flows to the clearance label ℓ_{clr} of the computation.

Finally, `labelOf` extracts the label of a labeled value. This operation does not return a value in the LIO monad, and thus no information-flow checks are performed when invoking `labelOf`. In other words, the label is protected only by the current label [9]. This fact is important when we define trust checking using strategies in Section IV-C.

b) Preventing label creep: As the program executes and confidential or untrusted information enters the computational context through `unlabel` operations, the current label continues to creep upwards, restricting the possible side effects. To avoid unnecessary label creep LIO introduces the following operation:

```
toLabeled :: Label l => l -> LIO l a ->
                     LIO l (Labeled l a)
```

Evaluating `toLabeled l e` when the current label is ℓ_{cur} will evaluate `e` and then reset the current label to ℓ_{cur}. To remain secure, the result is labeled with the label `l`. Furthermore, LIO checks that the evaluation of `e` never raises the current label above `l`.

III. Introduction to Flamio by Example

In this section we informally introduce Flamio and the concepts needed to combine LIO and FLAM. Section IV will then formalize the intuitions presented in this section. We proceed by demonstrating the usefulness of Flamio in the context of a secure, decentralized banking application, which we also discuss in Section VI. The intended security policy of the bank is that a user u can transfer money from an account only if the owner of the account trusts u.

Flamio incorporates FLAM into LIO by using the FLAM information-flow lattice $(\mathcal{P}, \sqsubseteq)$ as the label model of LIO. As FLAM unifies principals and labels, this allows Flamio to also reason about trust relationships between principals using the trust lattice $(\mathcal{P}, \succcurlyeq)$. Given a named principal $n \in \mathcal{N}$ (e.g., Alice or Bob) we call n a *node* when referring to the machine that executes code on n's behalf. We assume each named principal has a corresponding machine executing code on its behalf. Initially, the current label of node n is $\bot^{\rightarrow} \wedge n^{\leftarrow}$ and its clearance label is $n^{\rightarrow} \wedge \bot^{\leftarrow}$.[3] The initial current label states that the node has not observed any sensitive information and that $n^{\leftarrow}$ is the most trusted the node can be. Dually, the clearance label states that $n^{\rightarrow}$ is an upper bound on the confidentiality of the information n is permitted to observe and that its integrity is permitted to be affected by any information. We discuss two important aspects of Flamio: Cross node communication using remote procedure calls (RPC), and distributed proof search of trust relationships.

A. Remote Procedure Calls

Nodes in Flamio communicate by remote invocation of functions on different machines, and functions are thus annotated with the node on which they should be evaluated. That is, the application $(\lambda_p^n\, x.\, e)\ e'$ denotes applying a function $\lambda_p^n\, x.\, e$ located on machine n to an argument e', and the returned value is labeled with principal p. The label serves a purpose similar to toLabeled, where labeling the value helps mitigate label creep by delaying the effect of raising the current label until the value is needed.

Consider a login function for an online banking service. The function

```
login = λ^bank_bank← u p . if checkCredentials u p
                          then return u else return ⊥
```

is evaluated on the node bank, and returns a principal labeled with the integrity of the bank, which represents an access token. The expression `login Alice "password"` evaluates to a value Alice labeled with the principal $\mathsf{bank}^{\leftarrow}$ if Alice's password is "password", and $\bot$ labeled with $\mathsf{bank}^{\leftarrow}$ otherwise. As the principal $\mathsf{Alice}^{\leftarrow}$ (i.e., the current label of node Alice) does not satisfy $\mathsf{Alice}^{\leftarrow} \sqsubseteq \mathsf{bank}^{\leftarrow}$, Alice cannot grant herself access because the semantics does not allow her to label values with the principal $\mathsf{bank}^{\leftarrow}$. However, Alice can unlabel the returned access token, as it holds that $\mathsf{bank}^{\leftarrow} \sqsubseteq \mathsf{Alice}^{\rightarrow}$ (i.e., the clearance label of node Alice). So while Alice cannot forge new access tokens, she is free to inspect whether the token grants her access to the bank.

B. Proof search

In many practical scenarios, proving trust relationships of the form $p \succcurlyeq q$ requires distributed knowledge spread across multiple nodes. In this section, we will see examples of this, as well as how distributed proof search of trust relationships in Flamio is implemented. The three most important aspects of the proof search are using delegations, managing delegations using *strategies* and forwarding trust checking to other nodes. After discussing these concepts, we apply them in the context of the decentralized banking application. We keep the discussion informal and defer precise formulations of the concepts until Section IV.

[3]We follow previous conventions [1] and omit projections of the $\bot$ principal in the remainder of the paper.

a) Delegations: A delegation is of the form $p \succcurlyeq q \mathbin{@} r$ (pronounced r *says that* p *acts for* q); we call r the label of the delegation, or say that the delegation is labeled with principal r, and we call $p \succcurlyeq q$ the body of the delegation. The terminology is the result of delegations being implemented in Flamio as pairs (p, q) labeled with r, and are subject to similar runtime checks. Thus both the confidentiality (i.e., who can observe the presence of the delegation) and the integrity (i.e., who influenced the delegation) are captured by r. In particular, a delegation can be used by a proof search on a node n when $r \sqsubseteq n^{\rightarrow}$. This requirement enforces the policy that the label of the delegation flows to the clearance of node n, meaning that n can observe the presence of a delegation only if the delegation's label flows to n's clearance label.

b) Strategies: As delegations are implemented as labeled values, using a delegation in a proof search raises the current label by the label of the delegation, similar to how LIO unlabels labeled values. This makes fine-grained control of how delegations are used important to avoid unnecessary label creep. One approach to such control could be to always unlabel all delegations whose label flows to the clearance label of the node performing the proof search. This would be correct and secure (because all and only delegations whose label flows to the node's clearance label are used in the proof search), but it may raise the current label unnecessarily (i.e., the proof search may examine more delegations than it needs to). In particular, if a delegation is examined, the current label must be raised, even if the delegation is not ultimately used in the proof. Thus if a node has a delegation labeled with a very restrictive label, then all proof searches would be tainted by that label. An alternative approach would be a proof search algorithm that unlabels just the right delegations to prove the query. However, as delegations are labeled values, the body of the delegation cannot be inspected without unlabeling the value and raising the current label. So an algorithm has to decide whether to unlabel a delegation by inspecting only the label of the delegation and not its body.

For these reasons Flamio uses *strategies*, which are lists of principals, to specify which delegations are used in a proof search, and in which order. For instance, if the delegation $p \succcurlyeq q \mathbin{@} r$ is stored on a node that evaluates the expression `withStrategy [s, r, t] (p ≽ q)` the node first searches for delegations with a label that flows to s. Assuming we cannot prove the query $p \succcurlyeq q$ using only these delegations, then delegations with a label that flows to r are used, and the delegation $p \succcurlyeq q \mathbin{@} r$ can now be used to complete the proof search. This example demonstrates how programmers can use strategies to control how Flamio performs fine-grained proof search with specialized strategies for handling delegations. The choice of which strategy to use is application specific, but a reasonable default strategy is $[\mathsf{cur}, \mathsf{clr}]$, where cur and clr are the current label and clearance labels of the node respectively. That is, proof search will first try to find a proof for a delegation query without raising the current label at all; if that is not successful, it will then try to use all available delegations.

```
transfer = λ^bank_⊥ tok u_from u_to n . do
             u <- unlabel tok
             withStrategy [bank^→ ∧ u^←_from]
               (if u ≽ u_from then transfer# u_from u_to n
                else return ())
```

Fig. 2: Secure transfer of n dollars from u_{from} to u_{to} using access token tok.

c) Forwarding: In Flamio, nodes can forward trust checks to other nodes that might have local information (i.e., delegations) about particular trust relationships. Forwarding queries is straightforward as a query $p \succcurlyeq q$ is simply a monadic expression returning a boolean, and such a query can be forwarded to other nodes in Flamio using remote procedure calls. A node n can forward trust checking to a node m only if n's current label flows to $m^{\rightarrow}$ (i.e., the clearance of m). This restriction ensures that m is allowed to learn about the information that caused n to initiate the query; a similar restriction is used to prevent the invocation of other remote procedure calls revealing information inappropriately. The following snippet demonstrates how a leak can be constructed if we did not perform the check:

```
(λ^Alice_{Alice→} _ . do h <- unlabel aliceSecret
                         if h then p ≽ q else return 0) ()
```

If aliceSecret is labeled with $\mathsf{Alice}^{\rightarrow}$, the unlabel operation will raise the current label ℓ_{cur} to $\ell_{\mathsf{cur}} \sqcup \mathsf{Alice}^{\rightarrow}$ tracking the fact that the computation context contains information at level at most $\ell_{\mathsf{cur}} \sqcup \mathsf{Alice}^{\rightarrow}$. Upon forwarding the query $p \succcurlyeq q$ to another node m, Alice reveals that aliceSecret contained a non-zero value. Checking that Alice's current label flows to $m^{\rightarrow}$ ensures that the forwarding of the query occurs only if m is allowed to learn this information.

Returning to the secure banking application, consider the implementation of transfer in Figure 2, which transfers n dollars from user u_{from} to u_{to} using access token tok.

First, the access token tok acquired by login is unlabeled, revealing the identity of the caller. Then, a check is done to ensure that the caller can act on behalf of the user from which money is being transferred. For this example the bank requires the trust relationship to be observable by the bank, and having integrity of the principal from which the money is transferred, which is reflected in the strategy specifying that only delegations at level $\mathsf{bank}^{\rightarrow} \wedge \mathsf{u}^{\leftarrow}_{\mathsf{from}}$ or below (i.e., weaker confidentiality and stronger integrity) are used. Finally, a primitive function transfer# performs the actual transfer of money, or if the trust relationship cannot be established, the function returns a unit value. Figure 3 demonstrates how Bob can transfer money from Alice if she grants Bob the appropriate trust. First, Alice adds a delegation specifying that Bob can act on behalf of Alice and that this information has the integrity of Alice. Then, Bob performs an RPC to the `login` function, which returns an access token. Finally, he transfers

```
(λ^Alice_⊥ _ . do assume Bob ≽ Alice @ Alice^←
             (λ^Bob_⊥ _ . do tok <- login Bob "password"
                             transfer tok Alice Bob 50) ()) ()
```

Fig. 3: Alice grants Bob access to perform transfers on her behalf.

50 dollars from Alice's account to his own account. Note that the use of the strategy $[\mathsf{bank}^{\rightarrow} \wedge \mathsf{u}^{\leftarrow}_{\mathsf{from}}]$ in `transfer` ensures not only that the bank uses only delegations it is allowed to observe, but also that the bank does not use a delegation provided by an inappropriate principal such as Bob, as Bob cannot add a delegation labeled with the principal $\mathsf{Alice}^{\leftarrow}$ (unless Alice first delegates trust to Bob).

IV. A Calculus for Flamio

This section introduces a formalization of the Flamio language. We first introduce the syntax and semantics of the language and afterward present the judgment for deriving trust between principals in the language. Finally, this section presents a standard type system for Flamio that guarantees basic type safety.

A. Syntax

Figure 4 shows the syntax of Flamio. The meta-variable v ranges over values, which include boolean literals, the unit value, runtime representations of principals, locations and variables, abstractions $\lambda^{n}_{\tau @ p}\, x.\, e$ (which are parameterized over the node n on which to evaluate the expression e, and the principal p representing the label on the value returned from invoking the abstraction, and τ: the type of e), and products. We will omit the type annotation on abstractions in examples. Finally, the language includes lists using nil and :: (pronounced *cons*) to express strategies.

Expressions are ranged over by the meta-variable e and include terms, applications, projections, elimination of booleans and lists, recursive functions, monadic expressions using return and $\gg\!=$, allocation, reading and writing of references. Following LIO, the language supports operations for labeling and unlabeling expressions, and the operation toLabeled for controlling label creep. The operations getLabel and getClearance returns the current label and clearance of the computational context respectively, and labelOf returns the label of a labeled value. The operation withScope e creates a new scope for delegations. The scoping for delegations have dynamic extent [18], i.e., the delegations added during the evaluation of e are visible until evaluation of e finishes. However, unlike traditional dynamic scoping, the delegation is removed once the evaluation of e terminates. This helps ensure safe and correct use of delegations.

