FLOW-LIMITED AUTHORIZATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Owen Arden

FLOW-LIMITED AUTHORIZATION

Owen Arden, Ph.D.

Cornell University

Enforcing the confidentiality and integrity of information is critical in distributed applications. Production systems typically use some form of authorization mechanism to protect information, but these mechanisms do not typically provide end-to-end information security guarantees. Information flow control mechanisms provide end-to-end security, but their guarantees break down when trust relationships may change dynamically, a common scenario in production environments. This dissertation presents *flow-limited authorization*, a new foundation for enforcing information security. Flow-limited authorization is an approach to authorization in that it can be used to reason about whether a principal is permitted to perform an action. It is an approach to information flow control in that it can be used to reason about whether a flow of information is secure.

This dissertation presents the theoretical foundation of this approach, the Flow-Limited Authorization Model (FLAM). FLAM uses a novel principal algebra that unifies authority and information flow policies and a logic for making secure authorization and information flow decisions. This logic ensures that such decisions cannot be influenced by attackers or leak confidential information.

We embed the FLAM logic in a core programming model, the Flow-Limited Authorization Calculus (FLAC). FLAC programs selectively enable flows of information; the type system ensures that attackers cannot create unauthorized flows. A well-typed FLAC not only ensures proper authorization, but also secure information flow.

The FLAC approach to secure programming is instantiated in FLAME, a library and compiler plugin for enforcing flow-limited authorization in Haskell programs. Flame uses type-level constraints and monadic effects to statically enforce flow-limited authorization for Haskell programs in a modular way.

BIOGRAPHICAL SKETCH

Owen Arden was born in Oxford, Mississippi to Celia and James Barnett Jr. An initial interest in video games gradually transformed into an obsession with building and programming computers upon his parents purchase of a second-hand personal computer running an Intel 286 CPU. He first learned to program in BASIC, and occasionally convinced his middle-school teachers to let him turn in a program as a class project.

Owen's first job was for Carter Computer and Blueprinting in Tupelo, MS, in 1995 repairing Packard Bell computers—in between running blueprints. He eventually became interested in computer security and networking, a fact that became apparent to Tim Tsai, the owner of his dial-up internet provider, Futuresouth Communications. Tim graciously gave him a job doing system administration and technical support.

Upon graduating from Tupelo High School in 1999, Owen accepted a scholarship from the National Security Agency to study at Georgia Tech. As part of this program he interned each summer for the NSA at Ft. Meade. After completing his Bachelor of Science (Highest Honors) in Computer Engineering, he joined the NSA's Tailored Access Operations Group and built systems for automated vulnerability analysis. After several years of finding vulnerabilities in programs, Owen felt the need to return to graduate school and learn how to prevent them in the first place. In August of 2009, he began graduate school at Cornell University and has since been working toward his Ph.D. in Computer Science.

For Caroline, who made this possible, and Maggie, who makes coming home extra fun.

ACKNOWLEDGMENTS

This dissertation would not exist without the help of many people. Foremost, I want to thank Andrew Myers, who encouraged and brought focus to my curiosity throughout my time at Cornell. He has made me a better writer, a better researcher, and a better thinker. His infectious intellectual enthusiasm spurred my interest in programming languages after taking his compilers course, and his approach to security changed the way I think about building secure systems. I hope to be half the scientist and teacher he is.

My committee members have also been indispensable during my study at Cornell. Dexter Kozen encouraged my early interest in abstract algebra, which proved quite useful. Sam Madden helped me understand the concrete aspects of distributed systems, how to measure them, and how to build better abstractions based on them. Gün Sirer sparked my interest in decentralized systems and instilled in me an appreciation of clean, readable code. Fred Schneider reminds me that it is our job to explain, not our reader's job to understand; he has helped bring more clarity to many ideas in this dissertation.

The graduate students at Cornell have also been a vital part of my time here. Cornell Ph.D. students excel at self-organization, and because of that I have never lacked for an opportunity to play frisbee, to "play" hockey, or to socialize with other graduate students. Aaron Rosenberg, from the English department, kept me sane with frequent squash games, along with good converstation about being old students and new dads.

I want to thank the members of the PLab, my office mates, Jonathan DiLorenzo, Fabian Muehlboeck, Andrew Hirsch, Fran Mota, Matthew Milano, and Xiang Long. They are all both funny and intelligent, and it was a great pleasure bouncing ideas off of them, talking about topics from type theory to effective doorstop mechanisms, and reacting to the foul balls from the baseball field that occasionally hit our window.

The students of the Applied Programming Languages group have been great colleagues. Jed Liu, Mike George, K. Vikram, Aslan Askarov, Danfeng Zhang, Chin Is-

radisaikul, Tom Magrino, Yizhou Zhang, Matthew Milano, Isaac Sheff, and Ethan Cecchetti were all frequent sources of insight (and code). I will always try to be the kind of programmer Jed is. He and Ethan are also particularly good at finding errors in my proofs and helping to fix them. Conversations with Aslan Askarov germinated the idea of unifying authorization and information flow control. Mike George introduced me to the Dependency Core Calculus and had the idea of treating proofs of authorization as first-class values. All of the members of APL have given feedback on many drafts and practice talks, and the presentation of my work is far better because of it.

There are also many people without whom I would likely not have made it to Cornell in the first place. My parents, Jamie and Celia Barnett, constantly supported my curiosity and growth and never let the fact we lived in a small town in Mississippi get in the way of that. They also overcame their initial electrical misgivings to let me, at 16, rewire an additional phone line (to share with my sister) for my U.S. Robotics 56K fax modem. They made me feel confident of my abilities without making me feel I needed to achieve for their approval. And they taught me that it is never too late to make a change. My sister, Elizabeth Barnett, reminds me of the fullness of life, how to take joy in it, and how to be open to what it brings. She was also very understanding even though my modem was online whenever she wanted to use the phone.

I have been lucky to have talented teachers throughout my life. I want to mention Lynn McAlpin and Bonnie Webb especially. Their tireless dedication to the Tupelo High School Academic Decathlon team taught me that knowledge and insight comes from hard work and practice, not innate ability. I would not be here without them.

I also want to thank Daniel Ridge and Vic Zandy of the Institute for Defense Analyses Center for Computing Sciences. They expanded my interest in computer security to include a fascination with programming language design. My decision to return to graduate school was due in large part to them, and to academics I met through them like

Mike Hicks, Jeff Foster, and Jeff Hollingsworth. Vic died on June 18, 2013, and I will always feel his absence as a colleague and a friend.

Finally, to my wife Caroline I owe my deepest gratitude. Without her support, I likely would not have returned to graduate school, and without her I could not have finished. It also would have been a far lonelier experience. She was a constant source of love and encouragement, a helpful editor, and a great partner. She is a phenomenal mother to our daughter Maggie, and brings all her considerable intelligence and creativity to bear as a parent. Caroline, I love you, and am deeply grateful.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A major concern of computer security is the protection of information. There are several dimensions of information worthy of protection, but of particular interest are its confidentiality and integrity. To protect the confidentiality of information, a system must ensure that unauthorized entities cannot learn that information, either in whole or in part. To protect the integrity of information, a system must ensure that unauthorized entities cannot corrupt or otherwise influence the content of that information.

Most production systems today protect information using some form of *access control*. An access control mechanism uses authorization data to represent access policies for protected resources. Authorization data may come in many forms, but typically falls into one of two categories: access control lists and capabilities. Access control lists associate a list of entities with each protected resource. Access requests from entities on the list are granted; other requests are denied. Capability-based mechanisms associate permissions with a credential or token possessed by an entity. Here, access requests are granted when they include a valid credential with appropriate permissions. Requests with invalid credentials or insufficient permissions are denied.

When configured and deployed properly, these mechanisms have proven relatively effective at securing information. However, modern systems are increasingly *distributed*, *decentralized*, and *dynamic*. Existing approaches are ill-suited for developing and deploying authorization mechanisms that are secure in these new environments.

Distribution Distributed applications often use distributed authorization mechanisms that store authorization data or perform authorization-related computations remotely. Distribution may introduce vulnerabilities that were not anticipated by their security models. Authorization data itself may be confidential, so permitting remote access may reveal secret information. Low-integrity authorization data could enable an attacker to

influence the application in harmful ways. Furthermore, the act of accessing remote authorization data may itself reveal secret information in the application's state.

Decentralization Modern systems are also increasingly decentralized. Entities in fully decentralized systems may have very limited or even no trust in each other. Deploying authorization mechanisms in decentralized settings is challenging since entities may disagree on what actions are authorized or who may authorize them.

Dynamism For large-scale applications with millions of users, security policies are constantly changing as new information is created, relationships between users are updated, or new services are integrated. Controlling when and how security policies may change is very challenging, but also crucial to the information security of modern distributed applications.

These aspects of modern systems demand new mechanisms for enforcing information security. One appealing approach is *information flow control*. Information flow control enables the expression of high-level information security policies describing the end-to-end behavior of the system. These policies are inherently compositional. Further, they can be formally characterized in terms of semantic security conditions such as noninterference [35], permitting rigorous proofs that enforcement mechanisms enforce policies as intended.

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

Decentralized information flow control (DIFC) [63] introduced the idea that information flow control mechanisms could control the use of downgrading mechanisms

through an authorization mechanism. In a DIFC system, information flow labels are therefore expressed using the vocabulary of the authorization mechanism. For example, the original Decentralized Label Model (DLM) [63] expresses labels in terms of principals. Delegations between principals (expressing the trust between those principals) affect which information flows are permitted. Information may be *relabeled* from one label to another only if the target label is at least as restrictive as the originating one. Subsequent DIFC systems use labels expressed in terms of *tags* combined with capabilities [28, 48, 73, 99], or tags combined with principals [22].

The narrow interactions between authorization and information flow in these DIFC systems permit many details of the authorization mechanism to be abstracted away. At this high level of abstraction, many existing approaches to authorization would seem applicable to DIFC settings, including authorization logics [2, 49, 78], role-based access control (RBAC) [33], and trust management [9, 51, 83]. Unfortunately, this level of abstraction omits important aspects of authorization mechanisms that impact the security of the information they are meant to protect—especially in the distributed, decentralized, and dynamic settings most relevant to modern applications.

An important oversight in previous work is that authorization mechanisms themselves are implemented as components of the system they are meant to secure. Like the other system components, authorization mechanisms access stored data and perform computation on it, but the information flows created by these mechanisms is often overlooked in the authorization models they are based upon. These flows are significant because they can create a channel through which information may be accidentally leaked or corrupted, or can even allow the authorization mechanism to be used as a means to attack the system.

This oversight has created a blind spot in many authorization models: confidentiality. Most models cannot express authorization policies that are confidential or are

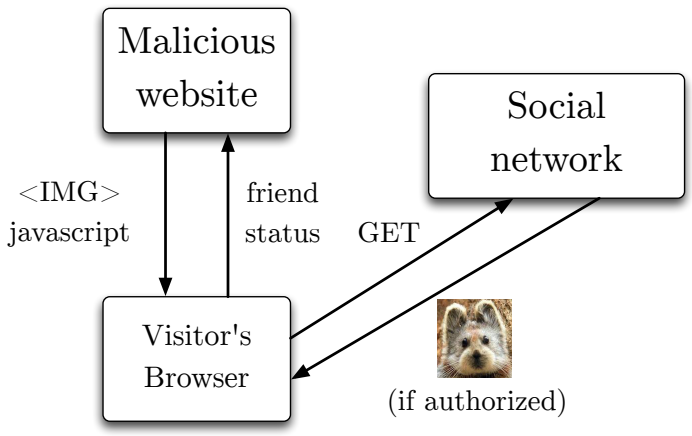


Figure 1.1: A JavaScript probing attack. A malicious website can learn the friends of a visitor by embedding images that only a friend of particular users may access.

based on confidential data. Real systems, however, use confidential data for authorization all the time: users on social networks receive access to photos based on friend lists, frequent fliers receive tickets based on credit card purchase histories, and doctors exchange patient data while keeping doctor–patient relationships confidential. While many models can ensure, for instance, that only friends are permitted to access a photo, few can say anything about the secondary goal of preserving the confidentiality of the friend list. Such authorization schemes may fundamentally require *some* information to be disclosed, but failing to detect these disclosures can lead to unintentional leaks.

For instance, a malicious website can "probe" the friends of a visitor by embedding an image that is only accessible to friends of a particular user on a social network. This attack is illustrated in figure 1.1. The website sends an `IMG` tag referring to the protected image, along with code that checks whether the image loads successfully. If the visitor is authorized to view the image, then she must be a friend of the user. The problem here is that access to the image is authorized based on confidential information: the visitor's friendship with the user. However, the result of the authorization is not treated as confidential. Thus the website is able to learn that the visitor and the user are friends.

Developers have been aware of these kinds of attacks for years [20], and even used them to gather information about their visitors [71].

Authorization without integrity is meaningless, so authorization mechanisms are typically better at enforcing integrity. However, many formal models make unreasonable or unintuitive assumptions about integrity. For instance, in many models (for example, [49], [2], [51]) authorization policies either do not change or change only when modified by a trusted administrator. This is a reasonable assumption in centralized systems where such an administrator will always exist, but in decentralized systems, there may be no single entity that is trusted by all other entities.

Even in centralized systems, administrators must be careful when performing updates based on partially trusted information, since malicious users may try to use the administrator to carry out an attack on their behalf. Unfortunately, existing models offer little help to administrators that need to reason about how attackers may have influenced security-critical update operations.

Another difficulty in building secure systems is that assumptions made by the authorization mechanism may be subtly incompatible with assumptions made by the application that employs it. For instance, in the original Fabric system [54], nodes only run code that is installed by an administrator of the node. The security model, therefore, makes the reasonable assumption that attackers cannot control what code is run by a node, though they might attempt to subvert security via remote procedure calls or malicious data.

Under this model, covert channels have limited utility to an attacker. For example, a read channel [98] is a covert channel that occurs when a public resource is accessed for a secret reason; the provider of the resource learns something about the secret since the resource was accessed. Because attackers have limited control over where resources are allocated or the contexts they were accessed in, permitting these channels was deemed

an acceptable risk.

A later extension of Fabric, called Mobile Fabric [6], provides support for secure mobile code. This extension gives attackers more control over what code runs at a node, meaning that covert channels, especially read channels, are much easier to leverage. Consequently, Mobile Fabric introduced *access policies*, which control where data is allocated and in what contexts it may be accessed. Enforcing these policies eliminates read channels.

However, access policies were not applied to authorization data in Mobile Fabric. Neither were there any restrictions on the context of authorization queries or how to enforce the information security of the results of these queries. This means that updates to authorization data are not adequately protected and that queries might reveal information about the querying context. Characterizing and eliminating the vulnerabilities that were introduced by this gap in enforcement led to a re-examination of how the Decentralized Label Model represents authorization.

The result, *flow-limited authorization*, recasts information flow control itself as a kind of authorization mechanism. Instead of authorizing *access* to information, information flow control authorizes *flows* of information. Furthermore, since any computation creates flows of information, all authorization mechanisms must enforce the information security of the authorization data they process. Thus, flow-limited authorization mechanisms make explicit any assumptions they have regarding the confidentiality and integrity of authorization data and query results. By representing the information security assumptions and guarantees of authorization mechanisms, we obtain a more general security model that unifies authorization with information flow control. This approach extends the notion of a principal's authority or privilege level from the set of actions a principal may perform to include the set of flows of a principal may receive or influence.

1.1 Modeling information flow and authorization

In real systems, it is important that the trust relationships be able to change by delegating or revoking trust, but these changes can lead to security vulnerabilities:

- Delegations of authority can enable information relabeling equivalent to unauthorized downgrading.
- Relabeling information limits a principal's ability to revoke access to that information in the future.
- Changes to trust relationships can leak information from the agent performing the change.
- Authorization queries can leak information from the agent performing the query.

All but the most limited existing information flow control and authorization models are susceptible to at least some of these security vulnerabilities, including several systems [9, 39, 59, 60, 83, 91, 92] designed to handle changes in trust securely.

To address these vulnerabilities, this dissertation describes a new approach called *flow-limited authorization*. Flow-limited authorization explicitly models how information flows both through updates to trust relationships and through the authorization mechanism itself.

The theoretical foundation of this approach is the Flow-Limited Authorization Model (FLAM). This model comprises two primary components. The first component is a novel *principal algebra* that unifies principals with information flow labels, providing a clean, abstract vocabulary for exploring interactions between authorization and information flow. The second component is a logic for reasoning about relationships between principals so that authorization and information flow decisions are made securely.

FLAM's system model, illustrated in Figure 1.2, consists of a set of principals that each store some authorization data locally. Principals derive trust relationships using

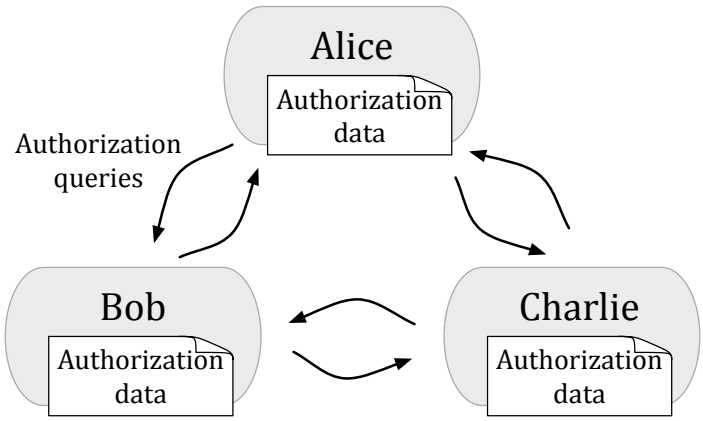


Figure 1.2: FLAM system model. Principals communicate to derive trust relationships from local and remote authorization data.

their local authorization data and by communicating with other principals. This view makes FLAM applicable to many challenging real-world scenarios, including federated distributed systems, where hosts in the network may have mutual distrust.

FLAM has several additional properties that make it attractive as a security model.

- It enables fully decentralized trust in the sense that each principal's view of the trust configuration is represented and all principals' policies are enforced simultaneously.
- It explicitly models communication, making it appropriate for systems whose authorization data is distributed across multiple hosts.
- It makes no assumptions regarding the well-formedness of authorization data or the honesty of hosts. Any number of malicious hosts may store malicious delegations and participate in distributed authorization.
- All derivations in the FLAM logic satisfy an important security property called *robust authorization* that ensures malicious delegations and revocations cannot affect which flows are authorized.

- It has been formalized in Coq [57], along with machine-checked proofs that FLAM enforces robust authorization.

1.2 Building information security abstractions

A great deal of effort has gone into developing new and better security mechanisms like cryptographic protocols, blockchain technology, and trusted hardware. In contrast, less effort has focused on developing better security *abstractions*. The lack of good abstractions makes these mechanisms more difficult to use, and errors that undermine the security of these mechanisms are common [56]. Incorporating information flow control into the design of security abstractions is attractive, since it offers compositional, end-to-end security guarantees. The Flow-Limited Authorization Model provides a foundation upon which information flow control can be applied to these mechanisms in a meaningful way.

The second part of this dissertation demonstrates that FLAM can be embedded into a core programming model for authorization and information flow control, so that dynamic authorization mechanisms—as well as the programs that employ them—can be statically verified. Approaching the verification of programs from this perspective is attractive for two reasons. First, it gives a model for building secure authorization mechanisms by construction rather than verifying them after the fact. This model offers programmers insight into the sometimes subtle interaction between information flow and authorization, and helps programmers address problems early, during the design process. Second, it addresses a core weakness lurking at the heart of existing language-based security schemes: that the underlying policies may change in a way that breaks security. By statically verifying the information security of dynamic authorization mechanisms, we expand the real-world scenarios in which language-based information flow control is useful and strengthen its security guarantees.

The Flow-Limited Authorization Calculus (FLAC) is a functional language for designing and verifying decentralized authorization protocols. FLAC supports dynamic authorization while enforcing information flow control. It uses FLAM's principal model and FLAM's logical reasoning rules to define an operational model and type system for authorization computations that preserve information security. FLAC is inspired by the Polymorphic Dependency Core Calculus [2] (DCC).[1] Abadi develops DCC as an authorization logic, but DCC is limited to static trust relationships defined externally to DCC programs by a lattice of principals.

The types in a FLAC program can be considered propositions [88] in an authorization logic, and the programs can be considered proofs that the proposition holds. Well-typed FLAC programs are not only proofs of authorization, but also proofs of secure information flow, ensuring the confidentiality and integrity of authorization policies and of the data those policies depend upon.

FLAC is useful from a logical perspective, but also serves as a core programming model for real language implementations. Since FLAC programs can dynamically authorize computation and flows of information, FLAC applies to more realistic settings than previous authorization logics. Thus FLAC offers more than a type system for proving propositions—FLAC programs do useful computation.

FLAC programs exhibit strong semantic security. Programs in low-integrity contexts exhibit *noninterference*, ensuring attackers cannot leak or corrupt information, and cannot subvert authorization mechanisms. Programs in higher-integrity contexts exhibit *robust declassification*, ensuring attackers cannot influence authorized disclosures of information.

[1] DCC was first presented in [3]. We use the abbreviation DCC to refer to the extension to polymorphic types in [2].

1.3 Enforcing flow-limited authorization in Haskell

The final component of this dissertation is FLAME, a library for enforcing flow-limited authorization in Haskell applications. Flame helps programmers develop security abstractions for their authorization mechanisms and ensures that applications use these abstractions securely.

Flame does this by embedding elements of the FLAC type system into Haskell programs using type-level *constraints* that are checked by the compiler. A custom GHC [85] type checker plug-in checks these constraints using implementations of algorithms developed for FLAM. Implementing Flame as a lightweight compiler plug-in and library makes it easier to interact with existing libraries, even libraries that implement authorization mechanisms.

Haskell is a particularly attractive language for our purposes. It is a pure functional language, keeping our implementation close to the syntax and abstractions of FLAC. In Haskell, input, output, and observable side effects must occur in the IO monad. This makes it easier to enforce information flow control via the type system. Despite its sophisticated type system, Haskell is a popular programming language [72] with an active developer community that is familiar with the approach of using expressive types to obtain strong run-time guarantees for their programs.

To demonstrate the versatility of Flame, we have created secure bindings for libmacaroons [32], an implementation of Macaroons [13], which authorizes users based on bearer credentials with caveats. We discuss the workflow for integrating this library into a Flame program.

CHAPTER 2

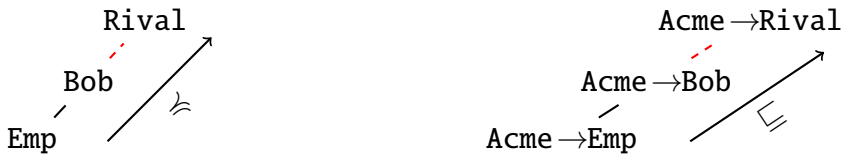
VULNERABILITIES IN EXISTING APPROACHES

Delegation and revocation of trust are important features of authorization mechanisms and the DIFC systems that build upon them, but previous approaches fall short with respect to both their expressiveness and the security guarantees they offer. Three classes of vulnerabilities arise in these authorization mechanisms. We motivate them primarily in the context of the Decentralized Label Model, but the vulnerabilities discussed in this section are generally applicable to other DIFC and authorization models.

2.1 Delegation loopholes

Delegation of trust allows principals in a system to specify other principals that may act on their behalf. In addition to representing trust between two entities, delegation may encode membership in groups or roles represented by principals. Most previous models treat delegations as universally agreed-upon, but in a decentralized system, different principals can have different opinions about delegations. Most previous information flow models, including the DLM, ignore the implications of allowing the trust configuration to be controlled by partially trusted principals. As we show, a partially trusted principal can choose to delegate to an untrusted principal, and thereby achieve the effect of downgrading information even when it has not been granted the authority to downgrade. We call this use of delegation to achieve downgrading the *delegation loophole*. Some previous work [9, 39, 83] does observe this connection between delegation and downgrading, but does not eliminate the influence attackers may exert on which flows are authorized.

To see how the delegation loophole works, consider the following example of an insider attack. Bob, who works for Acme, has been enticed to disclose valuable trade



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

Figure 2.1: Delegation loophole. By delegating to `Rival`, `Bob` effectively declassifies a label owned by `Acme`. Dashed lines indicate relationships influenced by `Bob`.

secrets to Rival, one of Acme's competitors. Acme's policies[1] are written in terms of principals using the DLM [63]. In the DLM, principals like `Emp` can express membership of a group by delegating to other principals using the acts-for relation $\succcurlyeq$, where we read the expression $p \succcurlyeq q$ as "p acts for q". Thus, the DLM trust configuration consists of a set of delegations of the form $p \succcurlyeq q$, called the *principal hierarchy*. In several DIFC systems [21,22,63], confidentiality and integrity policies have an associated *owner* principal, expressing the authority necessary to enforce or downgrade the label. In the DLM, Acme's trade secrets might be protected under the label component `Acme` $\rightarrow$ `Emp`. Here, `Acme` denotes the owner of a confidentiality [2] policy. This confidentiality policy states that the associated information is readable only by the employees of Acme, to whom the principal `Emp` delegates. These would include `Bob`, since `Bob` $\succcurlyeq$ `Emp`.

The idea of the DLM is that only Acme itself should be able to release data labeled `Acme` $\rightarrow$ `Emp`, because `Acme` is the owner of the policy. However, if an employee like `Bob` is able to control his own delegations, he can effectively release information to a third party. For instance, Figure 2.1 shows how Bob might abuse his access to Acme's trade secrets. Figure 2.1a shows a trust configuration $\mathcal{H}$ comprising two delegations: `Bob` $\succcurlyeq$ `Emp`, and `Rival` $\succcurlyeq$ `Bob`. An edge indicates that the higher principal is trusted to act on behalf of the lower principal; the dashed line indicates that Bob has delegated trust to Rival.

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

[2] indicated by the right arrow, $\rightarrow$. Integrity policies are denoted by a left arrow, $\leftarrow$

Figure 2.1b shows the restrictiveness of information flow policies in $\mathcal{H}$. An edge indicates that the higher policy is at least as restrictive as the lower policy, or in other words, information with the lower policy may be *relabeled* to the higher policy. Bob's delegation causes Acme to believe it is safe to relabel information from the policy `Acme` $\rightarrow$ `Emp` to the policy `Acme` $\rightarrow$ `Rival`, since `Bob` is trusted by `Emp` and `Rival` is trusted by `Bob`. This influence of Bob causes Acme's own system to disclose sensitive data to Rival.

Although the DLM allows the trust configuration to evolve by adding or removing delegations, it ignores the possibility that changes to the trust configuration may create insecure information flows, However, recent systems built on the DLM, such as SIF [24] and Fabric [54], give principals the power to control their delegations dynamically. These systems have therefore opened up the delegation loophole.

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

2.2 Poaching attacks

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

Existing DIFC systems have particularly unsatisfying semantics with regard to the consequences of revocation. The root of the problem is that current DIFC systems per-

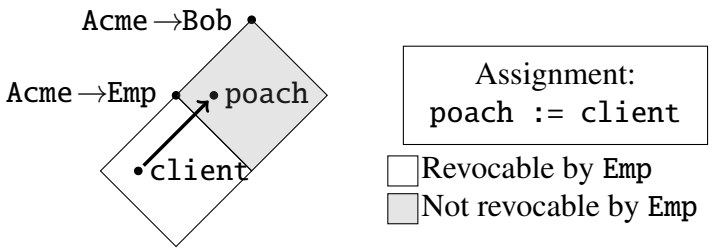


Figure 2.2: Poaching attack. If information at Acme → Emp is relabeled to a more restrictive policy, Emp can no longer revoke access.

mit information to flow between different policies without regard to a principal's ability to revoke access in the future. This makes it difficult to reason about what information a principal retains access to after a revocation.

Suppose Acme protects its client list with the policy Acme → Emp so that only employees may read it. Figure 2.2 illustrates how Bob can use his access to Emp data to "poach" Acme's client list, storing it with a more restrictive policy, to which he retains access in the event of a revocation. The white region beneath Acme → Emp represents the part of the lattice of information flow policies in which information can be relabeled to Acme → Emp. The shaded region represents information with policies that relabel to Acme → Bob, but not to Acme → Emp. The assignment poach:=client assigns the contents of variable client protected by a policy in the white region to variable poach protected by a policy in the shaded region. Since Emp delegates to Bob, policies in the white region may be relabeled to Acme → Bob.

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

Like the delegation loophole, poaching attacks demonstrate that relabeling is a kind of policy downgrade exploitable by an insider. However, the two vulnerabilities dif-

fer. Delegation *enables future relabelings* to occur; therefore, to eliminate loopholes, relabelings must only be based on trusted delegations. On the other hand, relabeling *prevents future revocations* from occurring; therefore, to prevent poaching, the decision to relabel a policy should be trusted by the policy owner.

2.3 Leaking information via authorization

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

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

This side channel is an instance of a *read channel* [98], in which accesses to data leak information from the accessor. Read channels arising from authorization queries have been largely ignored in the DIFC literature, perhaps because the implementation platform was originally assumed to be trusted. In a fully distributed system, however, different parts of the computing infrastructure, including the implementation of the trust configuration, may be provided by differently trusted principals. The Fabric system therefore adds *access labels* [6] to control information flows via read channels. Fabric does not, however, consider read channels arising from authorization requests.

The second source of information leakage via authorization arises if a public decision is based on the result of an authorization query whose answer depends on a secret trust relationship. Several distributed authorization systems [59, 60, 90–93, 100] protect sensitive credentials with access policies, but do not constrain how credentials are used after granting access, resulting in possible leaks. These systems do not guard against authorization side channels.

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

2.4 Vulnerabilities in other systems

Many systems have some degree of vulnerability to the attacks described above. Authorization mechanisms that make no attempt to control information flow are certainly vulnerable. Additionally, almost all previous DIFC systems that leverage an underlying authorization mechanism are vulnerable to attacks that abuse the way authorization controls information flow. Clearly, systems based on the DLM, such as Jif [62] and Fabric [54], have these weaknesses. Capability-based DIFC systems such as Asbestos [28], Histar [99], Flume [48], Laminar [73], and LIO [81] also exhibit delegation loopholes and poaching attacks, since processes may transfer capabilities and relabel information. Aeolus [22] has some characteristics of capability-based systems, but maintains a trust configuration like Fabric. It too is vulnerable to these attacks.

CHAPTER 3

A MODEL FOR FLOW-LIMITED AUTHORIZATION

3.1 Unifying principals and policies

Our goal is a simple model that supports reasoning about authorization, about information flow, and about their interactions, and that guides the construction of secure distributed systems. Our model, which we call the Flow-Limited Authorization Model (FLAM), addresses all the security issues discussed in Section 2. FLAM is both an authorization logic and an information flow model. It is an authorization logic (like [2, 49, 78]) since it derives judgments about trust. It is an information flow model (like [15, 63, 83]) since it derives judgments about secure information flow. FLAM integrates reasoning about trust and reasoning about information flow; this integration is central to preventing the security vulnerabilities identified in Section 2. We are unaware of prior models that support this kind of combined reasoning.