Strategies are introduced using the withStrategy e_{strat} e operation, which introduces a new strategy e_{strat} for the evaluation of e. Similar to the scoping of delegations, strategies have dynamic extent, and the current strategy can be

$$
\begin{aligned}
v ::= {} & \mathsf{true} \mid \mathsf{false} \mid () \mid p \mid a \mid x \mid \lambda^{n}_{\tau @ p} x.\, e \mid (e, e) \mid e :: e \\
& \mid \mathsf{nil} \mid \boxed{e \mathbin{@} p \mid (e)^{\mathsf{LIO}}} \\
e ::= {} & v \mid e\ e \mid \pi_i\, e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ e\ e \mid \mathsf{fix}\ e \\
& \mid \mathsf{return}\ e \mid e \gg= e \mid \mathsf{new}\ e\ e \mid !\, e \mid e := e \mid \mathsf{label}\ e\ e \\
& \mid \mathsf{unlabel}\ e \mid \mathsf{toLabeled}\ e\ e \mid \mathsf{getLabel} \mid \mathsf{getClearance} \\
& \mid \mathsf{labelOf}\ e \mid \mathsf{withScope}\ e \mid e \succcurlyeq e \mid \mathsf{withStrategy}\ e\ e \\
& \mid \mathsf{assume}\ (e \succcurlyeq e) \mathbin{@} e \mid \mathsf{getStrategy} \mid \boxed{\mathsf{wait}(\tau)} \\
& \mid \boxed{\mathsf{toLabeled}_p\ q\ e \mid \mathsf{resetStrategy}_{\overline{p}}(e) \mid \mathsf{resetScope}_{\Delta}(e)} \\
\tau ::= {} & \mathsf{Bool} \mid \mathsf{Unit} \mid \tau \to \tau \mid [\tau] \mid \tau \times \tau \mid \mathsf{Principal} \mid \mathsf{Labeled}\,\tau \\
& \mid \mathsf{LIO}\,\tau \mid \mathsf{Ref}_n\,\tau
\end{aligned}
$$

Fig. 4: The Flamio language

$$
E[\mathsf{return}\ v] \longrightarrow E[(v)^{\mathsf{LIO}}] \qquad E[(v)^{\mathsf{LIO}} \gg= e] \longrightarrow E[e\ v]
$$

$$
E[\mathsf{fix}\ e] \longrightarrow E[e\ (\mathsf{fix}\ e)] \qquad E[\pi_i\,(e_1, e_2)] \longrightarrow E[e_i]
$$

$$
E[\mathsf{if}\ b\ \mathsf{then}\ e_{\mathsf{true}}\ \mathsf{else}\ e_{\mathsf{false}}] \longrightarrow E[e_b]
$$

$$
E[\mathsf{case}\ \mathsf{nil}\ \mathsf{of}\ e_1\ e_2] \longrightarrow E[e_1]
$$

$$
E[\mathsf{case}\ (e_{\mathsf{hd}} :: e_{\mathsf{tl}})\ \mathsf{of}\ e_1\ e_2] \longrightarrow E[e_2\ e_{\mathsf{hd}}\ e_{\mathsf{tl}}]
$$

$$
E[\mathsf{labelOf}\ (e \mathbin{@} p)] \longrightarrow E[p] \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}
$$

Fig. 5: Pure reductions for Flamio.

obtained using getStrategy. Finally, delegations can be added, and trust relationships can be queried. The shaded regions describe syntax that is not part of the surface language but rather constructs used during evaluation. We explain these constructs as we describe the evaluation rules for expressions in Section IV-B.

Finally, types are ranged over by the meta-variable τ and include standard types like the boolean type; the unit type; as well as function-, list- and product types. Non-standard types include the type of FLAM principals (Principal), the type of labeled values (Labeled τ), the type of LIO computations LIO τ, as well as location-aware reference types $\mathsf{Ref}_n\,\tau$ where location n refers to the node on which the reference is allocated.

B. Semantics

The semantics of Flamio is split into two judgments: local reduction rules, which evaluate an expression on a specific node; and global reduction rules, which perform remote procedure calls (RPCs) and returns. We first present the local reduction rules.

a) Local semantics: A structured operational semantics defines the local reduction rules with evaluation contexts [19]. We elide the definition of evaluation contexts, as this is mostly standard for a call-by-name calculus.

We further split the local reduction rules into two categories: pure reduction and monadic reduction. Pure reduction rules $e \longrightarrow e'$ reduce expressions independently of the store and of which node is evaluating the expression. They are given in Figure 5. Pure reductions include injecting terms into monadic contexts, monadically binding terms, recursive applications, projecting pairs, eliminating booleans and lists and obtaining the label of a labeled value.

The remaining local reductions all depend either on the store or on the identity of the node that evaluates the expression. We denote these as *monadic reductions*.

Before introducing the monadic small-step local reduction, we introduce the remaining necessary concepts: first, a store $\phi : \mathsf{Loc} \rightharpoonup v$ is a partial mapping from locations to terms, and we write the empty store as $\varnothing$. A local configuration is a pair $\langle \phi \mid \overline{e} \rangle$ consisting of a store ϕ and a stack of expressions $\overline{e}$. We use stacks of expressions to handle incoming remote procedure calls which "interrupts" the current computation to evaluate the RPC, and write the empty stack of expressions as $\bullet$. A global environment $\Sigma : \mathcal{N} \to \sigma$ is a mapping from names to local environments σ. A local environment $(\mathsf{lbl}, \Delta, \mathsf{strat})$ contains the current label (lbl), the set of delegations local to the node (Δ) and the current strategy of the node (strat). We use record notation for these and write $\sigma.\mathsf{lbl}$, $\sigma.\Delta$ and $\sigma.\mathsf{strat}$ respectively. We let $\varnothing$ denote the initial global environment satisfying $\varnothing(n) = (\bot^{\to} \wedge n^{\leftarrow}, \mathsf{nil}, \mathsf{nil})$. That is, the initial global environment maps each name n to an initial local environment with a current label $\bot^{\to} \wedge n^{\leftarrow}$, the empty list of delegations and the empty strategy.

The monadic small-step relation is defined by the judgment $n; \Sigma \vdash \langle \phi \mid e \rangle \longrightarrow \langle \phi' \mid e' \rangle : \sigma$ and is read as "the global environment is Σ and node n performs a single reduction and updates its local environment to σ". Figure 6 and 7 show the local monadic reduction rules. Many of the rules verify some trust relationship between principals, written $n; \Sigma \vdash p \succcurlyeq q : \ell$ and is read as "node n proves that p acts for q using delegations labeled up to ℓ". We discuss this judgment in Section IV-C.

Rule E-LIFT-PURE lifts pure reductions to monadic reductions. Only the expression at the top of the expression stack can reduce. Rule E-APP applies a function to an argument and labels the resulting value with the given principal p. Labeling the result of function applications allows us to combine local function application and RPC into the same typing rule, which simplifies the calculus and its proofs. However, it is straightforward to have different syntactic constructions for local and remote function application.

Rules E-NEW, E-READ, E-WRITE, E-LABEL and E-

UNLABEL are all equivalent to the ones presented in LIO [9], but now also take into account the possible information-flows arising via deriving trust relationships [1]. Rule E-TOLABELED-1 saves the current label $\Sigma(n).\mathsf{lbl}$, and evaluates e using the E-CTX rule. Once e has evaluated to a value E-TOLABELED-2 restores the current label and labels the value with the given label q. This presentation of toLabeled is different from the original formulation of LIO [9] and avoids interleaving small-step and big-step operations.

Rules E-ACTS-FOR-TRUE and E-ACTS-FOR-FALSE query the trust relationship between two given principals using the trust judgment, which we explain in Section IV-C. The result depends on which delegations are in scope, and on the current strategy. A new scope for delegations is created using E-WITH-SCOPE which evaluates an expression e in a new scope, and once e has reduced to a value, rule E-RESET-SCOPE eliminates the scope. Similarly, rule E-WITH-STRATEGY introduces a new strategy and evaluates an expression in the scope of the new strategy. Once the expression has reduced to a value, rule E-RESET-STRATEGY resets the strategy back to its previous value. Finally, E-ASSUME adds a new delegation. The rule uses the operator ∇ defined in Section II to ensure that the computational context has sufficient integrity to delegate trust on behalf of q.

Figure 8 demonstrates the usefulness of delegation scoping with dynamic extent: On line 2, Alice invokes a function on Bob's node that, on line 5, grants another function (given as an argument) the authority to read Bob's confidential information. In addition, on line 9, Bob enforces the policy that the function is only called once. Due to the use of withScope on line 4, the additional authority is given only to the function passed as an argument and only when it is invoked at that point. Any other function cannot read Bob's confidential information.

b) Global semantics: A global configuration is a triple $(\!|\overline{n}, \Sigma, S|\!)$ consisting of a stack of nodes $\overline{n} \in \mathcal{N}^*$ representing the RPC call-stack, a global environment Σ, and a mapping S from nodes to local configurations. Figure 9 presents the reduction rules for global configurations. Rule G-STEP-LOCAL lifts a local reduction to a global reduction, and rules G-STEP-APP and G-STEP-RET handle remote procedure calls and returns respectively. When node n sends an RPC to node m, we call n the source node and m the target node. The global reduction rule is written as $(\!|\overline{n}, \Sigma, S|\!) \Longrightarrow (\!|\overline{n}', \Sigma', S'|\!)$ and can be read as "the first node in $\overline{n}$ updates the environment Σ to Σ', updates the local configurations S to S', and modifies the call-stack to $\overline{n}'$". We write the reflexive, transitive closure of $\Longrightarrow$ as $\Longrightarrow^*$.

We now explain how to express RPC. Rule G-STEP-APP transfers control to the target node, and the computation is wrapped in a toLabeled construct at the top of the execution stack on the target node to prevent the evaluation of the expression from raising the current label. The rule ensures that m's new current label $\ell_n \sqcup \ell_m$, flows to m's clearance label $m^{\rightarrow}$. This check ensures that the clearance always upper bounds the current label. Finally, the evaluation on the source node is suspended using expression wait(τ), which waits for a value of type τ. Second, Rule G-STEP-RET returns control to a suspended source node when the top of the execution stack on the target node has reduced to a term.

Note that although computation is distributed, the semantics is deterministic: only one expression is reducible at any point. Determinism excludes internal timing leaks and other attacks usually found in concurrent systems [20], [21], while still allowing multiple nodes to share computation.