For simplicity, FLAM completely unifies principals, roles, privileges, and information flow labels, a perhaps surprising feature that distinguishes FLAM from previous models for either authorization or information flow[1]. In FLAM, principals are *both* authorization entities *and* information flow policies enforcing confidentiality and integrity. In subsequent discussion, we sometimes use *label* (or *policy*) to talk about a *principal* used to specify permitted information flow, but these concepts are interchangeable in FLAM. As we show, unifying principals with information flow labels enables a simpler, algebraic presentation of the relationships between information flow policies and the principals they concern.

This section provides the formal basis for unifying authority and decentralized information flow policies. Although the algebraic definitions given in this section may appear

[1] Some prior work has unified *roles* with information flow labels, while distinguishing principals from roles [9, 77, 83].

complex at first, we show in Section 3.3 that they enable a concise logic, collected in Figures 3.3 and 3.4. Authorization decisions derived from this logic are protected from the problems discussed in Section 2.

3.2 Authority projections

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

Let $\mathcal{N}$ be the set of all primitive principals, which are essentially uninterpreted names. Starting from primitive principals, we can construct more complex compound principals. For any two principals p and q, we represent the conjunction of their authority, the authority of *both* p and q, as the compound principal $p \wedge q$. Likewise, the authority of *either* p or q is written $p \vee q$. These conjunction and disjunction operators, as in Boolean algebra, define a lattice[2] over principals. If a principal q trusts principal p, then we say p *acts for* q and write $p \succcurlyeq q$. If q represents the privilege or permission to perform an action, the statement $p \succcurlyeq q$ means p has the right to perform that action. Lattice properties imply $p \wedge q \succcurlyeq p \succcurlyeq p \vee q$ for any p and q.

Conjunction and disjunction are already familiar from previous logics for authentication and authorization, and the acts-for relation of FLAM is related to the speaks-for relation of authentication logics [2, 49], but Section 3.3.4 draws a distinction between the speaks-for relation for FLAM and the acts-for relation.

In many DIFC models, the flows-to ordering $\sqsubseteq$ between information flow policies

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

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

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

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

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

These five properties capture the essence of limited authority derived from a principal's general authority, without requiring separate classes of entities such as *roles* [33], *subprincipals*, or *groups* [78]. Naturally, the originating principal acts for the derived authority (1), and projection preserves the properties of the authorization lattice (2, 3, 4). Finally, projections are idempotent (5).

FLAM defines two classes of authority projections, *basis projections* and *ownership projections*. Basis projections define the different kinds of authority a principal

may possess, specifically, confidentiality and integrity, whereas ownership projections (discussed in Section 3.2.2) attenuate a principal's authority relative to other principals.

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

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

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

We represent all authority as a combination of confidentiality and integrity authority (1), so any principal that acts for both projections of a principal also acts for the principal. Additionally, composing (2) or taking the meet (3) of confidentiality and integrity projections yields $\bot$. In this paper, we focus on information flow policies for confidentiality and integrity, but we expect it is possible to extend FLAM with additional projections that represent other aspects of security. For instance, [101] adds *availability* policies, and [55] includes *reference authority* and *persistence* policies. We leave representing such policies as basis projections to future work.

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

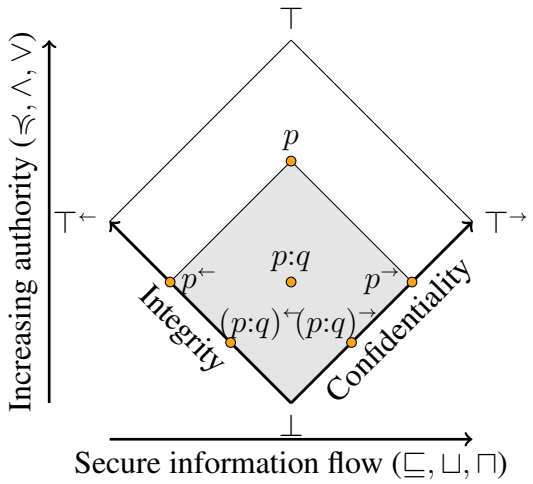


Figure 3.1: The FLAM lattices for trust and information flow

cipals with highest and lowest authority, respectively. Joins in $\mathcal{P}_0$ are the conjunctions of principals ($\wedge$), and meets are disjunctions ($\vee$).

3.2.1 The information flow ordering

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

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

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

$$p \sqsubseteq q \overset{\triangle}{\iff} q^{\rightarrow} \wedge p^{\leftarrow} \succcurlyeq p^{\rightarrow} \wedge q^{\leftarrow}$$

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

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

The flows-to relation $\sqsubseteq$ is a preorder, so we can lift it to a partial order just as we did for acts-for, with equivalences defined by $a \equiv_{\sqsubseteq} b \iff (a \sqsubseteq b \text{ and } b \sqsubseteq a)$. The relation $\sqsubseteq$ induces an *information flow* lattice $(\mathcal{P}_0, \sqsubseteq)$. In this lattice, we represent joins by $\sqcup$ and meets by $\sqcap$. The top element of $(\mathcal{P}_0, \sqsubseteq)$ is the policy that most restricts use of the information, *secret* and *untrusted*: $\top^{\rightarrow} \wedge \bot^{\leftarrow}$. The bottom element is the least restrictive policy, *public* and *trusted*: $\bot^{\rightarrow} \wedge \top^{\leftarrow}$. We often omit projections of the $\bot$ principal to obtain the more concise (but equivalent) principal representation; for example, $p^{\rightarrow}$ instead of $p^{\rightarrow} \wedge \bot^{\leftarrow}$ and $p^{\leftarrow}$ instead of $\bot^{\rightarrow} \wedge p^{\leftarrow}$.

By the definitions above, the equivalence classes of $\succcurlyeq$ and $\sqsubseteq$ are identical, and there is a one-to-one correspondence between the elements of $(\mathcal{P}_0, \succcurlyeq)$ and $(\mathcal{P}_0, \sqsubseteq)$, even though the two orderings are "at right angles" to each other. Figure 3.1 illustrates this correspondence by aligning both lattices on the same set of elements. Secure information flow is from left to right, toward increasing confidentiality and decreasing integrity. The trust ordering is bottom to top, toward increasing authority. This correspondence allows us to easily translate relationships from one ordering to another when convenient.

3.2.2 Owned principals

To give FLAM the expressive power of some previous authorization systems, such as *role-based access control* (RBAC) [33] and the DLM [63], we introduce another way to construct principals. In RBAC, principals are assigned *roles* which they may select when performing sensitive tasks, and access control policies are specified in terms of

roles that are permitted access. It is tempting to use delegation to express authorization concepts such as roles and *groups* [78]. However, this approach fails to adequately control modification of role membership. For instance, if Acme uses the principal `Emp` to represent a role by delegating to all Acme employees, then Bob can effectively add employees via delegation. What Acme requires is a way to refer to principals like Bob while retaining control over their trust relationships. Then a principal like `Emp` can delegate to such a principal without risking subversion of its authorization mechanism.

From the perspective of information flow control, the principals from the set $\mathcal{P}_0$ can represent both authority and information flow policies, but the information flow policies expressible with these principals are rather limited—they are not *decentralized* in the sense of the DLM [63]. The key aspect of decentralized policies is that policy *owners* retain control over decisions to release information.

In FLAM, we express ownership as a special class of authority projections called *ownership projections*. The *owned principal* `Acme:Bob` represents[3] Bob as a principal whose trust relationships Acme retains control of. Intuitively, `Acme:Bob` delegates trust to the same principals as Bob, but only if Acme allows the delegation. Acme may also create new delegations of trust from `Acme:Bob` even though Acme doesn't act for Bob. Owned principals are similar in spirit to *roles* [33], *groups*, and *subprincipals* [78], but are first-class principals that may delegate and be delegated to.

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

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

and $s^{\rightarrow} \succcurlyeq q^{\rightarrow}$.

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

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

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

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

delegation to a corresponding owned principal $p{:}q$, but only if the owners have an acts-for relationship ($p \succcurlyeq r$).

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

Using ownership projections, we can further extend our set of principals. Let $\mathcal{O} = \{:p \mid p \in \mathcal{P}_0\}$ be a set of ownership projections. Then let $\mathcal{P}$ be the closure of $\mathcal{P}_0$ under the projections in $\mathcal{O}$. Like $\mathcal{P}_0$, the equivalence classes of $\mathcal{P}$ form lattices $(\mathcal{P}, \succcurlyeq)$ and $(\mathcal{P}, \sqsubseteq)$, whose elements have a one-to-one correspondence. Figure 3.1 relates an owned principal, $p{:}q$ and its projections, to the other elements of these lattices. For the remainder of this paper, principals are implicitly members of the set $\mathcal{P}$ unless otherwise specified.

3.2.3 FLAM normal form

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

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

$$p ::= J^{\rightarrow} \wedge J^{\leftarrow} \qquad M ::= L \mid L \vee M$$
$$J ::= M \mid M \wedge J \quad L ::= n \mid L{:}L$$

Our prototype implementation, discussed in Section 3.5, includes an algorithm for converting FLAM principals to normal form. This algorithm is relatively straightforward: it applies lattice properties and equational rules of authority projections as rewrite rules to reduce principals to normal form. We have formalized and proved this algorithm correct in Coq. The rewriting rules are found in Appendix A.2.

3.3 Secure reasoning with dynamic trust

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

3.3.1 System model and trust configuration

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

This means that a host may specify the confidentiality of a delegation to prevent other hosts from learning it, but it also means that a host may label a delegation with high integrity when other hosts might deem it untrustworthy.

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

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

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

3.3.2 Flow-limited judgments

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

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

However, constructing a proof in a distributed system creates information flows. Consequently, this form of judgment has two fundamental problems. First, it fails to

characterize the confidentiality and integrity of the conclusion $p \succcurlyeq q$. Second, it fails to represent or constrain the side-effects inherent in the distributed computation that is performed across the hosts that collectively store the trust configuration $\mathcal{H}$. Communicating with these hosts to obtain the delegations in D could leak confidential information about the query. Failure to constrain how authority may be used to relabel these delegations could permit poaching attacks.

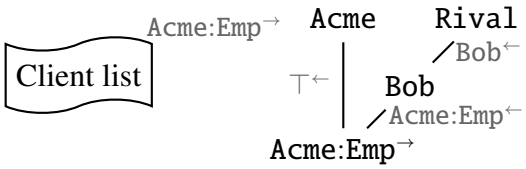
FLAM solves both problems by parameterizing authorization queries with policies that restrict the flow of information as the query is answered. The resulting *flow-limited judgments* have the following form:

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

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

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

As with the trust configuration $\mathcal{H}$, we make no well-formedness assumptions about the query label or derivation label specified in authorization queries. However, to protect their own security, we assume that honest hosts specify a query label for top-level queries that characterizes the confidentiality and integrity of the issuing context; hence



$$\boxed{\mathcal{H}; c; \texttt{Acme:Emp}^{\leftarrow}; \texttt{Acme:Emp}^{\leftarrow} \not\Vdash \texttt{Acme:Emp}^{\rightarrow} \sqsubseteq \texttt{Rival}^{\rightarrow}} \quad (3.1)$$

$$\boxed{\mathcal{H}; c; \texttt{Bob}^{\leftarrow}; \texttt{Acme:Emp}^{\leftarrow} \not\Vdash \texttt{Acme:Emp}^{\rightarrow} \sqsubseteq \texttt{Bob}^{\rightarrow}} \quad (3.2)$$

Figure 3.2: Section 2 attacks prevented. The boxed judgments do not hold robustly with the illustrated delegations. Judgment (3.1) does not hold since Bob's delegation to Rival cannot be used to robustly relabel Acme's policies, closing the delegation loophole. In (3.2), the query label $\texttt{Bob}^{\leftarrow}$ has insufficient integrity to relabel Acme's policies, preventing Bob from poaching the client list.

the name *pc* for the *program counter* label, as in Jif [62]. Likewise, we assume honest hosts will treat query results in accordance with the derivation label.

3.3.3 Robust derivations

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

Consider the example of Section 2.1. We can model this scenario with the delegation set shown in Figure 3.2. Acme grants Bob read-only access with the delegation $(\texttt{Bob} \succcurlyeq \texttt{Acme:Emp}^{\rightarrow}, \texttt{Acme:Emp}^{\leftarrow})$. As before, Bob delegates to Acme's competitor Rival.

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

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

Consider the following judgment, which holds in our example trust configuration.

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

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

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

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

For instance, Figure 3.2 shows Acme's client list labeled with confidentiality $\texttt{Acme:Emp}^{\rightarrow}$. Suppose Bob wants to copy this list to a file with confidentiality $\texttt{Bob}^{\rightarrow}$ so he can maintain access if he is fired. To do so, Acme's system requires that the following judgment holds.

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

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

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

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

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

3.3.4 Speaking for other principals

Prior work on *robust downgrading* [23, 54, 65] of information flow policies places constraints on the influence an attacker may have on declassification and endorsement. Specifically, a principal should not be able to leak information by influencing downgrading decisions. Here, we seek similar constraints, but on information flow authorizations in general, whether they represent a downgrade or not.

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

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

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

As its name suggests, the voice of a principal is related to the *speaks-for* relation [2, 49] found in authorization logics. In these models, if Bob speaks for Alice (sometimes written Bob $\Rightarrow$ Alice) and Bob says some proposition P is true, then Alice also says P is true. Flow-limited judgments permit a refinement of speaks-for since we can

reason directly about the influence of principals on authorization decisions. In FLAM, a principal's voice is the integrity needed to speak on its behalf, so Bob speaks for Alice if $\texttt{Bob} \succcurlyeq \nabla(\texttt{Alice})$.

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

As in [2], FLAM's speaks-for relation distinguishes the concepts of *speaking for* and *acting for* a principal. Previous DIFC models [63] have considered these concepts to be similar, but they are distinct in FLAM to support reasoning separately about the confidentiality and integrity of principals. For instance, the principal $\texttt{Acme}^{\leftarrow}$ speaks for both Acme and $\texttt{Acme}^{\rightarrow}$, but acts for neither.

This distinction is often useful when modeling real systems. For instance, $\texttt{Acme}^{\leftarrow}$ might represent information cryptographically signed with Acme's key, and $\texttt{Acme}^{\rightarrow}$ might represent information encrypted with Acme's key. In such a system, it is clearly useful to be able to distinguish the ability to decrypt a message encrypted with Acme's key from the ability to sign a message on Acme's behalf.

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

3.3.5 Rules for flow-limited reasoning

Figures 3.3 and 3.4 give inference rules for deriving flow-limited judgments. The purpose of these rules is to enforce the security of delegations in the trust configuration

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

$$[\textsc{ProjR}]\quad C \vdash p \succcurlyeq p^{\pi} \qquad [\textsc{Own1}]\quad \frac{C \vdash o \succcurlyeq o' \quad C \vdash p \succcurlyeq p'}{C \vdash o{:}p \succcurlyeq o'{:}p'} \qquad [\textsc{Own2}]\quad \frac{C \vdash o \succcurlyeq o' \quad C \vdash p \succcurlyeq o'{:}p'}{C \vdash o{:}p \succcurlyeq o'{:}p'}$$

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

$$[\textsc{DisjR}]\ \frac{C \vdash p \succcurlyeq p_k \quad k \in \{1,2\}}{C \vdash p \succcurlyeq p_1 \vee p_2} \qquad [\textsc{Trans}]\ \frac{C \vdash p \succcurlyeq q \qquad C \vdash q \succcurlyeq r}{C \vdash p \succcurlyeq r} \qquad [\textsc{Del}]\ \frac{(p \succcurlyeq q, \ell) \in \mathcal{H}(c)}{\mathcal{H}; c; \mathit{pc}; \ell \vdash p \succcurlyeq q}$$

$$[\textsc{Fwd}]\ \frac{\mathcal{H}; c; \mathit{pc}; \ell \Vdash n \succcurlyeq \mathit{pc}^{\rightarrow} \wedge \ell \quad \mathcal{H}; n; \mathit{pc} \sqcup \ell \sqcup c^{\leftarrow}; \ell \sqcap c^{\rightarrow} \vdash p \succcurlyeq q}{\mathcal{H}; c; \mathit{pc}; \ell \vdash p \succcurlyeq q} \qquad [\textsc{Weaken}]\ \frac{\mathcal{H}; c; \mathit{pc}'; \ell' \vdash p \succcurlyeq q \quad \mathcal{H}; c; \mathit{pc} \sqcup \ell'; \ell \Vdash \mathit{pc} \sqsubseteq \mathit{pc}' \quad \mathcal{H}; c; \mathit{pc} \sqcup \ell'; \ell \Vdash \ell' \sqsubseteq \ell}{\mathcal{H} \cup \mathcal{H}'; c; \mathit{pc}; \ell \vdash p \succcurlyeq q}$$

Figure 3.3: Inference rules for flow-limited judgments. For brevity, C denotes the context $\mathcal{H}; c; \mathit{pc}; \ell$. The union of trust configurations is defined pointwise: $(\mathcal{H} \cup \mathcal{H}')(n) = \mathcal{H}(n) \cup \mathcal{H}'(n)$.

and ensure derivations cannot be influenced by attackers. We formalize and verify these properties in Section 3.4.

Figure 3.3 presents rules for non-robust judgments. Most of these rules are straightforward, encoding properties of conjunctions (rules ConjL, ConjR), disjunctions (rules DisjL, DisjR), authority projections (rules Proj and ProjR), ownership projections (rules Own1, Own2), and lattices in general (rules Bot, Top, Refl, Trans). The Del rule allows the use of a local delegation if its label matches the derivation label of the context.

The Weaken rule allows judgment contexts to be weakened. If $p \succcurlyeq q$ is derivable with trust configuration $\mathcal{H}$ and bounds $\mathit{pc}'; \ell'$, then it is still derivable after adding delegations[4] to $\mathcal{H}$ or increasing the restrictiveness of the bounds ($\mathit{pc} \sqsubseteq \mathit{pc}'$ and $\ell' \sqsubseteq \ell$).

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

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

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

The rules for reasoning about robust judgments are shown in Figure 3.4. The first three rules specify how robust judgments derive from non-robust judgments. Rule R-STATIC permits static judgments to be treated as robust judgments in any context, whereas rule R-LIFT derives robust judgments from dynamic judgments. The first premise of R-LIFT ensures the judgment holds with the voice $\nabla(q)$ of the delegating principal. The second premise ensures that principals that speak for p's confidentiality also speak for q's confidentiality[5]. The third premise ensures that the query's context is sufficiently trusted to influence this authorization decision. Rule R-LIFTPC handles judgments regarding the query label as a special case. Rules R-CONJR, R-DISJL, R-WEAKEN, and R-TRANS are similar to their non-robust counterparts but possess robust

$\mathcal{H}(n) \cup \mathcal{H}'(n)$

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

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

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

$$[\text{R-DISJL}] \quad \frac{\begin{array}{c}C \Vdash p_1 \succcurlyeq p \\ C \Vdash p_2 \succcurlyeq p\end{array}}{C \Vdash p_1 \vee p_2 \succcurlyeq p} \qquad [\text{R-TRANS}] \quad \frac{\begin{array}{c}\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q \qquad \mathcal{H}; c; pc; \ell \Vdash q \succcurlyeq r \\ \mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(r^{\rightarrow})\end{array}}{\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq r}$$

$$[\text{R-FWD}] \quad \frac{\begin{array}{c}\mathcal{H}; c; pc; \ell \Vdash n \succcurlyeq pc^{\rightarrow} \wedge \ell \wedge \nabla(q) \\ \mathcal{H}; n; pc \sqcup \ell \sqcup c^{\leftarrow}; \ell \sqcap c^{\rightarrow} \Vdash p \succcurlyeq q\end{array}}{\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q} \qquad [\text{R-WEAKEN}] \quad \frac{\begin{array}{c}\mathcal{H}; c; pc'; \ell' \Vdash p \succcurlyeq q \\ \mathcal{H}; c; pc \sqcup \ell'; \ell \Vdash pc \sqsubseteq pc' \\ \mathcal{H}; c; pc \sqcup \ell'; \ell \Vdash \ell' \sqsubseteq \ell\end{array}}{\mathcal{H} \cup \mathcal{H}'; c; pc; \ell \Vdash p \succcurlyeq q}$$

Figure 3.4: Inference rules for robust judgments.

premises. Rule R-TRANS adds a query label restriction to TRANS to ensure that the query's context speaks for r. Likewise, R-FWD adds the restriction that remote principals must speak for the principal that the judgment concerns.

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

3.4 Robust authorization

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

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

$$\mathcal{H}; c; \mathit{pc}; \ell \Vdash \ell' \vee N \sqsubseteq \ell \tag{3.1}$$

$$\mathcal{H}; c; \mathit{pc}; \ell \Vdash N \succcurlyeq \mathit{pc}^{\rightarrow} \wedge \ell^{\leftarrow} \tag{3.2}$$

$$k > 0 \Rightarrow \ \mathcal{H}; c; \mathit{pc}; \ell \Vdash c \succcurlyeq (\ell' \vee N)^{\rightarrow} \tag{3.3}$$

Proof. By induction on the derivation of $\mathcal{H}; c; \mathit{pc}; \ell \vdash p \succcurlyeq q$. Verified in Coq [7]. □

The guarantees robust authorization bestows on authorization queries are quite strong. Remote principals cannot exceed their authority to influence the derivation, despite having the power to create arbitrary delegations and participate in the derivation itself. In particular, the authorization mechanism preserves the end-to-end security of each delegation's information flow policy ℓ' (3.1) while preserving the confidentiality $\mathit{pc}^{\rightarrow}$ of the query and the integrity $\ell^{\leftarrow}$ of the result (3.2), and without leaking confidential information to c (3.3). Conclusion (3.3) only applies to distributed derivations (where $k > 0$) since we permit a node to use a local delegation without requiring proof that it acts for the confidentiality of the delegation label.

FLAM derivations therefore never require unsafe communication: every remote node that participates in a derivation must robustly act for the confidentiality $\mathit{pc}^{\rightarrow}$ of

the query and integrity $\ell^{\leftarrow}$ of the result. Results are received by c only if c is permitted to learn (implicitly) that c acts for $(\ell' \vee N)^{\rightarrow}$. Because FLAM makes no assumptions about the relationship between n and ℓ', the disjunction N limits the authority of ℓ' to be no greater than the nodes in the derivation, ensuring that malicious delegations do not influence the derivation beyond the authority of these nodes. From the perspective of confidentiality, the disjunction also ignores information flows in which the claimed confidentiality of the delegation label exceeds the confidentiality authority of nodes providing the delegation; ignoring such flows makes sense because confidentiality is enforced by the providers, not by the recipient c.

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

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

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

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

Proof. By induction on the derivation of $\mathcal{H}; c; \mathit{pc}; \ell \vdash p \succcurlyeq q$. Verified in Coq [7]. □

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

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

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

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

Theorem 1 and Lemma 1 prove that attackers cannot use delegation and revocation to interfere with authorization queries, eliminating the delegation loophole (Section 2.1) and poaching attacks (Section 2.2). New delegations cannot cause unsafe communication to occur or cause existing delegations to be disclosed (Section 2.3) unless the new delegations are sufficiently trusted. Furthermore, this result serves as a useful guide to developers of DIFC systems and languages: supporting delegation and revocation while

enforcing information flow policies requires *all* relabeling of policies to be robust—otherwise, changes in the trust configuration could be exploited to create new flows.

We formalized FLAM principals and our inference rules for deriving flow-limited judgments in Coq, and used this formalization to prove Theorem 1 and Lemma 1. We make one primary assumption, that principals that statically act for each other are equivalent. We believe this assumption can be avoided with some refactoring, which we leave as future work.

3.5 FLAM prototype

We have demonstrated that FLAM can be used to provide robust authorization in realistic authorization mechanisms by developing a prototype implementation and using it to implement ARBAC97 [76], an expressive role-based access control model. Our version of ARBAC97 uses owned principals to represent roles and extends the strong security guarantees of FLAM to role-based access control; for example, untrusted users cannot use authorization queries to infer the secret membership of roles. Our prototype currently only uses rules R-LIFT and R-LIFTPC for reasoning about robust judgments, but these were sufficient for our purposes.

3.5.1 Efficient flow-limited query processing

Our FLAM prototype answers acts-for queries through a proof search; the relationship being queried is said to hold exactly when a proof of the relationship can be found. This proof search is NP-hard: a FLAM query can encode any 3-SAT problem. To see this, for any 3-SAT CNF formula $\mathcal{F}$, let $\mathsf{Lit}(\mathcal{F})$ be the set of literals in $\mathcal{F}$. Choose the set of primitive principals $\mathcal{N}$ such that $a \in \mathsf{Lit}(\mathcal{F})$ implies that $a, \neg a \in \mathcal{N}$. That

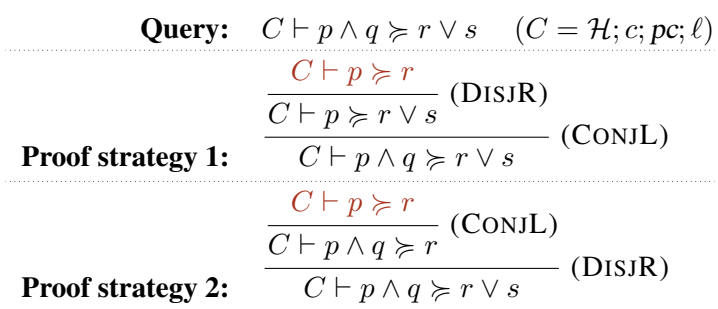


Figure 3.5: Redundant work in the basic search algorithm. If the query is not provable, an exhaustive proof search must be made before a negative result can be returned. Here, both CONJL and DISJR apply, so the search will try both proof strategies shown. Without caching, redundant proof searches would be made for the two identical premises shown in red.

is, every literal in $\mathcal{F}$ is represented by two principals in $\mathcal{N}$: one for the literal and one for its negation. Choose $\mathcal{H}$ with a single delegation at host c whose disjuncts encode all possible assignments to the literals in $\mathcal{F}$. This delegation has the form $\langle \bot \succcurlyeq (a \wedge b \wedge c \wedge \ldots) \vee (\neg a \wedge b \wedge c \wedge \ldots), \top^{\leftarrow} \rangle$. In each disjunction of the delegation, a literal or its negation appears at most once, encoding an assignment of 1 or 0 for that literal. Use $\mathcal{F}$ to create a FLAM query $\mathcal{H}; c; pc; \ell \vdash \bot \succcurlyeq \mathcal{F}$. Since $\mathcal{F}$ has the form $(a \vee \neg b \vee c) \wedge (\ldots)$, $\bot \succcurlyeq \mathcal{F}$ if and only if at least one disjunct (that is, a or $\neg b$ or c) in every conjunct of $\mathcal{F}$ appears in a single disjunct of the delegation in $\mathcal{H}$. Therefore, finding a proof for the judgment $\mathcal{H}; c; pc; \ell \vdash \bot \succcurlyeq \mathcal{F}$ requires finding a disjunct that represents a satisfying assignment for the literals in $\mathcal{F}$. If no such proof exists, then there is no satisfying assignment.

In practice, however, we expect most queries and trust configurations to be relatively small, making proof search tractable for most applications. For the purposes of presenting our algorithm, we assume that the trust configuration does not change during the proof search; in practice, query isolation can be provided by existing mechanisms for distributed transactions (for example, [54]). The basic proof-search algorithm is a simple depth-first search with cycle detection. It returns two types of results: `PROVED` (which comes with a proof) and `FAILED`.

(a) A pruned proof strategy

(b) A successful proof strategy

Figure 3.6: Proof diagrams showing two strategies for proving a query. Nodes represent premises. Edges represent proof dependencies; unexplored edges are dotted. In strategy (a), the proof search for the blue node is pruned because its proof depends on the red node, which would introduce a cycle in the proof diagram. Strategy (b) results in a successful proof: the proof forms a DAG, wherein all leaf nodes are axioms.

This algorithm alone performs poorly, however, owing to much duplicated work. Queries with `FAILED` results are particularly expensive, since they require a full exhaustive proof search. For example, in Figure 3.5, if the query $C \vdash p \wedge q \succcurlyeq r \vee s$ is unprovable, the algorithm must explore all possible proof strategies, including using ConjL and DisjR, as shown. Both of these strategies have the unprovable subquery $C \vdash p \succcurlyeq r$, shown in red. Without caching, redundant proof searches would be made for these identical subqueries. Furthermore, caching only positive results would not significantly improve the performance of unprovable queries.

Naively caching intermediate negative results can lead to incompleteness due to searches that are *pruned* to avoid infinite recursion and circular reasoning. Figure 3.6 illustrates this using proof diagrams. Nodes represent premises to be proved, and edges represent their dependencies. Unexplored edges are dotted. In the first proof strategy (Figure 3.6a), the proof of the blue node is pruned to avoid circular reasoning with the red node. While it would be sound to cache a `FAILED` result for the blue node, doing so would be incomplete. When the proof search later attempts the second proof strategy (Figure 3.6b), it finds a successful proof for the red node via the green node. With a cached `FAILED` result for the blue node, the proof of the white node would simply use the cached result, failing to notice that because the circularity with the red node has been resolved, the blue node can now be proved.

To prevent this incompleteness, our implementation of FLAM uses an intermediate caching strategy for pruned results. Instead of caching `FAILED` for pruned subqueries, we introduce an additional result type, `PRUNED`. When a cycle is detected during proof search, the current subproof is abandoned and the subquery is added to a cache of pruned queries. Each `PRUNED` cache entry contains a *progress condition*, a boolean formula that expresses the conditions under which further progress can be made on the proof of the subquery. In Figure 3.6, the first proof strategy would result in a `PRUNED` cache entry for the blue subquery, with the progress condition $Q = \bullet$, indicating that further progress can be made on the proof of the blue node exactly when the red node can be proved. Another progress condition might have the form $Q_1 \vee (Q_2 \wedge Q_3)$, meaning that progress can be made if Q_1 is proved or if both Q_2 and Q_3 are proved.

This cache is used by the proof search to improve performance when resolving shared subqueries. The cache has three components: an *acts-for cache* for proofs of `PROVED` subqueries, a *failed cache* for `FAILED` subqueries, and a *pruned-search cache* for `PRUNED` subqueries and their progress conditions. Figure 3.7 gives the algorithm for updating the cache with a new result for a subquery $query$. At the core of this algorithm is the rewriting of progress conditions in the pruned-search cache. If the new result is `PROVED`, the progress conditions are rewritten to substitute instances of $query$ with `True` (line 7), to indicate that the $query$ condition is satisfied. If this satisfies the progress condition of a pruned search q, then q should be provable, and is removed from the cache (lines 8–9); a `PROVED` entry is not added for q yet because we do not yet have a proof. If the new result is `PRUNED`, then instances of $query$ are substituted with $query$'s progress condition (line 15). Finally, if the new result is `FAILED`, then instances of $query$ are substituted with `False` (line 21), to indicate that the $query$ condition is not satisfiable. If the progress condition of a pruned search q becomes unsatisfiable, then q is also unprovable, and the cache is updated with a `FAILED` result for q (lines 27–28).

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

Figure 3.7: Algorithm for managing entries of the proof search cache. For $type$ equal to PROVED or PRUNED, $data$ is either a proof of $query$ or a progress condition, respectively.