E-LIFT-PURE
$$\frac{e \longrightarrow e'}{n; \Sigma \vdash \langle \phi \mid e \rangle \longrightarrow \langle \phi \mid e' \rangle : \Sigma(n)}$$

E-CTX
$$\frac{n; \Sigma \vdash \langle \phi \mid e \rangle \longrightarrow \langle \phi' \mid e' \rangle : \sigma}{n; \Sigma \vdash \langle \phi \mid E[e] \rangle \longrightarrow \langle \phi' \mid E[e'] \rangle : \sigma}$$

E-GET-LABEL
$$\frac{\ell = \Sigma(n).\mathsf{lbl}}{n; \Sigma \vdash \langle \phi \mid \mathsf{getLabel} \rangle \longrightarrow \langle \phi \mid \mathsf{return}\ \ell \rangle : \Sigma(n)}$$

E-GET-CLEARANCE
$$n; \Sigma \vdash \langle \phi \mid \mathsf{getClearance} \rangle \longrightarrow \langle \phi \mid \mathsf{return}\ n^{\rightarrow} \rangle : \Sigma(n)$$

E-APP
$$\frac{\begin{array}{c} n; \Sigma \vdash \Sigma(n).\mathsf{lbl} \sqsubseteq p \sqsubseteq n^{\rightarrow} : \ell \\ e' = e_1[e_2/x] \gg\!= \lambda^n_\tau\, x.\, x \mathbin{@} p \\ n; \Sigma \vdash \Sigma(n).\mathsf{lbl} \sqcup \ell \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n)\,[\mathsf{lbl} \mapsto \Sigma(n).\mathsf{lbl} \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid (\lambda^n_{\tau @ p}\, x.\, e_1)\ e_2 \rangle \longrightarrow \langle \phi \mid e' \rangle : \sigma}$$

E-NEW
$$\frac{\begin{array}{c} a \notin \mathsf{dom}(\phi) \qquad n; \Sigma \vdash \Sigma(n).\mathsf{lbl} \sqsubseteq p \sqsubseteq n^{\rightarrow} : \ell \\ n; \Sigma \vdash \Sigma(n).\mathsf{lbl} \sqcup \ell \sqsubseteq n^{\rightarrow} \qquad \phi' = \phi\,[a \mapsto e \mathbin{@} p] \\ \sigma = \Sigma(n)\,[\mathsf{lbl} \mapsto \Sigma(n).\mathsf{lbl} \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid \mathsf{new}\ p\ e \rangle \longrightarrow \langle \phi' \mid \mathsf{return}\ a \rangle : \sigma}$$

E-READ
$$\frac{\begin{array}{c} \phi(a) = e \mathbin{@} p \qquad n; \Sigma \vdash \Sigma(n).\mathsf{lbl} \sqcup p \sqsubseteq n^{\rightarrow} : \ell \\ n; \Sigma \vdash \Sigma(n).\mathsf{lbl} \sqcup p \sqcup \ell \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n)\,[\mathsf{lbl} \mapsto \Sigma(n).\mathsf{lbl} \sqcup p \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid\ !\,a \rangle \longrightarrow \langle \phi \mid \mathsf{return}\ e \rangle : \sigma}$$

E-WRITE
$$\frac{\begin{array}{c} \phi(a) = e \mathbin{@} p \qquad n; \Sigma \vdash \Sigma(n).\mathsf{lbl} \sqsubseteq p \sqsubseteq n^{\rightarrow} : \ell \\ n; \Sigma \vdash \Sigma(n).\mathsf{lbl} \sqcup \ell \sqsubseteq n^{\rightarrow} \qquad \phi' = \phi\,[a \mapsto e' \mathbin{@} p] \\ \sigma = \Sigma(n)\,[\mathsf{lbl} \mapsto \Sigma(n).\mathsf{lbl} \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid a := e' \rangle \longrightarrow \langle \phi' \mid \mathsf{return}\ () \rangle : \sigma}$$

Fig. 6: Monadic reductions for Flamio.

E-LABEL
$$\frac{\begin{array}{c} n;\Sigma \vdash \Sigma(n).\mathsf{lbl} \sqsubseteq p \sqsubseteq n^{\rightarrow} : \ell \\ n;\Sigma \vdash \Sigma(n).\mathsf{lbl} \sqcup \ell \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n)\,[\Sigma(n).\mathsf{lbl} \mapsto \Sigma(n).\mathsf{lbl} \sqcup \ell] \end{array}}{n;\Sigma \vdash \langle \phi \mid \mathsf{label}\ p\ e \rangle \longrightarrow \langle \phi \mid \mathsf{return}\ (e \mathbin{@} p) \rangle : \sigma}$$

E-UNLABEL
$$\frac{\begin{array}{c} n;\Sigma \vdash \Sigma(n).\mathsf{lbl} \sqcup p \sqsubseteq n^{\rightarrow} : \ell \\ n;\Sigma \vdash \Sigma(n).\mathsf{lbl} \sqcup p \sqcup \ell \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n)\,[\mathsf{lbl} \mapsto \Sigma(n).\mathsf{lbl} \sqcup p \sqcup \ell] \end{array}}{n;\Sigma \vdash \langle \phi \mid \mathsf{unlabel}\ (e \mathbin{@} p) \rangle \longrightarrow \langle \phi \mid \mathsf{return}\ e \rangle : \sigma}$$

E-TOLABELED-1
$$\frac{q = \Sigma(n).\mathsf{lbl} \qquad e' = \mathsf{toLabeled}_q\ p\ e}{n;\Sigma \vdash \langle \phi \mid \mathsf{toLabeled}\ p\ e \rangle \longrightarrow \langle \phi \mid e' \rangle : \Sigma(n)}$$

E-TOLABELED-2
$$\frac{\sigma = \Sigma(n)\,[\mathsf{lbl} \mapsto p]}{n;\Sigma \vdash \langle \phi \mid \mathsf{toLabeled}_p\ q\ v \rangle \longrightarrow \langle \phi \mid \mathsf{label}\ q\ v \rangle : \sigma}$$

E-ACTS-FOR-TRUE
$$\frac{\begin{array}{c} n;\Sigma \vdash p \succcurlyeq q : \ell \qquad n;\Sigma \vdash \Sigma(n).\mathsf{lbl} \sqcup \ell \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n)\,[\mathsf{lbl} \mapsto \Sigma(n).\mathsf{lbl} \sqcup \ell] \end{array}}{n;\Sigma \vdash \langle \phi \mid p \succcurlyeq q \rangle \longrightarrow \langle \phi \mid \mathsf{return\ true} \rangle : \sigma}$$

E-ACTS-FOR-FALSE
$$\frac{\begin{array}{c} n;\Sigma \vdash p \succcurlyeq q : \mathsf{fail} \qquad \ell = n^{\rightarrow} \sqcap \bigsqcup_{\mathsf{s} \in \Sigma(n).\mathsf{strat}} \mathsf{s} \\ \sigma = \Sigma(n)\,[\mathsf{lbl} \mapsto \Sigma(n).\mathsf{lbl} \sqcup \ell] \end{array}}{n;\Sigma \vdash \langle \phi \mid p \succcurlyeq q \rangle \longrightarrow \langle \phi \mid \mathsf{return\ false} \rangle : \sigma}$$

E-WITH-SCOPE
$$\frac{e' = \mathsf{resetScope}_{\Sigma(n).\Delta}(e)}{n;\Sigma \vdash \langle \phi \mid \mathsf{withScope}\ e \rangle \longrightarrow \langle \phi \mid e' \rangle : \Sigma(n)}$$

E-RESET-SCOPE
$$\frac{\sigma = \Sigma(n)\,[\Delta \mapsto \Delta']}{n;\Sigma \vdash \langle \phi \mid \mathsf{resetScope}_{\Delta'}(v) \rangle \longrightarrow \langle \phi \mid v \rangle : \sigma}$$

E-WITH-STRATEGY
$$\frac{e' = \mathsf{resetStrategy}_{\Sigma.\mathsf{strat}}(e) \qquad \sigma = \Sigma(n)\,[\mathsf{strat} \mapsto \vec{p}]}{n;\Sigma \vdash \langle \phi \mid \mathsf{withStrategy}\ \vec{p}\ e \rangle \longrightarrow \langle \phi \mid e' \rangle : \sigma}$$

E-RESET-STRATEGY
$$\frac{\sigma = \Sigma(n)\,[\mathsf{strat} \mapsto \vec{p}]}{n;\Sigma \vdash \langle \phi \mid \mathsf{resetStrategy}_{\vec{p}}(v) \rangle \longrightarrow \langle \phi \mid v \rangle : \sigma}$$

E-ASSUME
$$\frac{\begin{array}{c} n;\Sigma \vdash \Sigma(n).\mathsf{lbl} \sqsubseteq r : \ell_1 \qquad n;\Sigma \vdash \Sigma(n).\mathsf{lbl} \succcurlyeq \nabla(q) : \ell_2 \\ n;\Sigma \vdash \Sigma(n).\mathsf{lbl} \sqcup \ell_1 \sqcup \ell_2 \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n)\,[\Delta \mapsto (p,q) \mathbin{@} r :: \Sigma.\Delta),\mathsf{lbl} \mapsto \Sigma(n).\mathsf{lbl} \sqcup \ell_1 \sqcup \ell_2] \end{array}}{n;\Sigma \vdash \langle \phi \mid \mathsf{assume}\ (p \succcurlyeq q) \mathbin{@} r \rangle \longrightarrow \langle \phi \mid \mathsf{return}\ () \rangle : \sigma}$$

Fig. 7: Monadic reductions for Flamio (cont).

```
(λ^Alice_⊥ _ . [...]
  let g = (λ^Bob_⊥ f . do
                bref := new Bob^← true
                withScope (do
                  assume Alice^→ ≽ Bob^→ @ Bob^←
                  return (λ^Alice_Alice^← x . do
                            b <- !bref
                            bref := false
                            if b then f x else false)))
                              aliceCode
  in g bobSecret) ()
```

Fig. 8: Bob grants a function, supplied by Alice, authority to read Bob's confidential information once.

G-STEP-LOCAL
$$\frac{n;\Sigma \vdash S(n) \longrightarrow s : \sigma}{(\!| n \cdot \overline{n}, \Sigma, S |\!) \Longrightarrow (\!| n \cdot \overline{n}, \Sigma\,[n \mapsto \sigma]\,, S\,[n \mapsto s] |\!)}$$

G-STEP-APP
$$\frac{\begin{array}{c} \ell_n = \Sigma(n).\mathsf{lbl} \qquad \ell_m = \Sigma(m).\mathsf{lbl} \qquad S(m) = \langle \phi_m \mid \overline{e_m} \rangle \\ m;\Sigma \vdash \ell_n \sqcup \ell_m \sqsubseteq m^{\rightarrow} \qquad S(n) = \langle \phi_n \mid E[(\lambda^{m}_{\tau @ p}\, x.\, e_n)\ e'_n] \rangle \\ s'_n = \langle \phi_n \mid E[\mathsf{wait}(\tau)] \rangle \qquad \sigma'_m = \Sigma(m)\,[\mathsf{lbl} \mapsto \ell_n \sqcup \ell_m] \\ s'_m = \langle \phi_m \mid (\mathsf{toLabeled}_{\ell_m}\ p\ (e_n[e'_n/x]));\overline{e_m} \rangle \end{array}}{(\!| n \cdot \overline{n}, \Sigma, S |\!) \Longrightarrow (\!| m \cdot n \cdot \overline{n}, \Sigma\,[m \mapsto \sigma'_m]\,, S\,[n \mapsto s'_n, m \mapsto s'_m] |\!)}$$

G-STEP-RET
$$\frac{\begin{array}{c} S(n) = \langle \phi_n \mid E[\mathsf{wait}(\tau)] \rangle \qquad S(m) = \langle \phi_m \mid v;\overline{e_m} \rangle \\ s'_n = \langle \phi_n \mid E[v] \rangle \qquad s'_m = \langle \phi_m \mid \overline{e_m} \rangle \end{array}}{(\!| m \cdot n \cdot \overline{n}, \Sigma, S |\!) \Longrightarrow (\!| n \cdot \overline{n}, \Sigma, S\,[n \mapsto s'_n, m \mapsto s'_m] |\!)}$$

Fig. 9: Semantics of global steps

C. Deriving trust relationship in Flamio

Flamio allows, in the style of FLAM, the trust relationship between principals to be changed and queried dynamically throughout the evaluation of a program. We show how the ideas from FLAM on how to provide guarantees of confidentiality and integrity can be incorporated into the floating-label model of LIO.

Rule ACTS-FOR in Figure 10 formalizes the top-level judg-

ACTS-FOR-CONS-1
$$\frac{\mathcal{H};n;\Sigma \vdash_{\mathsf{s} \sqcap n^{\rightarrow}} p \succcurlyeq q : \ell}{\mathcal{H};n;\Sigma \vdash_{\mathsf{s::ss}} p \succcurlyeq q : \ell}$$

ACTS-FOR-CONS-2
$$\frac{\begin{array}{c} \mathcal{H};n;\Sigma \vdash_{\mathsf{s} \sqcap n^{\rightarrow}} p \succcurlyeq q : \mathsf{fail} \\ \mathcal{H};n;\Sigma \vdash_{\mathsf{ss}} p \succcurlyeq q : \ell^{?} \end{array}}{\mathcal{H};n;\Sigma \vdash_{\mathsf{s::ss}} p \succcurlyeq q : \ell^{?}}$$

ACTS-FOR-NIL
$$\frac{}{C \vdash_{\mathsf{nil}} p \succcurlyeq q : \mathsf{fail}}$$

ACTS-FOR
$$\frac{\begin{array}{c} \mathsf{ss} = \Sigma(n).\mathsf{strat} \\ \mathcal{H};n;\Sigma \vdash_{\mathsf{ss}} p \succcurlyeq q : \ell \end{array}}{\mathcal{H};n;\Sigma \vdash p \succcurlyeq q : \ell}$$

Fig. 10: Top-level judgment for proving authorization queries in Flamio. The meta-variable C abbreviates $\mathcal{H};n;\Sigma$.

$$\textsc{Bot}\quad C \vdash_{\mathsf{s}} p \succcurlyeq \bot : \bot^{\sqsubseteq} \qquad \textsc{Top}\quad C \vdash_{\mathsf{s}} \top \succcurlyeq p : \bot^{\sqsubseteq} \qquad \textsc{Refl}\quad C \vdash_{\mathsf{s}} p \succcurlyeq p : \bot^{\sqsubseteq}$$

$$\textsc{Assump}\ \frac{(p \succcurlyeq q) \in \mathcal{H}}{C \vdash_{\mathsf{s}} p \succcurlyeq q : \bot^{\sqsubseteq}} \qquad \textsc{Proj}\ \frac{C \vdash_{\mathsf{s}} p \succcurlyeq q : \ell}{C \vdash_{\mathsf{s}} p^{\pi} \succcurlyeq q^{\pi} : \ell} \qquad \textsc{ProjR}\quad C \vdash_{\mathsf{s}} p \succcurlyeq p^{\pi} : \bot^{\sqsubseteq}$$

$$\textsc{Own-1}\ \frac{\begin{array}{c} C \vdash_{\mathsf{s}} o \succcurlyeq o' : \ell_1 \\ C \vdash_{\mathsf{s}} p \succcurlyeq p' : \ell_2 \end{array}}{C \vdash_{\mathsf{s}} o{:}p \succcurlyeq o'{:}p' : \ell_1 \sqcup \ell_2} \qquad \textsc{Own-2}\ \frac{\begin{array}{c} C \vdash_{\mathsf{s}} o \succcurlyeq o' : \ell_1 \\ C \vdash_{\mathsf{s}} p \succcurlyeq o'{:}p' : \ell_2 \end{array}}{C \vdash_{\mathsf{s}} o{:}p \succcurlyeq o'{:}p' : \ell_1 \sqcup \ell_2}$$

$$\textsc{Conj-L}\ \frac{\begin{array}{c} j \in \{1,2\} \\ C \vdash_{\mathsf{s}} p_j \succcurlyeq p : \ell \end{array}}{C \vdash_{\mathsf{s}} p_1 \wedge p_2 \succcurlyeq p : \ell} \qquad \textsc{Conj-R}\ \frac{\begin{array}{c} C \vdash_{\mathsf{s}} p \succcurlyeq p_1 : \ell_1 \\ C \vdash_{\mathsf{s}} p \succcurlyeq p_2 : \ell_2 \end{array}}{C \vdash_{\mathsf{s}} p \succcurlyeq p_1 \wedge p_2 : \ell_1 \sqcup \ell_2}$$

$$\textsc{Disj-L}\ \frac{\begin{array}{c} C \vdash_{\mathsf{s}} p_1 \succcurlyeq p : \ell_1 \\ C \vdash_{\mathsf{s}} p_2 \succcurlyeq p : \ell_2 \end{array}}{C \vdash_{\mathsf{s}} p_1 \vee p_2 \succcurlyeq p : \ell_1 \sqcup \ell_2} \qquad \textsc{Disj-R}\ \frac{\begin{array}{c} j \in \{1,2\} \\ C \vdash_{\mathsf{s}} p \succcurlyeq p_j : \ell \end{array}}{C \vdash_{\mathsf{s}} p \succcurlyeq p_1 \vee p_2 : \ell}$$

$$\textsc{Trans}\ \frac{\begin{array}{c} C \vdash_{\mathsf{s}} p \succcurlyeq q : \ell_1 \\ C \vdash_{\mathsf{s}} q \succcurlyeq r : \ell_2 \end{array}}{C \vdash_{\mathsf{s}} p \succcurlyeq r : \ell_1 \sqcup \ell_2} \qquad \textsc{Del}\ \frac{\begin{array}{c} p \succcurlyeq q \mathbin{@} \ell \in \Sigma(n).\Delta \\ \mathcal{H}, p \succcurlyeq q; n; \Sigma \vdash_{\mathsf{s}} \ell \sqsubseteq \mathsf{s} : \ell' \\ \mathcal{H}, p \succcurlyeq q; n; \Sigma \vdash_{\mathsf{s}} \ell' \sqsubseteq \mathsf{s} : \bot^{\sqsubseteq} \end{array}}{\mathcal{H}; n; \Sigma \vdash_{\mathsf{s}} p \succcurlyeq q : \ell}$$

$$\textsc{Fwd}\ \frac{\begin{array}{cc} \ell_n = \Sigma(n).\mathsf{lbl} & \sigma_m = \Sigma(m)\,[\mathsf{lbl} \mapsto \ell_n \sqcup \ell_m] \\ \ell_m = \Sigma(m).\mathsf{lbl} & \mathcal{H}; n; \Sigma \vdash_{\mathsf{s}} \ell_n \sqcup \ell_m \sqsubseteq m^{\rightarrow} : \ell_1 \\ \multicolumn{2}{c}{\mathcal{H}; m; \Sigma\,[m \mapsto \sigma_m] \vdash_{\mathsf{s}} p \succcurlyeq q : \ell_2} \end{array}}{\mathcal{H}; n; \Sigma \vdash_{\mathsf{s}} p \succcurlyeq q : \ell_1 \sqcup \ell_2}$$

Fig. 11: Acts for judgment of Flamio. The meta-variable C abbreviates $\mathcal{H}; n; \Sigma$.

ment for deriving trust, which iterates through the strategy principals in the current strategy and attempts to prove the trust relationship at each strategy principal in the list. We write $\ell^?$ to mean either a label ℓ, or a failure value fail meaning that the trust relationship could not be established. Rule ACTS-FOR-CONS-1 states that, if we can prove the trust relationship limiting our use of delegations to those bounded above by strategy principal s then the query succeeds. Rule ACTS-FOR-CONS-2 states that, if we cannot prove the trust relationship using delegations bounded above by strategy principal s ($C \vdash_{\mathsf{s} \sqcap n^{\rightarrow}} p \succcurlyeq q : \mathsf{fail}$), then the search continues with the tail ss of the strategy. In both rules the strategy principal s is attenuated with the clearance of the node ($n^{\rightarrow}$) to ensure that the node does not attempt to use delegations above its clearance label. Finally, in Rule ACTS-FOR-NIL, if we have tried all strategy principals and have not proved the trust relationship, when the proof search fails.

Figure 11 shows how the judgment $\mathcal{H}; n; \Sigma \vdash_{\mathsf{s}} p \succcurlyeq q : \ell$ derives trust relationships between principals using the strategy principal s. The judgment means that node n proves that that q trusts p in the global environment Σ using delegations with labels that are upper bounded by ℓ, and assuming that r acts for s for all $(r, s) \in \mathcal{H}$. We call $\mathcal{H}$ the assumptions of the query, and write $r \succcurlyeq s$ for the assumption (r, s). We write $n; \Sigma \vdash_{\mathsf{s}} p \succcurlyeq q : \ell$ to mean $\varnothing; n; \Sigma \vdash_{\mathsf{s}} p \succcurlyeq q : \ell$. That is, the judgment holds with no assumptions. We also write $\mathcal{H}; n; \Sigma \vdash_{\mathsf{s}} p \succcurlyeq q$ to mean $\exists \ell \ .\ \mathcal{H}; n; \Sigma \vdash_{\mathsf{s}} p \succcurlyeq q : \ell$. Finally, guided by (1) we write $\mathcal{H}; n; \Sigma \vdash_{\mathsf{s}} p \sqsubseteq q : \ell$ to mean $\mathcal{H}; n; \Sigma \vdash_{\mathsf{s}} q^{\rightarrow} \wedge p^{\leftarrow} \succcurlyeq p^{\rightarrow} \wedge q^{\leftarrow} : \ell$ and use similar abbreviations as above.

Many rules translate directly from FLAM, except for using delegations and querying remote nodes for trust relationships. This discrepancy is because the upper bound on the label of usable delegations in FLAM is given as "input" to the judgment, while in Flamio the upper bound is an "output" of the judgment.[4] This relieves the programmer from having to manually annotate trust queries (and operations that perform trust queries) with explicit upper bounds for delegation labels.

Rule ASSUMP states that any assumptions can be used to derive trust without raising the current label any further. This use of assumptions is an instance of a checked endorsement [23], [24], and is discussed later in this section. Rules OWN-1 and OWN-2 derives trust between ownership projections. First, OWN-1 shows that trust between principals imply trust between owned principals, and OWN-2 states that, if an ownership projection $o'{:}p'$ trusts a principal p and owner o' trusts o then another ownership principal $o{:}p$ also trusts $o'{:}p'$. Rule FWD expresses how a node n can query another node m for a trust relationship, but only if n allows the information that caused n to contact m to be learned by m (i.e., $\ell_n \sqsubseteq m^{\rightarrow}$, which is implied by the premise $\ell_n \sqcup \ell_m \sqsubseteq m^{\rightarrow}$). Furthermore, when forwarding a query, node m must raise its current label to $\ell_n \sqcup \ell_m$ to propagate the sensitivity of the computational context of node n. Finally, when node n forwards the query to m, the strategy principal used by n is also used by m as otherwise, local reasoning about trust relationship queries would be impossible without knowing the strategies of every node in the system.

Finally, rule DEL expresses how delegations are used to derive trust. First, a delegation that proves the trust relationship must be present ($p \succcurlyeq q \mathbin{@} \ell \in \Sigma(n).\Delta$), then, the label on the delegation must flow to the strategy principal s that is currently bounding how much the current label can be raised ($\mathcal{H}, p \succcurlyeq q; n; \Sigma \vdash \ell \sqsubseteq s : \ell'$). However, the fact that ℓ is bounded by s could also be used to leak information, so ℓ' should also be bounded by s. This checking could potentially continue ad infinitum, so we apply a pragmatic approach and

[4]That the upper bound on the delegation labels is an "input" to the judgment can be seen in the $F\lambda$ calculus [22] where delegation labels appear in the surface syntax of expressions.

require this check not to use any delegations labeled higher than $\perp^{\sqsubseteq}$. Section VI motivates this decision, which shows several interesting examples that can all be implemented using this simplified checking mechanism.

When checking if a delegation can be used in the rule DEL, the assumptions $\mathcal{H}$ is extended to include the trust relationship that is being checked. This usage of assumptions is a form of checked endorsement which was also noted to be a useful extension to the Jif programming language [23]. To see the effect of this style of reasoning, consider the query $n; \Sigma \vdash p \sqsubseteq q : \ell$ where $\Sigma(n) = (n^{\leftarrow}, \Delta, [q])$ and $\Delta = \{(q^{\rightarrow} \wedge p^{\leftarrow} \succcurlyeq p^{\rightarrow} \wedge q^{\leftarrow}) \,@\, p\}$. This query is equivalent to $n; \Sigma \vdash q^{\rightarrow} \wedge p^{\leftarrow} \succcurlyeq p^{\rightarrow} \wedge q^{\leftarrow} : \ell$, so applying DEL the goal reduces to proving $p \sqsubseteq q; n; \Sigma \vdash p \sqsubseteq q : \ell'$ for some ℓ', which (ignoring the assumption) is the exact same query we started out with! However, we now have the assumption $p \sqsubseteq q$, and the goal follows by applying ASSUMP. If no assumption was added when checking that the label on the delegation $p \sqsubseteq q @ p$ flows to the strategy principal q, this trust relationship could not be proven in any finite derivation. While this situation might appear artificial, Section VI presents a use case where this problem arises naturally.