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

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

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

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

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

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

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

Figure 3.8: User–role assignment. The FLAM judgments ensure a is a member of the administrative role ar, that u meets criteria cr (in Figure 3.8a), and that r is in the range $[mn, mx]$. Each judgment requires the integrity of ar to ensure only administrators influence role management.

query is false.

The complete search algorithm is found in Appendix A.1.

3.5.2 Example: ARBAC97 access control

To demonstrate the expressiveness of FLAM and the functionality of our implementation, we have adapted the ARBAC97 role-based access control model for role management [76] using our FLAM implementation. The implementation required only 242 lines of code, showing that FLAM is already quite expressive. Using FLAM means

our implementation of ARBAC97 also enjoys stronger security properties; in particular, robust authorization means that attackers can neither influence the membership of roles nor learn anything about confidential role assignments.

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

The key difficulty in representing ARBAC's role-management authorization policies is in the separation between management authority and role membership. FLAM simplifies the ARBAC model since administrative roles, roles, users, and permissions may all be represented as principals. This allows the unification of the three relations UA, PA, and RH into a single trust configuration $\mathcal{H}$. Our version of ARBAC97, adapted from the formalization presented in [86], leverages FLAM's information flow tracking and expressive principal algebra to preserve the separation of management authority and role membership in $\mathcal{H}$.

In ARBAC, the authorization criteria for making modifications to the trust configuration are defined by additional relations[6]. The relations *can_assign* and *can_revoke* encode policies for user–role assignment. Entries of *can_assign* are tuples of principals (ar, c, mn, mx), where ar represents an *administrative role*, cr represents some *minimal criteria*[7] that users must meet to be assigned the role, and $[mn, mx]$ represents a range that bounds the role assignments ar is permitted to make. Entries of can_revoke are tuples of principals (ar, mn, mx) which are similar to those of *can_assign*, but have no

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

[7]In [76], criteria are more general, allowing cr to specify both roles that a user *must* have, as well as roles a user *must not* have. By separating positive and negative criteria we can represent the general case in FLAM, but for simplicity of exposition we omit negative criteria.

minimal criteria.

FLAM strengthens the guarantees of ARBAC97 by tracking information flow on modifications to trust configuration and ensuring robust authorization. Figures 3.8a and 3.8b illustrate our encoding of user–role assignment authorization. Each method includes a parameter *pc* that represents the information flow context of the caller, and a label ℓ for specifying the confidentiality and integrity of the role assignment. In Figure 3.8a, the assignment of user u to role r by administrator a is authorized if there is an entry in the can_assign relation such that the subsequent FLAM judgments hold robustly with the integrity of ar. The first judgment ensures a is a member of the ar role. The second judgment ensures that the user acts for a principal representing some minimal criteria. Finally, the third and fourth judgments ensure r is within the range $[mn, mx]$.

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

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

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

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

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

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

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

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

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

Figure 3.9: Permission–role assignment

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

The remaining methods for permission–role management and role–role management share many similarities with the above methods for user–role management. Figure 3.9 defines methods for permission–role management, Figure 3.10 defines range assignment methods, and Figure 3.11 defines role–role assignment methods.

Our implementation suggests a general approach for extending robust authorization to traditional access control models. Translating the *authority* implied by the ARBAC97 roles to FLAM trust relationships allow FLAM queries to securely implement ARBAC authorization requests without creating authorization side channels. Coupling this translation with specialized role management code yields a more secure access control sys-

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

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

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

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

Figure 3.10: Range assignment functions

tem. This exercise demonstrates the expressiveness of FLAM policies as well as the effectiveness of the implemented algorithm; we expect other access control systems could be enhanced in a similar way.

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

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

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

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

Figure 3.11: Role–role assignment functions

CHAPTER 4

A CALCULUS FOR FLOW-LIMITED AUTHORIZATION

4.1 Dynamic authorization mechanisms

Dynamic authorization is challenging to implement and use correctly, since authority, confidentiality, and integrity interact in subtle ways. This chapter presents the Flow-Limited Authorization Calculus (FLAC), which helps programmers securely implement both authorization mechanisms and code that uses them. FLAC types support the definition of compositional security abstractions, and vulnerabilities in the implementations of these abstractions are caught statically. Further, the guarantees offered by FLAC simplify reasoning about the security properties of these abstractions.

We illustrate the usefulness and expressive power of FLAC using two important security mechanisms: commitment schemes and bearer credentials. We show in Section 4.4 that these mechanisms can be implemented using FLAC, and that their security goals are easily verified in the context of FLAC.

4.1.1 Commitment schemes

A commitment scheme [67] allows one party to give another party a "commitment" to a secret value without revealing the value. The committing party may later reveal the secret in a way that convinces the receiver that the revealed value is the value originally committed.

Commitment schemes provide three essential operations: `commit`, `receive`, and `open`. Suppose p wants to commit to a value to principal q. First, p applies `commit` to the value and provides the result to q. Next, q applies `receive` to the committed value. Finally, when p wishes to reveal the value, p applies the `open` operation to the received value, permitting q to learn it.

A commitment scheme must have several properties in order to be secure. First, q should not be able to receive a value that hasn't been committed by p, since this could allow q to manipulate p to `open` a value it had not committed to. Second, q should not learn any secret of p that has not been opened by p. Third, p should not be able to `open` a different value than the one received by q.

One might wonder why a programmer would bother to create high-level *implementations* of operations like `commit`, `receive`, and `open`. Why not simply treat these as primitive operations and give them type signatures so that programs using them can be type-checked with respect to those signatures? The answer is that an error in a type signature could lead to a serious vulnerability. Therefore, we want more assurance that the type signatures are correct. Implementing such operations in FLAC is often easy and ensures that the type signature is consistent with a set of assumptions about existing trust relationships and the information flow context the operations are used within. These FLAC-based implementations serve as language-based models of the security properties achieved by implementations that use cryptography or trusted third parties.

4.1.2 Bearer credentials with caveats

A bearer credential is a capability that grants authority to any entity that possesses it. Many authorization mechanisms used in distributed systems employ bearer credentials in some form. Browser cookies that store session tokens are one example: after a website authenticates a user's identity, it gives the user a token to use in subsequent interactions. Since it is infeasible for attackers to guess the token, the website grants the authority of the user to any requests that include the token.

Bearer credentials create an information security conundrum for authorization mechanisms. Though they efficiently control access to restricted resources, they create vulnerabilities and introduce covert channels when used incorrectly. For example, suppose

Alice shares a remotely-hosted photo with her friends by giving them a credential to access the photo. Giving a friend such a credential doesn't disclose their friendship, but each friend that accesses the photo implicitly discloses the friendship to the hosting service. Such covert channels are pervasive, both in classic distributed authorization mechanisms like SPKI/SDSI [31], as well as in more recent ones like Macaroons [13].

Bearer credentials can also lead to vulnerabilities if they are leaked. If an attacker obtains a credential, it can exploit the authority of the credential. Thus, to limit the authority of a credential, approaches like SPKI/SDSI and Macaroons provide *constrained delegation* in which a newly issued credential attenuates the authority of an existing one by adding *caveats*. Caveats require additional properties to hold for the bearer to be granted authority. Session tokens, for example, might have a caveat that restricts the source IP address or encodes an expiration time. As pointed out by Birgisson et al. [13], caveats themselves can introduce covert channels if the properties reveal sensitive information.

FLAC is an effective framework for reasoning about bearer credentials with caveats since it captures the flow of credentials in programs as well as the sensitivity of the information the credentials and caveats derive from. We can reason about credentials and the programs that use them in FLAC with an approach similar to that used for commitment schemes. That we can do so in a straightforward way is somewhat remarkable; prior formalizations of credential mechanisms (for example, [12, 13, 40]) usually do not consider confidentiality nor provide end-to-end guarantees about credential propagation.

4.2 The FLAM principal lattice

In FLAC, we use a simplified version of the FLAM principal lattice and inference rules introduced in Chapter 3 to express authority and information flow policies, which we briefly review here. For FLAC, it is convenient to define static and dynamic rules

$\boxed{\mathcal{L} \vDash p \succcurlyeq q}$

$$[\textsc{Bot}]\ \ \mathcal{L} \vDash p \succcurlyeq \bot \qquad [\textsc{Top}]\ \ \mathcal{L} \vDash \top \succcurlyeq p \qquad [\textsc{Refl}]\ \ \mathcal{L} \vDash p \succcurlyeq p \qquad [\textsc{Proj}]\ \ \frac{\mathcal{L} \vDash p \succcurlyeq q}{\mathcal{L} \vDash p^{\pi} \succcurlyeq q^{\pi}}$$

$$[\textsc{ProjR}]\ \ \mathcal{L} \vDash p \succcurlyeq p^{\pi} \qquad [\textsc{ConjL}]\ \ \frac{\mathcal{L} \vDash p_k \succcurlyeq p \quad k \in \{1,2\}}{\mathcal{L} \vDash p_1 \wedge p_2 \succcurlyeq p} \qquad [\textsc{ConjR}]\ \ \frac{\mathcal{L} \vDash p \succcurlyeq p_1 \quad \mathcal{L} \vDash p \succcurlyeq p_2}{\mathcal{L} \vDash p \succcurlyeq p_1 \wedge p_2}$$

$$[\textsc{DisjL}]\ \ \frac{\mathcal{L} \vDash p_1 \succcurlyeq p \quad \mathcal{L} \vDash p_2 \succcurlyeq p}{\mathcal{L} \vDash p_1 \vee p_2 \succcurlyeq p} \qquad [\textsc{DisjR}]\ \ \frac{\mathcal{L} \vDash p \succcurlyeq p_k \quad k \in \{1,2\}}{\mathcal{L} \vDash p \succcurlyeq p_1 \vee p_2} \qquad [\textsc{Trans}]\ \ \frac{\mathcal{L} \vDash p \succcurlyeq q \qquad \mathcal{L} \vDash q \succcurlyeq r}{\mathcal{L} \vDash p \succcurlyeq r}$$

Figure 4.1: Static principal lattice rules, adapted from FLAM. The projection π may be either confidentiality ($\rightarrow$) or integrity ($\leftarrow$).

$\boxed{\Pi; pc; \ell \Vdash p \succcurlyeq q}$

$$[\text{R-}\textsc{Static}]\frac{\mathcal{L} \vDash p \succcurlyeq q}{\Pi; pc; \ell \Vdash p \succcurlyeq q} \qquad [\text{R-}\textsc{Assume}]\frac{\langle p \succcurlyeq q \mid \ell \rangle \in \Pi}{\Pi; pc; \ell \Vdash p \succcurlyeq q} \qquad [\text{R-}\textsc{ConjR}]\frac{\Pi; pc; \ell \Vdash p \succcurlyeq p_1 \quad \Pi; pc; \ell \Vdash p \succcurlyeq p_2}{\Pi; pc; \ell \Vdash p \succcurlyeq p_1 \wedge p_2}$$

$$[\text{R-}\textsc{DisjL}]\ \ \frac{\Pi; pc; \ell \Vdash p_1 \succcurlyeq p \quad \Pi; pc; \ell \Vdash p_2 \succcurlyeq p}{\Pi; pc; \ell \Vdash p_1 \vee p_2 \succcurlyeq p} \qquad [\text{R-}\textsc{Trans}]\ \ \frac{\Pi; pc; \ell \Vdash p \succcurlyeq q \qquad \Pi; pc; \ell \Vdash q \succcurlyeq r \quad \Pi; pc; \ell \Vdash pc \succcurlyeq \nabla(r^{\rightarrow})}{\Pi; pc; \ell \Vdash p \succcurlyeq r}$$

$$[\text{R-}\textsc{Weaken}]\ \ \frac{\Pi; pc'; \ell' \Vdash p \succcurlyeq q \qquad \Pi; pc \sqcup \ell'; \ell \Vdash \ell' \sqsubseteq \ell \qquad \Pi; pc \sqcup \ell'; \ell \Vdash pc \sqsubseteq pc'}{\Pi \cup \Pi'; pc; \ell \Vdash p \succcurlyeq q}$$

Figure 4.2: Inference rules for robust assumption, adapted from FLAM.

separately. The static inference rules, presented in Figure 4.1, reason about relationships between principals arising from the structure of the lattice, and are thus present in any context. For reasoning about dynamic relationships between principals, we adapt FLAM's robust inference rules, presented in Figure 4.2. For simplicity, FLAC omits ownership projections.

The delegation context, Π, is analogous to FLAM's trust configuration $\mathcal{H}$, but with two primary differences. First, Π encodes static reasoning about dynamic relationships in program, so all delegations are local in the sense that no communication is necessary

to obtain them. For this reason, the current host is omitted from the context of the judgements in Figure 4.2. Second, unlike $\mathcal{H}$, the type system preserves the invariant that all delegations in Π are robust. This invariant is leveraged by the ASSUME rule, which is a robust version of FLAM's DEL rule.

4.3 Flow-Limited Authorization Calculus

FLAC uses information flow to reason about the security implications of dynamically computed authority. Like previous information-flow type systems [75], FLAC incorporates types for reasoning about information flow, but FLAC's type system goes further by using flow-limited authorization to ensure that principals cannot use FLAC programs to exceed their authority, or to leak or corrupt information. FLAC is based on DCC [2], but unlike DCC, FLAC supports reasoning about authority deriving from the evaluation of FLAC terms. In contrast, all authority in DCC derives from trust relationships defined by a fixed, external lattice of principals. Thus, using an approach based on DCC in systems where trust relationships change dynamically could introduce vulnerabilities like delegation loopholes, probing and poaching attacks, and authorization side channels.

Figure 4.3 defines the FLAC syntax; evaluation contexts [94] are defined in Figure 4.4. The core operational semantics in Figure 4.5 is mostly standard except for `assume` terms, discussed below.

The core FLAC type system is presented in Figure 4.6. Programs that type check under these rules are guaranteed to enforce the information flow policies of the information they process. We formalize the semantics of this security guarantee in Section 4.6, and prove that it holds for all well-typed FLAC programs.

FLAC typing judgments have the form $\Pi; \Gamma; pc \vdash e : s$. The *delegation context*, Π, contains a set of labeled dynamic trust relationships $\langle p \succcurlyeq q \mid \ell \rangle$ where $p \succcurlyeq q$ (read

$n \in \mathcal{N}$ (primitive principals)
$x \in \mathcal{V}$ (variable names)

$$
\begin{array}{rcl}
p, \ell, \mathit{pc} & ::= & n \mid \top \mid \bot \mid p^{\rightarrow} \mid p^{\leftarrow} \mid p \wedge p \mid p \vee p \\
s & ::= & (p \succcurlyeq p) \mid \mathtt{unit} \mid (s + s) \mid (s \times s) \\
& & \mid s \xrightarrow{\mathit{pc}} s \mid \ell \ \mathtt{says}\ s \mid X \mid \forall X.\, s \\
v & ::= & () \mid \langle v, v \rangle \mid \langle p \succcurlyeq p \rangle \mid (\eta_\ell\ v) \\
& & \mid \mathtt{inj}_i\ v \mid \lambda(x\!:\!s)[\mathit{pc}].\, e \mid \Lambda X.\, e \\
& & \mid v\ \mathtt{where}\ v \\
e & ::= & x \mid v \mid e\ e \mid \langle e, e \rangle \mid (\eta_\ell\ e) \\
& & \mid es \mid \mathtt{proj}_i\ e \mid \mathtt{inj}_i\ e \\
& & \mid \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inj}_1(x).\, e \mid \mathtt{inj}_2(x).\, e \\
& & \mid \mathtt{bind}\ x = e\ \mathtt{in}\ e \mid \mathtt{assume}\ e\ \mathtt{in}\ e \\
& & \mid e\ \mathtt{where}\ v
\end{array}
$$

Figure 4.3: FLAC syntax. Terms using `where` are syntactically prohibited in the source language and are produced only during evaluation.

$$
\begin{array}{rcl}
E & ::= & [\cdot] \mid E\ e \mid v\ E \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \mathtt{proj}_i\ E \mid \mathtt{inj}_i\ E \\
& & \mid (\eta_\ell\ E) \mid \mathtt{bind}\ x = E\ \mathtt{in}\ e \mid \mathtt{bind}\ x = v\ \mathtt{in}\ E \\
& & \mid Es \mid \mathtt{assume}\ E\ \mathtt{in}\ e \mid E\ \mathtt{where}\ v \\
& & \mid \mathtt{case}\ E\ \mathtt{of}\ \mathtt{inj}_1(x).\, e \mid \mathtt{inj}_2(x).\, e
\end{array}
$$

Figure 4.4: FLAC evaluation contexts

as "p acts for q") is a delegation from q to p , and ℓ is the confidentiality and integrity of that information. The *typing context*, Γ, is a map from variables to types, and *pc* is the *program counter label*, a FLAM principal representing the confidentiality and integrity of control flow. The type system makes frequent use of judgments adapted from FLAM's inference rules. The query and derivation labels of all FLAM judgements used by the type system are equal. This reflects the design of the delegation context: the label on delegations in Π is always bound by the current program counter label.