For technical reasons we assume that the delegation $p \succcurlyeq q \,@\, \ell$ in $\Sigma(n).\Delta$ is picked in some deterministic way (e.g., it is the *first* delegation in $\Sigma(n).\Delta$ that satisfies the other premises).

We end this section with an example of a query that morally should hold, but which cannot be justified using our pragmatic trust judgment. Given the following four delegation sets:

$$\begin{aligned}
\Delta_1 &= \{a \succcurlyeq b \,@\, \perp^{\sqsubseteq}\} \\
\Delta_2 &= \{a \succcurlyeq b \,@\, c, c \sqsubseteq \ell \,@\, \perp^{\sqsubseteq}\} \\
\Delta_3 &= \{a \succcurlyeq b \,@\, c, c \sqsubseteq \ell \,@\, d, d \sqsubseteq \ell \,@\, \perp^{\sqsubseteq}\} \\
\Delta_4 &= \{a \succcurlyeq b \,@\, c, c \sqsubseteq \ell \,@\, d, d \sqsubseteq \ell \,@\, e, e \sqsubseteq \ell \,@\, \perp^{\sqsubseteq}\}
\end{aligned}$$

and the environments $\Sigma_i(n) = (n^{\leftarrow}, \Delta_i, [\ell])$, the query $n; \Sigma \vdash a \succcurlyeq b : \ell$ holds for $i \in \{1, 2, 3\}$, but does not hold for $i = 4$. To see why, consider proving the query using DEL. We must prove that

$$a \succcurlyeq b \,@\, c \in \Delta_4 \tag{4}$$
$$\{a \succcurlyeq b\}; n; \Sigma_4 \vdash_{\ell} c \sqsubseteq \ell : d \tag{5}$$
$$\{a \succcurlyeq b\}; n; \Sigma_4 \vdash_{\ell} d \sqsubseteq \ell : \perp^{\sqsubseteq} \tag{6}$$

Condition (4) holds by definition of Δ_4, and (5) holds by applying DEL. But (6) does not hold: we can only show $\{a \succcurlyeq b\}; n; \Sigma_4 \vdash d \sqsubseteq \ell : e$ and $\{a \succcurlyeq b\}; n; \Sigma_4 \vdash e \sqsubseteq \ell : \perp^{\sqsubseteq}$, but this does not imply (6). That is, Flamio cannot prove that the label on the information "the label on the required delegation flows to the current strategy principal" is $\perp^{\sqsubseteq}$. We have not found a realistic scenario where this presents a problem, and we leave lifting this restriction as future work.

D. A type system for Flamio.

Since Flamio controls information-flows via dynamic checks, the type system for Flamio is straightforward. We write $n; \Gamma \vdash e : \tau$ when expression e can be given type τ in a global type environment $\Gamma : \mathcal{N} \rightarrow (\mathsf{Var} \uplus \mathsf{Loc} \rightharpoonup \tau)$ on node n. Note that the typing environment maps both variables and locations to types. Figure 12 shows excerpts of this judgment. Rule T-ABS states that a function has a function type and that, the typing environment for node m is used when checking the type of the body, where m is the target node. Rule T-LAB states that labeled expressions have labeled types. Rules T-RETURN and T-BIND are standard typing rules for monadic expressions. Rule T-TO-LABELED states that an expression $\mathsf{toLabeled}\; e_1\; e_2$ is well-typed when e_1 is a principal, and that the expression has type $\mathsf{Labeled}\,\tau$ if e_2 has type τ. We say a location a belongs to node n if $a \in \mathsf{dom}(\Gamma_n)$. Rule T-NEW states that when a reference is allocated on a node n the type of the location returned belongs to n, and T-READ states that a reference can only be read on a node to which the location belongs. Finally, T-WAIT states that the type attached to a waiting expression is the type of the expression.

Given a global typing environment Γ we write $n; \Gamma \vdash \phi$ if, for all a such that $\phi(a) = e \,@\, p$ and $\Gamma_n(a) = \mathsf{Ref}_n\, \tau$ it holds that $n; \Gamma \vdash e : \tau$. We write $n; \Gamma \vdash \langle \phi \mid \overline{e} \rangle : \overline{\tau}$ if $\overline{e} = e_1 \cdots e_n$ and $n; \Gamma \vdash \phi$ and $n; \Gamma \vdash e_i : \tau_i$ for $i = 1, \ldots, n$ and $\overline{\tau} = \tau_1 \ldots \tau_n$. We lift this definition to global configurations and write $\Gamma \vdash_m (\!| n, \Sigma, S |\!) : \overline{\tau}$ if for all $n' \in \mathcal{N}$ there exists a type $\overline{\tau}'$ such that $n'; \Gamma \vdash S_{n'} : \overline{\tau}'$, and furthermore, when $n' = m$ we have $\overline{\tau}' = \overline{\tau}$.

V. SECURITY GUARANTEES

In this section, we define the attacker model and show the security guarantees given by Flamio. Specifically, we show that Flamio executions satisfy termination-insensitive noninterference (TINI) [25]. Formally, an attacker is some principal $\mathcal{A}$. Note that this principal might be a conjunction of named principals $n_1 \wedge \cdots \wedge n_k$ representing a set of k colluding principals. In the sections that follow, we denote a $\mathcal{N}$-indexed set of memories as a function $\Phi : \mathcal{N} \rightarrow (\mathsf{Loc} \rightharpoonup v)$.

A. Trace semantics

We express the attacker model in terms of a trace semantics, in which certain operations in the language emit *events* which may or may not be observable by $\mathcal{A}$. The grammar for events is given in Figure 13. A non-empty event (α, Σ, n) contains the type of the event α, the current environment when the event was emitted Σ, and the node n that emitted the event. The types of events include: write events $\mathsf{write}(a, e)$, emitted when a node writes an expression e to reference a; allocation events $\mathsf{new}(a, e)$, emitted when a node allocates a new reference a, initialized to e; call events $\mathsf{call}(e, m)$, emitted when a node invokes an RPC e on node m; and return events $\mathsf{ret}(v, m)$, emitted when a node finishes an RPC, returning value v to the caller node m. In addition, we have release events release, explained below. Finally, the empty event ε is emitted by

$$\textsc{T-Principal}\quad \frac{}{n;\Gamma \vdash p : \mathsf{Principal}} \qquad \textsc{T-Var}\quad \frac{\Gamma_n(x) = \tau}{n;\Gamma \vdash x : \tau} \qquad \textsc{T-Ref}\quad \frac{\Gamma_n(a) = \mathsf{Ref}_n\,\tau}{n;\Gamma \vdash a : \mathsf{Ref}_n\,\tau}$$

$$\textsc{T-Abs}\quad \frac{m;\Gamma, x : \tau_1 \vdash e : \mathsf{LIO}\,\tau_2 \qquad \tau = \tau_1 \to \mathsf{LIO}\,(\mathsf{Labeled}\,\tau_2)}{n;\Gamma \vdash \lambda^m_{\tau_2 @ p}\, x.\, e : \tau} \qquad \textsc{T-Lab}\quad \frac{n;\Gamma \vdash e_2 : \tau \qquad n;\Gamma \vdash e_1 : \mathsf{Principal}}{n;\Gamma \vdash e_2 \mathbin{@} e_1 : \mathsf{Labeled}\,\tau}$$

$$\textsc{T-Return}\quad \frac{n;\Gamma \vdash e : \tau}{n;\Gamma \vdash \mathsf{return}\ e : \mathsf{LIO}\,\tau} \qquad \textsc{T-Bind}\quad \frac{n;\Gamma \vdash e_1 : \mathsf{LIO}\,\tau_1 \qquad n;\Gamma \vdash e_2 : \tau_1 \to \mathsf{LIO}\,\tau_2}{n;\Gamma \vdash e_1 \gg\!= e_2 : \mathsf{LIO}\,\tau_2}$$

$$\textsc{T-To-Labeled}\quad \frac{n;\Gamma \vdash e_1 : \mathsf{Principal} \qquad n;\Gamma \vdash e_2 : \mathsf{LIO}\,\tau \qquad \tau' = \mathsf{LIO}\,(\mathsf{Labeled}\,\tau)}{n;\Gamma \vdash \mathsf{toLabeled}\ e_1\ e_2 : \tau'} \qquad \textsc{T-Read}\quad \frac{n;\Gamma \vdash e : \mathsf{Ref}_n\,\tau}{n;\Gamma \vdash\ !e : \mathsf{LIO}\,\tau}$$

$$\textsc{T-New}\quad \frac{n;\Gamma \vdash e_1 : \mathsf{Principal} \qquad n;\Gamma \vdash e_2 : \tau \qquad \tau' = \mathsf{LIO}\,(\mathsf{Ref}_n\,\tau)}{n;\Gamma \vdash \mathsf{new}\ e_1\ e_2 : \tau'} \qquad \textsc{T-Wait}\quad \frac{\tau' = \mathsf{LIO}\,(\mathsf{Labeled}\,\tau)}{n;\Gamma \vdash \mathsf{wait}(\tau) : \tau'}$$

Fig. 12: Typing judgment for Flamio.

$$\begin{aligned} ev &::= (\alpha, \Sigma, n) \mid \varepsilon \\ \alpha &::= \mathsf{write}(a, e) \mid \mathsf{new}(a, e) \mid \mathsf{call}(e, n) \\ &\quad\mid \mathsf{ret}(v, n) \mid \mathsf{release}(p, q, r) \end{aligned}$$

Fig. 13: The syntax of events.

operations that do not emit some other event. We call a sequence of events a *trace*. We write the concatenation of traces as $t_1 \cdot t_2$ and we write the empty trace as ε.

a) Release events: Flamio is a very expressive language that permits downgrading, i.e., intentionally relaxing information-flow restrictions on data [4]. To define noninterference, we are concerned only with executions that do not downgrade information to $\mathcal{A}$ (or, more precisely, from p to q where $p \not\sqsubseteq \mathcal{A}$ and $q \sqsubseteq \mathcal{A}$). We expect generalizations of noninterference, like robust declassification [26], [27] and nonmalleable information-flow [28] to hold for Flamio, but in this work we consider only the case where no node downgrades information to $\mathcal{A}$. To capture this intuition, we introduce the notion of a *bad* release event. Intuitively, when no bad release events are emitted nothing is being downgraded to $\mathcal{A}$.

b) Bad release events: We call downgrading of confidentiality labels *declassification*, and downgrading of integrity labels *endorsement*. As Flamio permits both declassification and endorsement using delegations, a bad event should capture

$$\textsc{E-Write-Ev}\quad \frac{[\ldots] \qquad ev = (\mathsf{write}(a, e'), \Sigma, n)}{n;\Sigma \vdash \langle \phi \mid E[a := e'] \rangle \xrightarrow{ev} \langle \phi' \mid E[\mathsf{return}\ ()] \rangle : \sigma}$$

$$\textsc{G-Step-Ret-Ev}\quad \frac{S(n) = \langle \phi_n \mid E[\mathsf{wait}(\tau)] \rangle \qquad S(m) = \langle \phi_m \mid v; \overline{e_m} \rangle \qquad ev = (\mathsf{ret}(v, n), \Sigma, m)}{(\!| m \cdot n \cdot \overline{n}, \Sigma, S |\!) \overset{ev}{\Longrightarrow} (\!| n \cdot \overline{n}, \Sigma, S\left[n \mapsto s'_n, m \mapsto s'_m\right] |\!)}$$

$$\textsc{G-Step-Local-Ev}\quad \frac{n;\Sigma \vdash S(n) \xrightarrow{ev} s : \sigma}{(\!| n \cdot \overline{n}, \Sigma, S |\!) \overset{ev}{\Longrightarrow} (\!| n \cdot \overline{n}, \Sigma\left[n \mapsto \sigma\right], S\left[n \mapsto s\right] |\!)}$$

$$\textsc{E-Assume-Ev}\quad \frac{[\ldots] \qquad ev = (\mathsf{release}(p, q, r), \Sigma, n)}{n;\Sigma \vdash \langle \phi \mid E[\mathsf{assume}\ (p \succcurlyeq q) \mathbin{@} r] \rangle \xrightarrow{ev} \langle \phi \mid E[\mathsf{return}\ ()] \rangle : \sigma}$$

Fig. 14: Augmented semantics emitting events.

both cases. We call a release event $ev = (\mathsf{release}(p, q, r), \Sigma, n)$ bad, written $\mathcal{A} \vdash \mathsf{bad}(ev)$, if $n;\Sigma \vdash r \sqsubseteq \mathcal{A}$ and one of the following conditions hold:

1) $n;\Sigma \vdash p \sqsubseteq \mathcal{A}^{\to}$ and $n;\Sigma \vdash q \not\sqsubseteq \mathcal{A}^{\to}$
2) $n;\Sigma \vdash \mathcal{A}^{\leftarrow} \sqsubseteq p$ and $n;\Sigma \vdash \mathcal{A}^{\leftarrow} \not\sqsubseteq q$

The condition $n;\Sigma \vdash r \sqsubseteq \mathcal{A}$ captures that a release event can be bad only if $\mathcal{A}$ can observe the delegation. Condition 1 captures bad declassifications: the new delegation gives $\mathcal{A}$ the authority to observe values labeled as q since $\mathcal{A}$ can already observe values labeled as p. Similarly, condition 2 captures bad endorsements: the new delegation gives $\mathcal{A}$ the authority to write values labeled as q since $\mathcal{A}$ can already write values labeled as p.

c) Examples of bad release events: The release event $ev = (\mathsf{release}(\mathcal{A}, \mathsf{Alice}^{\to}, \perp^{\sqsubseteq}), \Sigma, n)$ generated by the local step

$$n;\Sigma \vdash \langle \phi \mid \mathsf{assume}\ (\mathcal{A} \succcurlyeq \mathsf{Alice}^{\to}) \mathbin{@} \perp^{\sqsubseteq} \rangle \xrightarrow{ev} \langle \phi \mid e \rangle : \sigma$$

(i.e., a declassification from $\mathsf{Alice}^{\to}$ to $\mathcal{A}$) is bad since $n;\Sigma \vdash \perp^{\sqsubseteq} \sqsubseteq \mathcal{A}$ (i.e., $\mathcal{A}$ can observe the delegation), $n;\Sigma \vdash \mathcal{A} \sqsubseteq \mathcal{A}^{\to}$ (i.e., it is a declassification to a principal that $\mathcal{A}$ can observe), and $n;\Sigma \vdash \mathsf{Alice}^{\to} \not\sqsubseteq \mathcal{A}^{\to}$ (i.e., it is a declassification from a principal that $\mathcal{A}$ previously could not observe). The release event $ev = (\mathsf{release}(\mathcal{A}, \mathsf{Bob}^{\leftarrow}, \perp^{\sqsubseteq}), \Sigma, n)$ generated by the local step

$$n;\Sigma \vdash \langle \phi \mid \mathsf{assume}\ (\mathcal{A} \succcurlyeq \mathsf{Bob}^{\leftarrow}) \mathbin{@} \perp^{\sqsubseteq} \rangle \xrightarrow{ev} \langle \phi \mid () \rangle : \sigma$$

(i.e., an endorsement from $\mathcal{A}$ to $\mathsf{Bob}^{\leftarrow}$) is bad since $n;\Sigma \vdash \perp^{\sqsubseteq} \sqsubseteq \mathcal{A}$ (i.e., $\mathcal{A}$ can observe the delegation), $n;\Sigma \vdash \mathcal{A}^{\leftarrow} \sqsubseteq \mathcal{A}$ (i.e., it is an endorsement from a principal that $\mathcal{A}$ can modify), and $n;\Sigma \vdash \mathcal{A}^{\leftarrow} \not\sqsubseteq \mathsf{Bob}^{\leftarrow}$ (i.e., it is an endorsement to a principal that $\mathcal{A}$ previously could not modify).

Finally, the release event $ev = (\mathsf{release}(\mathcal{A}, \mathsf{Charlie}, \perp^{\sqsubseteq}), \Sigma, n)$ is bad as it corresponds to both a declassification and an endorsement, as both condition 1 and condition 2 holds.

When a release event is not bad, we say that it is *good*, and we extend the definition of good release events to traces: a trace t is *good*, written $\mathcal{A} \vdash \mathsf{good}(t)$, if t does not contain any bad release events. Our noninterference result, presented at the end of this section, is quantified over good traces only, and we leave the problem of extending this result to more relaxed notions of noninterference as future work.

d) $\mathcal{A}$-equivalence: Given a trace t the $\mathcal{A}$-observable trace of t is the trace $t \upharpoonright \mathcal{A}$, defined as

$$\varepsilon \upharpoonright \mathcal{A} = \varepsilon$$

$$((\alpha, \Sigma, n) \cdot t) \upharpoonright \mathcal{A} = \begin{cases} (\alpha, \Sigma, n) \cdot (t \upharpoonright \mathcal{A}) & n; \Sigma \vdash \Sigma(n).\mathsf{lbl} \sqsubseteq \mathcal{A} \\ t \upharpoonright \mathcal{A} & \text{otherwise} \end{cases}$$

We augment the semantics from Section IV with events. Figure 14 shows an excerpt of the augmented semantics (we use $[\ldots]$ to elide the premises presented in Section IV). Except for the emitted event these rules correspond exactly to the rules in Figures 6, 7 and 9. We write $(\!|\overline{n}, \Sigma, S|\!) \xRightarrow{t}{}^{*}$ when there exists a configuration $(\!|\overline{n}', \Sigma', S'|\!)$ such that $(\!|\overline{n}, \Sigma, S|\!) \xRightarrow{t}{}^{*} (\!|\overline{n}', \Sigma', S'|\!)$ and $S'(n) = \langle \phi_n \mid \bullet \rangle$ for all n. We also write $(\!|\overline{n}, \Sigma, S|\!) \xRightarrow{\mathcal{A} \rightsquigarrow t'}{}^{*}$ when $(\!|\overline{n}, \Sigma, S|\!) \xRightarrow{t}{}^{*}$ and $t' = t \upharpoonright \mathcal{A}$.

We define an $\mathcal{A}$-equivalence relation that makes explicit which traces and memories an attacker $\mathcal{A}$ can distinguish. As they both contain expressions, we define an $\mathcal{A}$-equivalence on expressions, and Figure 15 shows an excerpt of this judgment. Intuitively, two expressions e_1 and e_2 are considered $\mathcal{A}$-equivalent if the current label on each context does not flow to $\mathcal{A}$, or if the label on each context flows to $\mathcal{A}$ and e_1 and e_2 are equal "up to labeled values with a label that does not flow to $\mathcal{A}$". Figure 15 formalizes this intuition: rule EQ-HIGH states that two expressions are $\mathcal{A}$-equivalent in environments where the current label does not flow to $\mathcal{A}$, and the remaining rules state that two expressions are $\mathcal{A}$-equivalent if the current label of each environment flows to $\mathcal{A}$, and the expressions are equal up to labeled values that $\mathcal{A}$ cannot observe.

For most cases $C \vdash e_1 \simeq^{\theta}_{\mathcal{A}} e_2$ recursively inspects the subexpressions of each expression, but a few cases need special care: to relate dynamically allocated locations we use a partial bijection [29], [30] $\theta : \mathsf{Loc} \rightharpoonup \mathsf{Loc}$ in EQ-ADDR. The most important rules are EQ-LABELED-1 (that makes explicit the notion that an attacker cannot distinguish two terms labeled with a principal which does not flow to $\mathcal{A}$), and EQ-LABELED-2 (that states that $\mathcal{A}$ can "look inside" terms labeled with principals that flow to $\mathcal{A}$). When θ is not important we write $n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq_{\mathcal{A}} e_2$ to mean $n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq^{\theta}_{\mathcal{A}} e_2$ for some bijection θ.

The $\mathcal{A}$-equivalence relation on expressions induces an $\mathcal{A}$-equivalence relation on events, and $\mathcal{A}$-equivalence of traces are defined as pairwise $\mathcal{A}$-equivalence of the events in the trace. We write $t_1 \simeq^{\theta}_{\mathcal{A}} t_2$ for $\mathcal{A}$-equivalence on traces, and $t_1 \simeq_{\mathcal{A}} t_2$ to mean $t_1 \simeq^{\theta}_{\mathcal{A}} t_2$ for some bijection θ.

$$\textsc{Eq-High}\quad \frac{i = 1,2 \qquad n; \Sigma_i \vdash \Sigma_i(n).\mathsf{lbl} \not\sqsubseteq \mathcal{A}}{n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq^{\theta}_{\mathcal{A}} e_2}
\qquad
\textsc{Eq-Addr}\quad \frac{\theta(a_1) = a_2 \qquad n; \Sigma_i \vdash \Sigma_i(n).\mathsf{lbl} \sqsubseteq \mathcal{A} \quad i = 1,2}{n; \Sigma_1; \Sigma_2 \vdash a_1 \simeq^{\theta}_{\mathcal{A}} a_2}$$

$$\textsc{Eq-Labeled-1}\quad \frac{n; \Sigma_i \vdash p_i \not\sqsubseteq \mathcal{A} \qquad n; \Sigma_i \vdash \Sigma_i(n).\mathsf{lbl} \sqsubseteq \mathcal{A} \quad i = 1,2}{n; \Sigma_1; \Sigma_2 \vdash e_1 \mathbin{@} p_1 \simeq^{\theta}_{\mathcal{A}} e_2 \mathbin{@} p_2}$$

$$\textsc{Eq-Labeled-2}\quad \frac{n; \Sigma_i \vdash q \sqsubseteq \mathcal{A} \qquad n; \Sigma_i \vdash \Sigma_i(n).\mathsf{lbl} \sqsubseteq \mathcal{A} \quad i = 1,2 \qquad n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq^{\theta}_{\mathcal{A}} e_2}{n; \Sigma_1; \Sigma_2 \vdash e_1 \mathbin{@} q \simeq^{\theta}_{\mathcal{A}} e_2 \mathbin{@} q}$$

Fig. 15: $\mathcal{A}$-equivalence for terms and expressions.

$$\textsc{Store-Eq-Empty}\quad \frac{}{C \vdash \varnothing \simeq^{\theta}_{\mathcal{A}} \varnothing}
\qquad
\textsc{Store-Eq-Low}\quad \frac{\theta(a_1) = a_2 \qquad C \vdash e_1 \simeq^{\theta}_{\mathcal{A}} e_2 \qquad C_i \vdash q \sqsubseteq \mathcal{A} \quad i = 1,2 \qquad C \vdash \phi_1 \simeq^{\theta}_{\mathcal{A}} \phi_2 \qquad \phi'_i = \phi_i\,[a_i \mapsto (e_i \mathbin{@} q)]}{C \vdash \phi'_1 \simeq^{\theta}_{\mathcal{A}} \phi'_2}$$

$$\textsc{Store-Eq-High-1}\quad \frac{C_1 \vdash q \not\sqsubseteq \mathcal{A} \qquad C \vdash \phi_1 \simeq^{\theta}_{\mathcal{A}} \phi_2}{C \vdash \phi_1\,[a \mapsto e \mathbin{@} q] \simeq^{\theta}_{\mathcal{A}} \phi_2}
\qquad
\textsc{Store-Eq-High-2}\quad \frac{C_2 \vdash q \not\sqsubseteq \mathcal{A} \qquad C \vdash \phi_1 \simeq^{\theta}_{\mathcal{A}} \phi_2}{C \vdash \phi_1 \simeq^{\theta}_{\mathcal{A}} \phi_2\,[a \mapsto e \mathbin{@} q]}$$

Fig. 16: $\mathcal{A}$-equivalence for memories. The meta-variable C abbreviates $n; \Sigma_1; \Sigma_2$, and $C_i = n; \Sigma_i$.