Since FLAC is a pure functional language, it might seem odd for FLAC to have a label for the program counter; such labels are usually used to control implicit flows

$\boxed{e \longrightarrow e'}$

[E-APP] $(\lambda(x\!:\!s)[\mathit{pc}].\, e)\ v \longrightarrow e[x \mapsto v]$ [E-TAPP] $(\Lambda X.\, e)\ s \longrightarrow e[X \mapsto s]$

[E-UNPAIR] $\mathtt{proj}_i\ \langle v_1, v_2\rangle \longrightarrow v_i$

[E-CASE] $(\mathtt{case}\ (\mathtt{inj}_1\ v)\ \mathtt{of}\ \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2) \longrightarrow e_i[x \mapsto v]$

[E-BINDM] $\mathtt{bind}\ x = (\eta_\ell\ v)\ \mathtt{in}\ e \longrightarrow e[x \mapsto v]$

[E-ASSUME] $\mathtt{assume}\ \langle p \succcurlyeq q\rangle\ \mathtt{in}\ e \longrightarrow e\ \mathtt{where}\ \langle p \succcurlyeq q\rangle$ [E-EVAL] $\dfrac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$

Figure 4.5: FLAC operational semantics

through assignments (for example, in [62, 70]). The purpose of FLAC's *pc* label is to control a different kind of side effect: changes to the delegation context, Π.[1] In order to control what information can influence whether a new trust relationship is added to the delegation context, the type system tracks the confidentiality and security of control flow. Viewed as an authorization logic, FLAC's type system expresses deduction constrained by an information flow context, a unique feature it derives from FLAM. For instance, if we have $\varphi \xrightarrow{p^{\leftarrow}} \psi$ and φ, then (via APP) we may derive ψ in a context with integrity $p^{\leftarrow}$, but not in contexts that don't flow to $p^{\leftarrow}$. This feature offers needed control over how principals may apply existing facts to derive new facts.

Many FLAC terms are standard, such as pairs $\langle e_1, e_2\rangle$, projections $\mathtt{proj}_i\ e$, variants $\mathtt{inj}_i\ e$, polymorphic type abstraction, $\Lambda X.\, e$, and case expressions. Function abstraction, $\lambda(x : s)[\mathit{pc}].\, e$, includes a *pc label* that constrains the information flow context in which the function may be applied. The rule APP ensures that function application respects these policies, requiring that the robust FLAM judgment $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \sqsubseteq \mathit{pc}'$ holds. This judgment ensures that the current program counter label, *pc*, flows to the function label, *pc*′.

[1] The same *pc* label could also be used to control implicit flows through assignments if FLAC were extended to support mutable references.

$\boxed{\Pi; \Gamma; pc \vdash e : s}$

[VAR] $\Pi; \Gamma, x : s, \Gamma'; pc \vdash x : s$ [UNIT] $\Pi; \Gamma; pc \vdash () : \texttt{unit}$ [DEL] $\Pi; \Gamma; pc \vdash \langle p \succcurlyeq q \rangle : (p \succcurlyeq q)$

[LAM] $$\frac{\Pi; \Gamma, x : s_1; pc' \vdash e : s_2}{\Pi; \Gamma; pc \vdash \lambda(x : s_1)[pc']. e : (s_1 \xrightarrow{pc'} s_2)}$$

[APP] $$\frac{\Pi; \Gamma; pc \vdash e : (s_1 \xrightarrow{pc'} s_2) \quad \Pi; \Gamma; pc \vdash e' : s_1 \quad \Pi; pc; pc \Vdash pc \sqsubseteq pc'}{\Pi; \Gamma; pc \vdash (e\ e') : s_2}$$

[TLAM] $$\frac{\Pi; \Gamma, X; pc' \vdash e : s}{\Pi; \Gamma; pc \vdash \Lambda X. e : \forall X. s}$$

[TAPP] $$\frac{\Pi; \Gamma; pc \vdash e : \forall X. s}{\Pi; \Gamma; pc \vdash (e s') : s[X \mapsto s']}$$ s' well-formed in Γ

[PAIR] $$\frac{\Pi; \Gamma; pc \vdash e_1 : s_1 \quad \Pi; \Gamma; pc \vdash e_2 : s_2}{\Pi; \Gamma; pc \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$$

[UNPAIR] $$\frac{\Pi; \Gamma; pc \vdash e : (s_1 \times s_2)}{\Pi; \Gamma; pc \vdash (\texttt{proj}_i\ e) : s_i}$$

[INJ] $$\frac{\Pi; \Gamma; pc \vdash e : s_i}{\Pi; \Gamma; pc \vdash (\texttt{inj}_i\ e) : (s_1 + s_2)}$$

[CASE] $$\frac{\Pi; \Gamma; pc \vdash e : (s_1 + s_2) \quad \Pi; pc \vdash pc \leq s \quad \Pi; \Gamma, x : s_1; pc \vdash e_1 : s \quad \Pi; \Gamma, x : s_2; pc \vdash e_2 : s}{\Pi; \Gamma; pc \vdash \texttt{case}\ e\ \texttt{of}\ \texttt{inj}_1(x).\ e_1 \mid \texttt{inj}_2(x).\ e_2 : s}$$

[UNITM] $$\frac{\Pi; \Gamma; pc \vdash e : s}{\Pi; \Gamma; pc \vdash (\eta_\ell\ e) : \ell\ \texttt{says}\ s}$$

[BINDM] $$\frac{\Pi; \Gamma; pc \vdash e : \ell\ \texttt{says}\ s' \quad \Pi; \Gamma, x : s'; pc \sqcup \ell \vdash e' : s \quad \Pi; pc \vdash pc \sqcup \ell \leq s}{\Pi; \Gamma; pc \vdash \texttt{bind}\ x = e\ \texttt{in}\ e' : s}$$

[ASSUME] $$\frac{\Pi; \Gamma; pc \vdash e : (p \succcurlyeq q) \quad \Pi; pc; pc \Vdash pc \succcurlyeq \nabla(q) \quad \Pi; pc; pc \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \quad \Pi; pc \vdash pc \leq s \quad \Pi, \langle p \succcurlyeq q \mid pc \rangle; \Gamma; pc \vdash e' : s}{\Pi; \Gamma; pc \vdash \texttt{assume}\ e\ \texttt{in}\ e' : s}$$

[WHERE] $$\frac{\Pi; \Gamma; pc \vdash v : (p \succcurlyeq q) \quad \Pi; pc'; pc' \Vdash pc' \sqsubseteq pc \quad \Pi; pc'; pc' \Vdash pc' \succcurlyeq \nabla(q) \quad \Pi; pc'; pc' \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \quad \Pi; pc' \vdash pc' \leq s \quad \Pi, \langle p \succcurlyeq q \mid pc' \rangle; \Gamma; pc' \vdash e : s}{\Pi; \Gamma; pc \vdash (e\ \texttt{where}\ v) : s}$$

Figure 4.6: FLAC type system.

Branching occurs in `case` expressions, which conditionally evaluate one of two expressions. The rule CASE ensures that both expressions have the same type and thus the same protection level. The premise $\Pi; pc \vdash pc \leq s$ ensures that this type protects the current *pc* label.[2]

Like DCC, FLAC uses monadic operators to track dependencies. The monadic unit term $(\eta_\ell\ v)$ (UNITM) says that a value v of type s is *protected at level* ℓ. This protected value has the type ℓ `says` s, meaning that it has the confidentiality and integrity of principal ℓ. Computation on protected values must occur in a protected context ("in the monad"), expressed using a monadic bind term. The typing rule BINDM ensures that the result of the computation protects the confidentiality and integrity of protected values. For instance, the expression `bind` $x = (\eta_\ell\ v)$ `in` $(\eta_{\ell'}\ x)$ is only well-typed if ℓ' protects values with confidentiality and integrity ℓ. Since `case` expressions may use the variable x for branching, BINDM raises the *pc* label to $pc \sqcup \ell$ to conservatively reflect the control-flow dependency.

Protection levels are defined by the set of inference rules in Figure 4.8, adapted from [87]. Expressions with unit type (P-UNIT) do not propagate any information, so they protect information at any ℓ. Product types protect information at ℓ if both components do (P-PAIR). Function types protect information at ℓ if the return type does (P-FUN), and polymorphic types protect information at whatever level the abstracted type does (P-TFUN). If a type s already protects information at ℓ, then ℓ' `says` s still does (P-LBL1). Finally, if ℓ flows to ℓ', then ℓ' `says` s protects information at ℓ (P-LBL2).

Most of the novelty of FLAC lies in its delegation values and `assume` terms. These terms enable expressive reasoning about authority and information flow control. A delegation value serves as evidence of trust. For instance, the term $\langle p \succcurlyeq q \rangle$, read "$p$ acts for

[2]This premise simplifies our proofs, but does not appear to be strictly necessary; BINDM ensures the same property.

q", is evidence that q trusts p. Delegation values have *acts-for types*; $\langle p \succcurlyeq q \rangle$ has type $(p \succcurlyeq q)$. [3] The `assume` term enables programs to use evidence securely to create new flows between protection levels. In the typing context $\varnothing; x : p^{\leftarrow}$ `says` $s; q^{\leftarrow}$ (specifically, $\Pi = \varnothing$, $\Gamma = x : p^{\leftarrow}$ `says` s, and $pc = q^{\leftarrow}$), the following expression is not well typed:

$$\texttt{bind}\ x' = x\ \texttt{in}\ (\eta_{q^{\leftarrow}}\ x')$$

since $p^{\leftarrow}$ does not flow to $q^{\leftarrow}$, as required by the premise $\Pi; pc \vdash \ell \leq s$ in rule BINDM. Specifically, we cannot derive $\Pi; pc \vdash p^{\leftarrow} \leq q^{\leftarrow}$ `says` s since P-LBL2 requires the FLAM judgment $\Pi; q^{\leftarrow}; q^{\leftarrow} \Vdash p^{\rightarrow} \sqsubseteq q^{\leftarrow}$ to hold.

However, the following expression is well typed:

$$\texttt{assume}\ \langle p^{\leftarrow} \succcurlyeq q^{\leftarrow} \rangle\ \texttt{in}\ \texttt{bind}\ x' = x\ \texttt{in}\ (\eta_{q^{\leftarrow}}\ x')$$

The difference is that the `assume` term adds a trust relationship, represented by an expression with an acts-for type, to the delegation context. In this case, the expression $\langle p^{\leftarrow} \succcurlyeq q^{\leftarrow} \rangle$ adds a trust relationship that allows $p^{\leftarrow}$ to flow to $q^{\leftarrow}$. This is secure since $pc = q^{\leftarrow}$, meaning that only principals with integrity $q^{\leftarrow}$ have influenced the computation. With $\langle p^{\leftarrow} \succcurlyeq q^{\leftarrow} \mid q^{\leftarrow} \rangle$ in the delegation context, added via the ASSUME rule, the premises of BINDM are now satisfied, so the expression type-checks.

Creating a delegation value requires no special privilege because the type system ensures only high-integrity delegations are used as evidence that enable new flows. Using low-integrity evidence for authorization would be insecure since attackers could use delegation values to create new flows that reveal secrets or corrupt data. The premises of the ASSUME rule ensure the integrity of dynamic authorization computations that produce values like $\langle p^{\leftarrow} \succcurlyeq q^{\leftarrow} \rangle$ in the example above.[4] The second premise, $\Pi; pc; pc \Vdash pc \succcurlyeq \nabla(q)$, requires that the *pc* has enough integrity to be trusted by q, the principal whose security is affected. For instance, to make the assumption $p \succcurlyeq q$,

[3] This correspondence with delegation values makes acts-for types a kind of singleton type [29].

[4] These premises are related to the robust FLAM rule LIFT.

$\boxed{e \longrightarrow e' \texttt{ where } \langle p \succcurlyeq q \rangle}$

[W-APP] $(v \texttt{ where } \langle p \succcurlyeq q \rangle)\ v' \longrightarrow (v\ v') \texttt{ where } \langle p \succcurlyeq q \rangle$

[W-TAPP] $(v \texttt{ where } \langle p \succcurlyeq q \rangle)\ s \longrightarrow (v\ s) \texttt{ where } \langle p \succcurlyeq q \rangle$

[W-UNPAIR] $\texttt{proj}_i\ (\langle v_1, v_2 \rangle \texttt{ where } \langle p \succcurlyeq q \rangle) \longrightarrow (\texttt{proj}_i\ \langle v_1, v_2 \rangle) \texttt{ where } v$

[W-CASE] $(\texttt{case } (v \texttt{ where } \langle p \succcurlyeq q \rangle) \texttt{ of } \texttt{inj}_1(x).\ e_1 \mid \texttt{inj}_2(x).\ e_2) \longrightarrow (\texttt{case } v \texttt{ of } \texttt{inj}_1(x).\ e_1 \mid \texttt{inj}_2(x).\ e_2) \texttt{ where } \langle p \succcurlyeq q \rangle$

[W-UNITM] $(\eta_\ell\ v \texttt{ where } \langle p \succcurlyeq q \rangle) \longrightarrow (\eta_\ell\ v) \texttt{ where } \langle p \succcurlyeq q \rangle$

[W-BINDM] $\texttt{bind } x = (v \texttt{ where } \langle p \succcurlyeq q \rangle) \texttt{ in } e \longrightarrow (\texttt{bind } x = v \texttt{ in } e) \texttt{ where } \langle p \succcurlyeq q \rangle$

[W-ASSUME] $\texttt{assume } (v \texttt{ where } \langle p \succcurlyeq q \rangle) \texttt{ in } e \longrightarrow (\texttt{assume } v \texttt{ in } e) \texttt{ where } \langle p \succcurlyeq q \rangle$

Figure 4.7: FLAC evaluation rules for `where` terms

the evidence represented by the term e must have at least the integrity of the voice of q, written $\nabla(q)$. Since the *pc* bounds the restrictiveness of the dependencies of e, this ensures that only information with integrity $\nabla(q)$ or higher may influence the evaluation of e. The third premise, $\Pi; \mathit{pc}; \mathit{pc} \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow})$, ensures that principal p has sufficient integrity to be trusted to enforce q's confidentiality, $q^{\rightarrow}$. This premise means that q permits data to be relabeled from $q^{\rightarrow}$ to $p^{\rightarrow}$.[5]

Assumption terms evaluate to `where` expressions (rule E-ASSUME). To simplify the formalization, these expressions are not part of the source language but are generated by the evaluation rules. The term e `where` v records that e is evaluated in a context that includes the delegation v. The rule WHERE gives a typing rule for `where` terms; though similar to ASSUME, it requires only that there exist a sufficiently trusted label *pc′* such that subexpression e type-checks. In the proofs in Section 4.6, we choose *pc′* using the typing judgment of the source-level `assume` that generates the `where` term.