Finally, Figure 16 shows $\mathcal{A}$-equivalence on memories. Rule STORE-EQ-EMPTY states that two empty memories are $\mathcal{A}$-equivalent, and rule STORE-EQ-LOW states that extending two memories with $\mathcal{A}$-equivalent expressions preserves $\mathcal{A}$-equivalence. Lastly, rules STORE-EQ-HIGH-1 and STORE-EQ-HIGH-2 states that both memories can be extended with terms labeled with a principal that does not flow to $\mathcal{A}$ without the attacker being able to distinguish the memories. We extend the notion of $\mathcal{A}$-equivalence on memories to $\mathcal{N}$-indexed sets of memories and write $m; \Sigma_1; \Sigma_2 \vdash \Phi \simeq^{\theta}_{\mathcal{A}} \Psi$ when $\forall n \in \mathcal{N}.\ m; \Sigma_1; \Sigma_2 \vdash \Phi(n) \simeq^{\theta}_{\mathcal{A}} \Psi(n)$.

To simplify the statement of our end-to-end security guarantee note that, for any $n, m \in \mathcal{N}$ we have $n; \varnothing; \varnothing \vdash \phi \simeq_{\mathcal{A}} \psi$ if and only if $m; \varnothing; \varnothing \vdash \phi \simeq_{\mathcal{A}} \psi$. Thus, we write $\phi \simeq_{\mathcal{A}} \psi$ to mean $n; \varnothing; \varnothing \vdash \phi \simeq_{\mathcal{A}} \psi$ for some n.

e) Attacker knowledge: We present our noninterference result using the notion of attacker knowledge [25], [31]. The attacker knowledge is the set of initial memories that could lead to a given observable trace, and a larger knowledge set corresponds to more uncertainty about the initial memory. Formally, attacker knowledge, given a trace t produced by expression e, is the set $k^n_{\mathcal{A}}(e,t)$.

$$k^n_{\mathcal{A}}(e,t) = \left\{ \Psi \,\middle|\, (\!|n, \varnothing, S|\!) \xRightarrow{\mathcal{A} \rightsquigarrow t'}{}^{*} \wedge t \simeq_{\mathcal{A}} t' \right\}$$

where $S(m) = \langle \Psi(n) \mid [\![e]\!]_n(m) \rangle$ and

$$[\![e]\!]_n(m) = \begin{cases} e & \text{if } n = m \\ \bullet & \text{otherwise} \end{cases}$$

ensures that expression e is initially evaluated on node n, and the remaining nodes start with an empty list of expressions. As is standard for termination-insensitive noninterference (TINI), the policy [14] is defined as all $\mathcal{A}$-equivalent terminating memories, which we denote by $k^{\downarrow n}_{\mathcal{A}}(\Phi, e)$.

$$k^{\downarrow n}_{\mathcal{A}}(\Phi, e) = \{ \Psi \mid \Phi \simeq_{\mathcal{A}} \Psi \wedge (\!|n, \varnothing, S|\!) \Longrightarrow^{*} \}$$

where $S(m) = \langle \Psi(n) \mid [\![e]\!]_n(m) \rangle$. TINI can now be stated for Flamio as a guarantee that two traces, generated by two evaluations of a well-typed expression e starting with $\mathcal{A}$-equivalent memories Φ and Ψ, are $\mathcal{A}$-equivalent. Using attacker knowledge, we can succinctly write this as the inclusion of $k^{\downarrow n}_{\mathcal{A}}(\Phi, e)$ in $k^n_{\mathcal{A}}(e,t)$: all $\mathcal{A}$-equivalent terminating memories generate $\mathcal{A}$-observable traces.

Theorem 5.1 (Noninterference): Let e satisfy $n; \Gamma \vdash e : \tau$. If $(\!|n, \varnothing, S|\!) \xRightarrow{\mathcal{A} \rightsquigarrow t}{}^{*}$ for $S(m) = \langle \Phi(m) \mid [\![e]\!]_n(m) \rangle$ such that $\mathcal{A} \vdash \mathsf{good}(t)$ then $k^n_{\mathcal{A}}(e,t) \supseteq k^{\downarrow n}_{\mathcal{A}}(\Phi, e)$.

The technical report [32] contains a complete proof of Theorem 5.1. The theorem shows that we have securely integrated FLAM into an LIO-like setting with a floating label, where proofs of trust relationships do not inappropriately reveal confidential information, nor are they inappropriately affected by untrusted information.

VI. Implementation and case studies

We have implemented Flamio as a monadic library in Haskell [13]. The code is approximately 2,100 lines of code in total, and uses FLAM's efficient query resolution algorithm for authorization queries [1]. Proof search for trust relationships is implemented as computations in the Flamio monad, ensuring that delegations are not used inappropriately. The case studies demonstrate how application-specific search strategies are used to prevent label creep during proof search.

Along with the efficient query resolution algorithm, we cache query results to avoid repeated network communication.[5] To simplify the implementation, we differ from the calculus in the following ways:

1) An RPC does not send the function that should be called across the network. Instead, the receiver of the RPC has a table mapping identifiers to functions, and the caller sends this identifier along with the list of arguments.
2) Since query results obtained via network communication are cached on a per-query basis, no two identical queries are sent to the same node.

First, 1) does not lead to loss of expressivity: as shown by Cooper and Wadler [33], a program in a calculus similar to that of Section IV can be translated by performing defunctionalization to a Haskell program (which can then use the Flamio implementation).

Second, 2) significantly reduces network communication but means that the implementation is unsound if the trust relationship between principals changes during query resolution. Orthogonal work on query isolation [34] can provide transactional behavior for distributed systems like Flamio.

Using this implementation, we have constructed three use cases for Flamio consisting of roughly 500 lines of code. The first use case is a distributed bank, which was already presented in Section III. The banking example shows how users can perform remote procedure calls to handle transactions across accounts between different users. A user u can authorize user u' to transfer money on behalf of u by adding a delegation $u' \succcurlyeq u \mathbin{@} \mathsf{bank}^{\rightarrow} \wedge u^{\leftarrow}$ locally on u's node. This delegation is read as "u trusts u', and this information is confidential to bank and has the integrity of u". When bank wishes to prove the trust relationship between u and u' to authorize a transfer of money, a proof search is issued, and using the Fwd rule, is forwarded to node u. Node u then proves the trust relationship using the local delegation $u' \succcurlyeq u \mathbin{@} \mathsf{bank}^{\rightarrow} \wedge u^{\leftarrow}$.

In addition to the banking example, we construct a secure social jukebox service where people schedule music during social gatherings. The third use case is a secure database containing confidential information about government agencies. The third example also demonstrates how the Assump rule prevents infinite derivation trees in authorization queries, and how such queries can show up in practical use cases.

A. Secure social jukebox service

Suppose a group of principals $\mathcal{N}$ is gathered at a party and want to vote on which songs should be played at the party, but do not want their votes leaked to unauthorized principals, nor do they want unauthorized principals to vote on their behalf. For instance, if Alice votes for "Taylor Swift - Shake It Off", she wants to ensure that only principals she trusts can learn her vote for this song. Furthermore, only principals that Alice trusts should be able to vote for a song on her behalf.

We assume a distinguished principal $J \in \mathcal{N}$ (for jukebox) representing a node on which two functions exist:

$$\begin{aligned} &\mathtt{get} : \mathsf{LIO}\,(\mathsf{String} \times [\mathsf{Labeled\,String}]) \\ &\mathtt{put} : \mathsf{String} \times [\mathsf{Labeled\,String}] \rightarrow \mathsf{LIO\,Unit} \end{aligned}$$

Function $\mathtt{get}$ returns a pair (s, lss) containing the current song being played s, and a list of labeled strings lss such that $s' \mathbin{@} p \in \mathit{lss}$ represents that p voted for s'. So if Alice wants to vote for "Taylor Swift - Shake It Off", she appends a labeled

[5]We do not consider side channel attacks introduced by caching queries, such as external timing attacks, in this work.

```
(λ^Alice_Alice← _ . do
  assume J→ ≽ Alice→ @ (⊥→ ∧ Alice←)
  ls <- label Alice "Taylor Swift - Shake It Off"
  (curSong, votedSongs) <- get
  put (curSong, ls :: votedSongs)) ()
```

Fig. 17: Alice places a secret vote for Taylor Swift.

string "Taylor Swift - Shake It Off" @ Alice using put. By labeling her vote with the principal Alice, she knows that only principals p such that $\mathsf{Alice}^{\rightarrow}$ flows to $p^{\rightarrow}$ can learn her vote (i.e., by E-UNLABEL it must be the case that Alice flows to $p^{\rightarrow}$). Furthermore, since the integrity of the label on the vote is $\mathsf{Alice}^{\leftarrow}$, she knows that any vote of the form $s \mathbin{@} \mathsf{Alice}$ for some song s must be placed by a principal p such that $\mathsf{Alice}^{\leftarrow}$ trusts $p^{\leftarrow}$ (i.e., by E-LABEL it must be the case that p's current label flows to Alice).

Figure 17 shows an example of Alice voting for "Shake it Off" by Taylor Swift. First, Alice adds a delegation that allows J to read Alice's labeled vote. She then labels her vote and inserts it into the list of labeled songs. When J then unlabels the labeled song title, a proof search will be issued checking $J^{\leftarrow} \sqcup \mathsf{Alice}$ flows to $J^{\rightarrow}$, which is equivalent to checking that $\mathsf{Alice}^{\rightarrow}$ trusts $\mathsf{Alice}^{\rightarrow} \wedge (J \vee \mathsf{Alice})^{\leftarrow}$. This trust relationship holds by the delegation that Alice placed in Figure 17.

B. Government agency records

As a final example demonstrating the usefulness of Flamio, we show how confidential delegations can be used to keep government agency records. Suppose the CIA hires a subset of $\mathcal{N}$ as agents. This information should visible only to other CIA agents, not to the general public. Furthermore, only the CIA should be able to hire agents.

We implement an enforcement mechanism for this security policy as follows: when $\mathsf{CIA} \in \mathcal{N}$ hires an agent $n \in \mathcal{N}$, two delegations are created: first, trust is delegated from CIA to $\mathsf{CIA}\colon n$. By the properties of ownership projections [1], this delegation can be read as "CIA trusts n, but CIA controls which principals can act for n".[6] To satisfy the security policy, this delegation should be visible only to other agents, and CIA should be able to trust that any such delegation could have been placed only by the CIA. This directly translates into the label $(\mathsf{CIA}\colon\mathsf{AgentDB})^{\rightarrow} \wedge \mathsf{CIA}^{\leftarrow}$ on the delegation.

Second, any agent n should be able to check if a given principal m is an agent. To achieve this, $(\mathsf{CIA}\colon\mathsf{AgentDB})^{\rightarrow}$ delegates trust to $n^{\rightarrow}$, meaning that n can learn information labeled as $\mathsf{CIA}\colon\mathsf{AgentDB}$. To satisfy the security policy, this delegation should also be labeled as $(\mathsf{CIA}\colon\mathsf{AgentDB})^{\rightarrow} \wedge \mathsf{CIA}^{\leftarrow}$.

Figure 18 shows how Alice can verify that Bob is also a secret agent. On line 4, Alice and Bob are both hired as secret agents using the function hire defined on lines 1-3.

[6]The use of ownership principals prevent *delegation loopholes* [1].

```
1  let hire = λ^CIA_CIA← n . do
2    assume CIA:n ≽ CIA @ (CIA:AgentDB)→ ∧ CIA←
3    assume n→ ≽ (CIA:AgentDB)→ @ (CIA:AgentDB)→ ∧ CIA←
4  in (λ^CIA_CIA← _ . do hire Alice; hire Bob) ()
5  [...]
6  (λ^Alice_Alice← _ . withStrategy [Alice→ ∧ CIA←] (do
7                isAgent <- CIA:Bob ≽ CIA
8                if isAgent then [...] else [...])) ()
```

Fig. 18: Alice verifies that Bob is a secret agent for the CIA.

At some point, Alice meets Bob "in the field" and wants to verify his claim that he is a secret agent. In order for Alice to verify this claim she checks if the principal CIA delegates to $\mathsf{CIA}\colon\mathsf{Bob}$. Using the strategy $[\mathsf{Alice}^{\rightarrow} \wedge \mathsf{CIA}^{\leftarrow}]$, Alice states that she is only interested in using delegations that have confidentiality at most Alice, and any delegation must have been placed by a principal that CIA trusts.