Figure 4.7 presents evaluation rules for where terms. These terms are simply a book-

[5]More precisely, it means that the voice of q's confidentiality, $\nabla(q^{\rightarrow})$, permits data to be relabeled from $q^{\rightarrow}$ to $p^{\rightarrow}$. Recall that $\nabla(\texttt{Alice}^{\rightarrow})$ is just `Alice`'s integrity projection: $\texttt{Alice}^{\leftarrow}$.

$\boxed{\Pi; pc \vdash \ell \leq s}$

[P-UNIT] $\Pi; pc \vdash \ell \leq \texttt{unit}$

[P-PAIR] $$\frac{\Pi; pc \vdash \ell \leq s_1 \qquad \Pi; pc \vdash \ell \leq s_2}{\Pi; pc \vdash \ell \leq (s_1 \times s_2)}$$

[P-FUN] $$\frac{\Pi; pc \vdash \ell \leq s_2}{\Pi; pc \vdash \ell \leq s_1 \xrightarrow{pc'} s_2}$$

[P-TFUN] $$\frac{\Pi; pc \vdash \ell \leq s}{\Pi; pc \vdash \ell \leq \forall X.\, s}$$

[P-LBL1] $$\frac{\Pi; pc \vdash \ell \leq s}{\Pi; pc \vdash \ell \leq \ell' \texttt{ says } s}$$

[P-LBL2] $$\frac{\Pi; pc; pc \Vdash \ell \sqsubseteq \ell'}{\Pi; pc \vdash \ell \leq \ell' \texttt{ says } s}$$

Figure 4.8: Type protection levels

keeping mechanism: these evaluation rules simply record and maintain the authorization evidence used to justify new flows of information that occur during the evaluation of a FLAC program. The rules are designed to treat `where` values like the value they enclose. For instance, applying a `where` term (rule W-APP) simply moves the value it is applied to inside the where term. If the `where` term was wrapping a lambda expression, then it may now be applied via APP. Otherwise, further reduction steps via W-APP may be necessary.

4.4 Examples revisited

We can now implement our examples from Section 4.1 in FLAC. Using FLAC ensures that authority and information flow assumptions are explicit, and that programs using these abstractions are secure with respect to those assumptions. In this section, we discuss at a high level how FLAC types help enforce specific end-to-end security properties for commitment schemes and bearer credentials. Section 4.6 formalizes the semantic security properties of all well-typed FLAC programs.

$$\texttt{commit}\colon \forall X.\, p^{\rightarrow}\ \texttt{says}\ X \xrightarrow{p^{\leftarrow}} p\ \texttt{says}\ X$$
$$\begin{array}{l}\texttt{commit} = \\ \quad \Lambda X.\, \lambda(x\colon p^{\rightarrow}\ \texttt{says}\ X)[p^{\leftarrow}]. \\ \quad\quad \texttt{assume}\ \langle \bot^{\leftarrow} \succcurlyeq p^{\leftarrow} \rangle\ \texttt{in}\ \texttt{bind}\ x' = x\ \texttt{in}\ (\eta_p\ x')\end{array}$$

$$\texttt{receive}\colon \forall X.\, p\ \texttt{says}\ X \xrightarrow{q^{\leftarrow}} p \wedge q^{\leftarrow}\ \texttt{says}\ X$$
$$\begin{array}{l}\texttt{receive} = \\ \quad \Lambda X.\, \lambda(x\colon p\ \texttt{says}\ X)[q^{\leftarrow}]. \\ \quad\quad \texttt{assume}\ \langle p^{\leftarrow} \succcurlyeq q^{\leftarrow} \rangle\ \texttt{in}\ \texttt{bind}\ x' = x\ \texttt{in}\ (\eta_{p \wedge q^{\leftarrow}}\ x')\end{array}$$

$$\texttt{open}\colon \forall X.\, p \wedge q^{\leftarrow}\ \texttt{says}\ X \xrightarrow{\nabla(p^{\rightarrow})} p^{\leftarrow} \wedge q\ \texttt{says}\ X$$
$$\begin{array}{l}\texttt{open} = \\ \quad \Lambda X.\, \lambda(x\colon p \wedge q^{\leftarrow}\ \texttt{says}\ X)[\nabla(p^{\rightarrow})]. \\ \quad\quad \texttt{assume}\ \langle \nabla(q^{\rightarrow}) \succcurlyeq \nabla(p^{\rightarrow}) \rangle\ \texttt{in} \\ \quad\quad\quad \texttt{assume}\ \langle q^{\rightarrow} \succcurlyeq p^{\rightarrow} \rangle\ \texttt{in}\ \texttt{bind}\ x' = x\ \texttt{in}\ (\eta_{p^{\leftarrow} \wedge q}\ x')\end{array}$$

Figure 4.9: FLAC implementations of commitment scheme operations.

4.4.1 Commitment schemes

Figure 4.9 contains the essential operations of a one-round commitment scheme—`commit`, `receive`, and `open`—implemented in FLAC. Typically, a principal p commits to a value and sends it to q, who receives it. Later, p opens the value, revealing it to q. The `commit` operation takes a value of any type (hence $\forall X$) with confidentiality $p^{\rightarrow}$ and produces a value with confidentiality and integrity p. In other words, p *endorses* [98] the value to have integrity $p^{\leftarrow}$.

Attackers should not be able to influence whether principal p commits to a particular value. The *pc* constraint on `commit` ensures that only principal p and principals trusted with at least p's integrity, $p^{\leftarrow}$, may apply `commit` to a value.[6] Furthermore, if the programmer omitted this constraint or instead chose $\bot^{\leftarrow}$, say, then `commit` would

[6] We make the reasonable assumption that an untrusted programmer cannot modify high-integrity code, thus the influence of attackers is captured by the pc and the protection levels of values. Enforcing this assumption is beyond the scope of FLAC, but has been explored in [6].

be rejected by the type system. Specifically, the `assume` term would not type-check via rule ASSUME since the *pc* does not act for $\nabla(p^{\leftarrow}) = p^{\leftarrow}$.

Next, principal q accepts a committed value from p using the `receive` operation. The `receive` operation endorses the value with q's integrity, resulting in a value at $p \wedge q^{\leftarrow}$, the confidentiality of p and the integrity of both p and q.

As with the `commit` operation, FLAC ensures that `receive` satisfies important information security properties. Other principals, including p, should not be able to influence which values q receives—otherwise an attacker could use `receive` to subvert q's integrity, using it to endorse arbitrary values. The *pc* constraint on `receive` ensures in this case that only q may apply `receive`. Furthermore, the type of x requires received values to have the integrity of p. Errors in either of these constraints would result in a typing error, either due to ASSUME as before, or due to BINDM, which requires that p must flow to $p \wedge q^{\leftarrow}$.

Additionally, `receive` accepts committed values with confidentiality at most $p^{\rightarrow}$. This constraint ensures that q does not `receive` values from p that might depend on q's secrets: unopened commitments, for example. In cryptographic protocols, this property is usually called *non-malleability* [27], and is important for scenarios in which security depends on the independence of values. Consider a sealed-bid auction where participants submit their bids via commitment protocols. Suppose that q `commits` a bid b, protected by label q. Then p could theoretically influence a computation that computes a value $b + 1$ with label $p \wedge q^{\rightarrow}$ since that label protects information at $q^{\rightarrow}$, but only has $p^{\leftarrow}$ integrity. If q received values from p that could depend on q's secrets, then p could outbid q by 1 without ever learning the value b.

Finally, `open` reveals a committed value to q by relabeling a value from $p \wedge q^{\leftarrow}$ to $p^{\leftarrow} \wedge q$, which is readable by principal q but retains the integrity of both p and q. Since `open` accepts a value protected by the integrity of both p and q and returns a value

with the same integrity, the opened value must have been previously committed by p and received by q. Since the `open` operation reveals a value with confidentiality $p^{\rightarrow}$, it should only be invoked by principals that are trusted to speak for $p^{\rightarrow}$. Otherwise, q could open p's commitments. Hence, the pc label of `open` is $\nabla(p^{\rightarrow})$. For $p = \texttt{Alice}$, say, the pc label would be $\texttt{Alice}^{\leftarrow}$. FLAC ensures these constraints are specified correctly; otherwise, `open`'s implementation could not produce a value with label $p^{\leftarrow} \wedge q$.

The implementation requires two `assume` terms. The outer term establishes that principals speaking for $q^{\rightarrow}$ also speak for $p^{\rightarrow}$ by creating an integrity relationship between their voices. With this relationship in place, the inner term may reveal the commitment to q.[7]

In DCC, functions are not annotated with *pc* labels and may be applied in any context. So a DCC function analogous to `open` might have type

$$\texttt{dcc\_open} : \forall X.\, p \wedge q^{\leftarrow}\ \texttt{says}\ X \rightarrow p^{\leftarrow} \wedge q\ \texttt{says}\ X$$

However, `dcc_open` would not be appropriate for a commitment scheme since any principal could use it to relabel information from p-confidential ($p^{\rightarrow}$) to q-confidential ($q^{\rightarrow}$).

To simplify the presentation of our commitment scheme operations, we make the assumption that q only receives one value. Therefore, p can only open one value, since only one value has been given the integrity of both p and q. A more general scheme can be achieved by pairing each committed value with a public identifier that is endorsed along with the value, but remains public. If q refuses to receive more that one commitment with the same identifier[8], p will be unable to open two commitments with the same value since it cannot create a pair that has the integrity of both p and q, even if p has multiple committed values (with different identifiers) to choose from. We present the simpler one-round commitment scheme above since it captures the essential information

[7] specifically, it satisfies the ASSUME premise $\Pi; pc; pc \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow})$.

[8] For cryptographic commitment schemes, the commitment ciphertext itself could act as a public identifier, and q could rely on cryptographic assumptions that distinct values cannot (with high probability) have the same identifier instead of explicitly checking whether the identifier has been used before.

security properties of commitment while avoiding the tedious digression of defining encodings for numeric values and numeric comparisons.

The real power of FLAC is that the security guarantees of well-typed FLAC functions like those above are compositional. The FLAC type system ensures the security of both the functions themselves and the programs that use them. For instance, the code should be rejected because it would permit q to open p's commitments:

$$\Lambda X.\, \lambda(x{:}p \wedge q^{\leftarrow}\ \texttt{says}\ X)[q^{\leftarrow}].\, \texttt{assume}\ \langle q \succcurlyeq p\rangle\ \texttt{in}\ \texttt{open}\ x$$

FLAC's guarantees make it possible to state general security properties of all programs that use the above commitment scheme, even if those programs are malicious. For example, suppose we have $\mathit{pc}_p = \nabla(p)$, $\mathit{pc}_q = \nabla(q)$, and

$$\Gamma_{cro} = \texttt{commit}, \texttt{receive}, \texttt{open}, x{:}p^{\rightarrow}\ \texttt{says}\ s, y{:}p \wedge q^{\leftarrow}\ \texttt{says}\ s$$

Intuitively, pc_p and pc_q are execution contexts under the control of p or q, respectively. Γ_{cro} is a typing context for programs using the commitment scheme.[9] The variable x represents an uncommitted value with p's confidentiality, whereas y is a committed value. Since we are interested in properties that hold for all principals p and q, we want the properties to hold in an empty delegation context: $\Pi = \varnothing$. Below, we omit the delegation context altogether for brevity.

Using results presented in Section 4.6, we can prove that:

- q **cannot `receive` a value that hasn't been committed.** For any e and s' such that $\Gamma_{cro}; \mathit{pc}_q \vdash e : p \wedge q^{\leftarrow}\ \texttt{says}\ s'$, result of e is independent of x; specifically, for any v_1 and v_2, if $e[x \mapsto v_1] \longrightarrow^* v_1'$ and $e[x \mapsto v_2] \longrightarrow^* v_2'$, then $v_1' = v_2'$.
- q **cannot learn a value that hasn't been opened.** For any e, ℓ, and s' such that $\Gamma_{cro}; \mathit{pc}_q \vdash e : \ell \sqcap q^{\rightarrow}\ \texttt{says}\ s'$, then the result of e is independent of x and y.

[9] For presentation purposes, we have omitted the types of `commit`, `receive`, and `open` in Γ_{cro}. Their types are as defined previously.

- p **cannot open a value that hasn't been received.** For any e such that $\Gamma_{cro}; \mathit{pc}_p \vdash e : p^{\leftarrow} \wedge q \texttt{ says } s'$, then the result of e is independent of x.

For the first two properties, we consider programs using our commitment scheme that q might invoke, hence we consider FLAC programs that type-check in the $\Gamma_{cro}; \mathit{pc}_q$ context. In the first property, we are concerned with programs that produce values protected by policy $p \wedge q^{\leftarrow}$. Since such programs produce values with the integrity of p but are invoked by q, we want to ensure that no program exists that enables q to obtain a value with p's integrity that depends on x, which is a value without p's integrity. The second property concerns programs that produces values at $\ell \sqcap q^{\rightarrow}$ for any ℓ; these are values readable by q. Therefore, we want to ensure that no program exists that enables q to produce such a value that depends on x or y, which are not readable by q.

The final property considers programs that p might invoke to produce values at $p^{\leftarrow} \wedge q$, thus we consider FLAC programs that type-check in the $\Gamma_{cro}; \mathit{pc}_p$ context. Here, we want to ensure that no program invoked by p can produce a value at $p^{\leftarrow} \wedge q$ that depends on x, an unreceived value. Complete proofs of these properties are found in Appendix B.2.

4.4.2 Bearer credentials

We can also use FLAC to implement bearer credentials, our second example of a dynamic authorization mechanism. We represent a bearer credential with authority k in FLAC as a term with the type

$$\forall X.\, k^{\rightarrow} \texttt{ says } X \xrightarrow{\mathit{pc}} k^{\leftarrow} \texttt{ says } X$$

which we abbreviate as $k^{\rightarrow} \overset{\mathit{pc}}{\Rightarrow} k^{\leftarrow}$. These terms act as bearer credentials for a principal k since they may be used as a proxy for k's confidentiality and integrity authority. Recall that $k^{\leftarrow} = k^{\leftarrow} \wedge \bot^{\rightarrow}$ and $k^{\rightarrow} = k^{\rightarrow} \wedge \bot^{\leftarrow}$. Then secrets protected by $k^{\rightarrow}$ can be

declassified to $\bot^{\rightarrow}$, and untrusted data protected by $\bot^{\leftarrow}$ can be endorsed to $k^{\leftarrow}$. Thus this term wields the full authority of k, and if $pc = \bot^{\leftarrow}$, the credential may be used in any context—any "bearer" may use it. From such credentials, more restricted credentials can be derived. For example, the credential $k^{\rightarrow} \overset{pc}{\Rightarrow} \bot^{\rightarrow}$ grants the bearer authority to declassify k-confidential values, but no authority to endorse values.

We postpone an in-depth discussion of terms with types of the form $k^{\rightarrow} \overset{pc}{\Rightarrow} k^{\leftarrow}$ until Section 4.5.2, but it is interesting to note that an analogous term in DCC is only well-typed if k is equivalent to $\bot$. This is because the function takes an argument with $k^{\rightarrow}$ confidentiality and no integrity, and produces a value with $k^{\leftarrow}$ integrity and no confidentiality. Suppose $\mathcal{L}$ is a security lattice used to type-check DCC programs with suitable encodings for k's confidentiality and integrity. If a DCC term has a type analogous to $k^{\rightarrow} \Rightarrow k^{\leftarrow}$, then $\mathcal{L}$ must have the property $k^{\rightarrow} \sqsubseteq \bot$ and $\bot \sqsubseteq k^{\leftarrow}$. This means that k has no confidentiality and no integrity. That FLAC terms may have this type for any principal k makes it straightforward to implement bearer credentials and demonstrates a useful application of FLAC's extra expressiveness.