For Alice to prove this trust relationship she uses the FWD rule and delegates the proof search to CIA. By FWD, she must check that $\mathsf{Alice}^{\leftarrow}$ (i.e., the current label of Alice) flows to $\mathsf{CIA}^{\rightarrow}$ (i.e., the clearance label of CIA), which holds by BOT. Applying DEL, CIA must now check that $(\mathsf{CIA}\colon\mathsf{AgentDB})^{\rightarrow}$ trusts $\mathsf{Alice}^{\rightarrow}$. Applying DEL again, this reduces to showing that $(\mathsf{CIA}\colon\mathsf{AgentDB})^{\rightarrow}$ trusts $\mathsf{Alice}^{\rightarrow}$ under the assumption that $(\mathsf{CIA}\colon\mathsf{AgentDB})^{\rightarrow}$ trusts $\mathsf{Alice}^{\rightarrow}$, and so the trust relationship is established using ASSUMP.

Notice that without the ASSUMP rule, we would not be able to prove that $(\mathsf{CIA}\colon\mathsf{AgentDB})^{\rightarrow}$ trusts $\mathsf{Alice}^{\rightarrow}$ (or, equivalently, that $\mathsf{CIA}\colon\mathsf{AgentDB}$ can learn information labeled as Alice): we would keep applying DEL without a way of terminating the derivation. But, intuitively, this relationship should hold as we have a delegation of trust from $(\mathsf{CIA}\colon\mathsf{AgentDB})^{\rightarrow}$ to $\mathsf{Alice}^{\rightarrow}$. Rule ASSUMP allows the proof search to assume the delegation when proving that it is secure to use the delegation, effectively expressing that "it is secure to use the delegation because the delegation says so": a form of checked endorsement [23], [24].

VII. RELATED WORK

A. FLAM

The FLAM technical report [22] presents a security-typed language $\mathsf{F}\lambda$ in which policies are FLAM principals. Much like Flamio, $\mathsf{F}\lambda$ can delegate trust during evaluation and allows querying of trust relationships. However, the decision of whether or not to allow downgrading (i.e., adding new trust relationships) must be performed statically using a relatively simple type system. By contrast, in Flamio all decisions about whether to allow downgrading are done during evaluation, meaning that Flamio can potentially allow more downgrading and remain secure. As $\mathsf{F}\lambda$ is a language with fine-grained IFC the programmer is also burdened with more label annotations than would be expected in similar Flamio programs.

FLAC [35] is a calculus for flow-limited authorization that allows static reasoning about mechanisms such as commitment schemes or bearer credentials that require dynamic authorization. FLAC builds a sophisticated type system on top of FLAM that provides noninterference and robust declassification guarantees. Although FLAC offers many high-level features to build practical authorization mechanisms, it uses a limited subset of FLAM, e.g., it does not have distributed trust checking (corresponding to FLAM's FWD rule).

Hyperflow [36] is a new processor architecture for non-malleable timing-sensitive IFC that uses FLAM principals encoded as bit vectors as the label model. This encoding offers efficient computation of joins, meets, and projections. Hyperflow extends the RISC-V processor with IFC instructions and limits how information flows through registers and memory pages. The hardware is programmed in a new hardware-description language, ChiselFlow, embedded in Scala. Much like Flamio, each process in a Hyperflow processor contains a current label and a clearance label, but unlike our model, Hyperflow requires the programmer to raise the current label explicitly. This decision avoids the possible side-channel caused by raising the current label depending on sensitive information (a channel that we close in Flamio), at the cost of putting the burden of raising the label on the programmer.

B. LIO and coarse-grained information-flow

Coarse-grained IFC has traditionally been applied mostly in operating system security [5], [8] by associating a single label with a process. LIO [9] implements coarse-grained IFC as a monadic Haskell library using a current label that can float up to the computation's clearance label, similar to Flamio.

Recent work [30], [37] on the expressiveness of fine-grained versus coarse-grained IFC shows that they have the same expressive power. Rajani and Garg [30] encourage the use of coarse-grained IFC as it places less annotation burden on the programmer. Based on our experience with Flamio, we agree with this observation.

C. RPCs and distributed computation

Our semantics for remote procedure calls is inspired by the *Location-aware Simple Abstract Machine* (LSAM) [38]. A global LSAM configuration is a set of local LSAM configurations indexed by names, and remote procedure calls are performed by suspending the computation and transferring control to another local LSAM configuration. Whereas LSAM considers a first-order language (useful as a target language for compiling other languages for distributed computation), our semantics handles higher-order functions in the style of the source language in [33], and LSAM would be a natural compilation target for our work.

Much work has been devoted to information-flow programming languages for distributed systems. Both SIF [39] and Swift [23] are based on Jif [4], and target web applications by tracking confidentiality and integrity of data sent between a server and a client. Fabric [34] extends Jif with remote procedure calls and transactions, and enforces security by using a combination of static and dynamic enforcement mechanisms. Fabric and Flamio share many features: Both have trust orderings on principals that can be queried and modified at runtime, and remote nodes communicate via RPC. On the other hand, Fabric and Flamio differ in many ways: Fabric is a language with fine-grained IFC, while Flamio is a coarse-grained system build as a library directly on top of LIO, which in itself is a library in Haskell.

Fabric prevents information leakage from *read channels* (i.e., if node n accesses data on node m depending on information confidential to n, node m learns about n's confidential information) using *access labels*, but unlike Flamio does not protect against read channels arising from authorization queries [1].

VIII. CONCLUSION

This paper demonstrates the usefulness of the FLAM authorization logic for a language with coarse-grained dynamic information-flow control, in the style of the floating label model of LIO. The paper shows that the two systems can be combined to obtain a provably strong noninterference result for a language with distributed computation and decentralized trust. The language has been implemented as a monadic library in Haskell, and the usability of the system has been validated via three use cases involving secure, distributed access to shared resources.

IX. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for insightful comments. We also thank Owen Arden, Anitha Gollamudi and Pablo Buiras for discussions regarding FLAC, FLAM, and LIO. The first author thanks the programming languages group at Harvard University for the inspiring environment during his visit. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1524052. This material is also supported by the Aarhus University Research Foundation (AUFF).

REFERENCES

[1] O. Arden, J. Liu, and A. C. Myers, "Flow-Limited Authorization", in *Proceedings of the 2015 IEEE 28th Computer Security Foundations Symposium*, IEEE Computer Society, 2015,

[2] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn, "A Role-based Access Control Model and Reference Implementation Within a Corporate Intranet", *ACM Transactions on Information and System Security*, Feb. 1999.

[3] F. Pottier and V. Simonet, "Information Flow Inference for ML", *ACM Transactions on Programming Languages and Systems*, Jan. 2003.

[4] A. C. Myers, "JFlow: Practical Mostly-static Information Flow Control", in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1999,

[5] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres, "Making Information Flow Explicit in HiStar", in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, USENIX Association, 2006,

[6] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres, “Securing Distributed Systems with Information Flow Control”, in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, USENIX Association, 2008,

[7] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris, “Labels and Event Processes in the Asbestos Operating System”, in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ACM, 2005,

[8] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, “Information Flow Control for Standard OS Abstractions”, in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ACM, 2007,

[9] D. Stefan, D. Mazieres, J. C. Mitchell, and A. Russo, “Flexible Dynamic Information Flow Control in the Presence of Exceptions”, *Journal of Functional Programming*, 2017.

[10] P. Buiras, D. Vytiniotis, and A. Russo, “HLIO: Mixing Static and Dynamic Typing for Information-flow Control in Haskell”, in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ACM, 2015,

[11] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazieres, “Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems”, in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ACM, 2012,

[12] P. Buiras and B. van Delft, “Dynamic Enforcement of Dynamic Policies”, in *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, ACM, 2015,

[13] M. Pedersen and S. Chong, *Flamio implementation in Haskell*, https://www.dropbox.com/s/zxy991pjeepl8nn/FLAMinLIO.zip, 2018.

[14] J. A. Goguen and J. Meseguer, “Security Policies and Security Models”, in *1982 IEEE Symposium on Security and Privacy*, Apr. 1982,

[15] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, “Authentication in Distributed Systems: Theory and Practice”, *ACM Transactions on Computer Systems*, Nov. 1992.

[16] M. Abadi, “Access Control in a Core Calculus of Dependency”, in *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, Portland, Oregon, USA: ACM, 2006,

[17] P. Wadler, “Monads for Functional Programming”, in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, Springer-Verlag, 1995,

[18] S. Moore, C. Dimoulas, R. B. Findler, M. Flatt, and S. Chong, “Extensible Access Control with Authorization Contracts”, in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, 2016,

[19] M. Felleisen, “The Theory and Practice of First-class Prompts”, in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1988,

[20] G. Smith and D. Volpano, “Secure Information Flow in a Multi-threaded Imperative Language”, in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1998,

[21] S. Muller and S. Chong, “Towards a Practical Secure Concurrent Language”, in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ACM, 2012,

[22] O. Arden, J. Liu, and A. C. Myers, “Flow-limited authorization: Technical Report”, Cornell University Computing and Information Science, Tech. Rep., May 2015.

[23] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, “Secure Web Applications via Automatic Partitioning”, in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ACM, 2007,

[24] A. Askarov and A. C. Myers, “Attacker Control and Impact for Confidentiality and Integrity”, *Logical Methods in Computer Science*, 2011.

[25] A. Askarov and A. Sabelfeld, “Gradual Release: Unifying Declassification, Encryption and Key Release Policies”, in *2007 IEEE Symposium on Security and Privacy*, May 2007,

[26] S. Zdancewic and A. C. Myers, “Robust Declassification”, in *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, IEEE Computer Society, 2001,

[27] A. C. Myers, A. Sabelfeld, and S. Zdancewic, “Enforcing Robust Declassification”, in *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, IEEE Computer Society, 2004,

[28] E. Cecchetti, A. C. Myers, and O. Arden, “Nonmalleable Information Flow Control”, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017,

[29] A. Banerjee and D. A. Naumann, “Secure Information Flow and Pointer Confinement in a Java-like Language”, in *Proceedings of the 15th IEEE Workshop on Computer Security Foundations*, IEEE Computer Society, 2002,

[30] V. Rajani and D. Garg, “Types for Information Flow Control: Labeling Granularity and Semantic Models”, in *2018 IEEE 31st Computer Security Foundations Symposium*, IEEE, Jul. 2018.

[31] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, “Termination-Insensitive Noninterference Leaks More Than Just a Bit”, in *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, Springer-Verlag, 2008,

[32] M. Pedersen and S. Chong, “Programming with Flow-Limited Authorization: Coarser is Better (Technical Report)”, Tech. Rep., 2018.

[33] E. E. Cooper and P. Wadler, “The RPC calculus”, in *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM, 2009,

[34] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ACM, 2009,

[35] O. Arden and A. C. Myers, “A Calculus for Flow-Limited Authorization”, in *2016 IEEE 29th Computer Security Foundations Symposium*, Jun. 2016,

[36] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh, “Hyperflow: A Processor Architecture for Nonmalleable, Timing-Safe Information-Flow Security”, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2018.

[37] M. Vassena, A. Russo, D. Garg, V. Rajani, and D. Stefan, “From Fine- to Coarse-Grained Dynamic Information Flow Control and Back”, *Proceedings of the ACM on Programming Languages*, Jan. 2019.

[38] K. Narita and S. Nishizaki, “A Parallel Abstract Machine for the RPC Calculus”, in *Informatics Engineering and Information Science*, A. Abd Manaf, S. Sahibuddin, R. Ahmad, S. Mohd Daud, and E. El-Qawasmeh, Eds., Springer Berlin Heidelberg, 2011,

[39] S. Chong, K. Vikram, and A. C. Myers, “SIF: Enforcing Confidentiality and Integrity in Web Applications”, in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Boston, MA: USENIX Association, 2007,