The *pc* of a credential $k^{\rightarrow} \overset{pc}{\Rightarrow} k^{\leftarrow}$ acts as a sort of caveat: it restricts the information flow context in which the credential may be used. We can add more general caveats to credentials by wrapping them in lambda terms. To add a caveat ϕ to a credential with type $k^{\rightarrow} \overset{pc}{\Rightarrow} k^{\leftarrow}$, we use a wrapper:

$$\lambda(x : k^{\rightarrow} \overset{pc}{\Rightarrow} k^{\leftarrow})[pc].\, \Lambda X.\, \lambda(y : \phi)[pc].\, xX$$

which gives us a term with type

$$\forall X.\, \phi \xrightarrow{pc} k^{\rightarrow}\ \texttt{says}\ X \xrightarrow{pc} k^{\leftarrow}\ \texttt{says}\ X$$

This requires a term with type ϕ (in which X may occur) to be applied before the authority of k can be used. Similar wrappers allow us to chain multiple caveats; specifically,

for caveats $\phi_1 \ldots \phi_n$, we obtain the type

$$\forall X.\, \phi_1 \xrightarrow{pc} \ldots \xrightarrow{pc} \phi_n \xrightarrow{pc} k^{\rightarrow}\ \texttt{says}\ X \xrightarrow{pc} k^{\leftarrow}\ \texttt{says}\ X$$

which abbreviates to

$$k^{\rightarrow} \xRightarrow{\phi_1 \times \cdots \times \phi_n ; pc} k^{\leftarrow}$$

Like any other FLAC terms, credentials may be protected by information flow policies. So a credential that should only be accessible to Alice might be protected by the type $\texttt{Alice}^{\rightarrow}\ \texttt{says}\ (k^{\rightarrow} \xRightarrow{\phi ; pc} k^{\leftarrow})$. This confidentiality policy ensures the credential cannot accidentally be leaked to an attacker. A further step might be to constrain uses of this credential so that only Alice may invoke it to relabel information. If we require $pc = \texttt{Alice}^{\leftarrow}$, this credential may only be used in contexts trusted by Alice: $\texttt{Alice}^{\rightarrow}\ \texttt{says}\ (k^{\rightarrow} \xRightarrow{\phi ; \texttt{Alice}^{\leftarrow}} k^{\leftarrow})$.

A subtle point about the way in which we construct caveats is that the caveats are polymorphic with respect to X, the same type variable the credential ranges over. This means that each caveat may constrain what types X may be instantiated with. For instance, suppose `isEduc` is a predicate for educational films; it holds (has a proof term with type $\texttt{isEduc}\ X$) for types like `Bio` and `Doc`, but not `RomCom`. Adding $\texttt{isEduc}\ X$ as a caveat to a credential would mean that the bearer of the credential could use it to access biographies and documentaries, but could not use it to access romantic comedies. Since no term of type $\texttt{isEduc}\ \texttt{RomCom}$ could be applied, the bearer could only satisfy `isEduc` by instantiating X with `Bio` or `Doc`. Once X is instantiated with `Bio` or `Doc`, the credential cannot be used on a `RomCom` value. Thus we have two mechanisms for constraining the use of credentials: information flow policies to constrain propagation, and caveats to establish prerequisites and constrain the types of data covered by the credential.

As a more in-depth example of using such credentials, suppose Alice hosts a file sharing service. For a simpler presentation, we use free variables to refer to these files;

for instance, $x_1 : (k_1\ \texttt{says}\ \texttt{ph})$ is a variable that stores a photo (type $\texttt{ph}$) protected by k_1. For each such variable x_1, Alice has a credential $k_1^{\rightarrow} \stackrel{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow}$, and can give access to users by providing this credential or deriving a more restricted one. To access x_1, Bob does not need the full authority of Alice or k_1—a more restricted credential suffices:

$$\lambda(c{:}k_1 \stackrel{\texttt{Bob}^{\leftarrow}}{\Longrightarrow} \texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow}\ \texttt{ph})[\texttt{Bob}^{\leftarrow}].$$
$$\texttt{bind}\ x_1' = c\ x_1\ \texttt{in}\ (\eta_{\texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow}}\ x_1')$$

Here, c is a credential $k_1 \stackrel{\texttt{Bob}^{\leftarrow}}{\Longrightarrow} \texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow}$ whose polymorphic type has been instantiated with the photo type $\texttt{ph}$. This credential accepts a photo protected at k_1 and returns a photo protected at $\texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow}$, which Bob is permitted to access.

The advantage of bearer credentials is that access to x_1 can be provided to principals other than k_1 in a decentralized way, without changing the policy on x_1. For instance, suppose Alice wants to issue a credential to Bob to access resources protected by k_1. Alice has a credential with type $k_1^{\rightarrow} \stackrel{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow}$, but she wants to ensure that only Bob (or principals Bob trusts) can use it. In other words, she wants to create a credential of type $k_1 \stackrel{\texttt{Bob}^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow}$, which needs Bob's integrity to use.

Alice can create such a credential using a wrapper that derives a more constrained credential from her original one.

$$\lambda(c{:}k_1^{\rightarrow} \stackrel{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow})[\texttt{Alice}^{\leftarrow}].$$
$$\Lambda X.\, \lambda(y{:}k_1\ \texttt{says}\ X)[\texttt{Bob}^{\leftarrow}].$$
$$\texttt{bind}\ y' = y\ \texttt{in}\ (c\ X)\ (\eta_{k^{\rightarrow}}\ y')$$

Then Bob can use this credential to access x_1 by deriving a credential of type $k_1 \stackrel{\texttt{Bob}^{\leftarrow}}{\Longrightarrow} \texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow}\ \texttt{ph}$ using the function

$$\lambda(c{:}k_1 \stackrel{\texttt{Bob}^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow})[\texttt{Bob}^{\leftarrow}].$$
$$\lambda(y{:}k_1\ \texttt{says}\ \texttt{ph})[\texttt{Bob}^{\leftarrow}].$$
$$\texttt{bind}\ y' = c\,\texttt{ph}\ y\ \texttt{in}\ (\eta_{\texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow}}\ y')$$

which can be applied to obtain a value readable by Bob.

Bob can also use this credential to share photos with friends. For instance, the function

$$
\begin{array}{l}
\lambda(c{:}k_1 \overset{\mathtt{Bob}^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow})[\mathtt{Bob}^{\leftarrow}]. \\
\quad \mathtt{assume}\ \langle \mathtt{Carol}^{\leftarrow} \succcurlyeq \mathtt{Bob}^{\leftarrow} \rangle\ \mathtt{in} \\
\quad\quad \lambda(\_{:}\mathtt{unit})[\mathtt{Carol}^{\leftarrow}]. \\
\quad\quad\quad \mathtt{bind}\ x_1' = c\,\mathtt{ph}\ x_1\ \mathtt{in}\ (\eta_{\mathtt{Carol}^{\rightarrow} \wedge k_1^{\leftarrow}}\ x_1')
\end{array}
$$

creates a wrapper around a specific photo x_1. Only principals trusted by Carol may invoke the wrapper, which produces a value of type $\mathtt{Carol}^{\rightarrow} \wedge k_1^{\leftarrow}\ \mathtt{says}\ \mathtt{ph}$, permitting Carol to access the photo.

The properties of FLAC let us prove many general properties about such bearer-credential programs; here, we examine three properties. For $i \in \{1..n\}$, let

$$\Gamma_{bc} = x_i{:}k_i\ \mathtt{says}\ s_i, c_i{:}\mathtt{Alice}\ \mathtt{says}\ (k_i^{\leftarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_i^{\leftarrow})$$

where k_i is a primitive principal protecting the i^{th} resource of type s_i, and c_i is a credential for the i^{th} resource and protected by Alice. Assume $k_i \notin \{\mathtt{Alice}, \mathtt{Friends}, p\}$ for all i where p represents a (potentially malicious) user of Alice's service, and $\mathtt{Friends}$ is a principal for Alice's friends, (for example, $\mathtt{Friends} = (\mathtt{Bob} \vee \mathtt{Carol})$). Also, define $pc_p = p^{\leftarrow}$ and $pc_A = \mathtt{Alice}^{\leftarrow}$.

- p **cannot access resources without a credential.** For any e, ℓ, and s' such that $\Gamma_{bc}; pc_p \vdash e : \ell \sqcap p^{\rightarrow}\ \mathtt{says}\ s'$, the value of e is independent of x_i for all i.
- p **cannot use unrelated credentials to access resources.** For any e, ℓ, and s' such that

  $$\Gamma_{bc}, c_p{:}(k_1^{\leftarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow}); pc_p \vdash e : \ell \sqcap p^{\rightarrow}\ \mathtt{says}\ s'$$

  the value e computes is independent of x_i for $i \neq 1$.

- **Alice cannot disclose secrets by issuing credentials.** For all i and $j \neq 1$, define

$$\Gamma'_{bc} = x_i : k_i \texttt{ says } s_i, c_i : \texttt{Alice says } (k_j^{\leftarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_j^{\leftarrow}),$$
$$c_F : \texttt{Friends says } (k_1^{\leftarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow})$$

  Then if $\Gamma'_{bc}; pc_A \vdash e : \ell \sqcap p^{\rightarrow} \texttt{ says } (k_j^{\leftarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_j^{\leftarrow})$ for some e, ℓ, and s', the value of e is independent of x_1.

These properties demonstrate the power of FLAC's type system. The first two ensure that credentials really are necessary for p to access protected resources, even indirectly. In the first, p has no credentials, and the type system ensures that p cannot invoke a program that produces a value p can read (represented by $\ell \sqcap p^{\rightarrow}$) that depends on any variable x_i. In the second, a credential c_p with type $k_1^{\leftarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow}$ is accessible to p, but p cannot use it to access other variables. The third property eliminates covert channels like the one discussed in Section 4.1.2. It implies that credentials issued by Alice do not leak information, in this case about Alice's friends. By implementing bearer credentials in FLAC, we can demonstrate these three properties with relatively little effort.

4.5 FLAC proof theory

4.5.1 Properties of says

FLAC's type system constrains how principals apply existing facts to derive new facts. For instance, a property of says in other authorization logics (for example, Lampson et al. [49] and Abadi [2]) is that implications that hold for top-level propositions also hold for propositions of any principal ℓ:

$$\vdash (s_1 \rightarrow s_2) \rightarrow (\ell \texttt{ says } s_1 \rightarrow \ell \texttt{ says } s_2)$$

The *pc* annotations on FLAC function types refine this property. Each implication (in other words, each function) in FLAC is annotated with an upper bound on the informa-

tion flow context it may be invoked within. To lift such an implication to operate on propositions protected at label ℓ, the label ℓ must flow to the *pc* of the implication. Thus, for all ℓ and s_i,

$$\vdash (s_1 \xrightarrow{pc \sqcup \ell} s_2) \xrightarrow{pc} (\ell \texttt{ says } s_1 \xrightarrow{pc} \ell \texttt{ says } s_2)$$

This judgment is a FLAC typing judgment in *logical form*, where terms have been omitted. We write such judgments with an empty typing context (as above) when the judgment is valid for any Π, Γ, and *pc*. A judgment in logical form is valid if a *proof term* exists for the specified type, proving the type is inhabited. The above type has proof term

$$\lambda(f : (s_1 \xrightarrow{pc \sqcup \ell} s_2))[pc].$$
$$\lambda(x : \ell \texttt{ says } s_1)[pc].\, \texttt{bind } x' = x \texttt{ in } (\eta_\ell\ f\ x')$$

In order to apply f, we must first `bind` x, so according to rules BINDM and APP, the function f must have a label at least as restrictive as $pc \sqcup \ell$. All theorems of DCC can be obtained by encoding them as FLAC implications with $pc = \top^{\rightarrow}$, the highest bound. Since any principal ℓ flows to $\top^{\rightarrow}$, such implications may be applied in any context.

These refinements of DCC's theorems are crucial for supporting applications like commitment schemes and bearer credentials. Recall from Sections 4.4.1 and 4.4.2 that the security of these mechanisms relied in part on restricting the *pc* to a specific principal's integrity. Without such refinements, principal q could open principal p's commitments using `open`, or create credentials with p authority: $p^{\rightarrow} \overset{pc}{\Rightarrow} p^{\leftarrow}$.

Other properties of `says` common to DCC and other logics (cf. [1] for examples) are similarly refined by *pc* bounds. Two examples are: $\vdash s \xrightarrow{pc} \ell \texttt{ says } s$ which has proof term: $\lambda(x : s)[pc].\, (\eta_\ell\ s)$ and

$$\vdash \ell \texttt{ says } (s_1 \xrightarrow{pc \sqcup \ell} s_2) \xrightarrow{pc} (\ell \texttt{ says } s_1 \xrightarrow{pc} \ell \texttt{ says } s_2)$$

with proof term:

$$\lambda(f : \ell \ \texttt{says}\ (s_1 \xrightarrow{pc \sqcup \ell} s_2))[pc].\ \texttt{bind}\ x' = x\ \texttt{in}$$
$$\lambda(y : \ell \ \texttt{says}\ s_1)[pc].\ \texttt{bind}\ y' = y\ \texttt{in}\ (\eta_\ell\ x'\ y')$$

As in DCC, chains of says are commutative in FLAC:

$$\vdash \ell_1 \ \texttt{says}\ \ell_2 \ \texttt{says}\ s \xrightarrow{pc} \ell_2 \ \texttt{says}\ \ell_1 \ \texttt{says}\ s$$

with proof term

$$\lambda(x : \ell_1 \ \texttt{says}\ \ell_2 \ \texttt{says}\ s)[pc].$$
$$\texttt{bind}\ y = x\ \texttt{in}\ \texttt{bind}\ z = y\ \texttt{in}\ (\eta_{\ell_2}\ (\eta_{\ell_1}\ z))$$

In some logics with different interpretations of says (for example, CCD [5]) differently ordered chains are distinct, but here we find commutativity appealing since it matches the intuition from information flow control. When principal ℓ_1 says that ℓ_2 says s, we should protect s with a policy at least as restrictive as both ℓ_1 and ℓ_2, specifically, the principal $\ell_1 \sqcup \ell_2$. Since $\sqcup$ is commutative, who said what first is irrelevant.

4.5.2 Dynamic hand-off

Many authorization logics support delegation using a "hand-off" axiom. In DCC, this axiom is actually a provable theorem:

$$\vdash (q \ \texttt{says}\ (p \Rightarrow q)) \rightarrow (p \Rightarrow q)$$

where $p \Rightarrow q$ is shorthand for

$$\forall X.\ (p \ \texttt{says}\ X \rightarrow q \ \texttt{says}\ X)$$

However, $p \Rightarrow q$ is only inhabited if $p \sqsubseteq q$ in the security lattice. Thus, DCC can reason about the consequences of $p \sqsubseteq q$ (whether it is true for the lattice or not), but a DCC program cannot produce a term of type $p \Rightarrow q$ unless $p \sqsubseteq q$.

FLAC programs, on the other hand, can create new trust relationships from delegation expressions using `assume` terms. The type analogous to $p \Rightarrow q$ in FLAC is

$$\forall X.\, (p \texttt{ says } X \xrightarrow{pc} q \texttt{ says } X)$$

which we wrote as $p \overset{pc}{\Rightarrow} q$ in Section 4.4.2. FLAC programs construct terms of this type from proofs of authority, represented by terms with acts-for types. This feature enables a more general form of hand-off, which we state formally below.

Proposition 1 (Dynamic hand-off)**.** *For all ℓ and pc', let $pc = \ell^{\rightarrow} \wedge \nabla(p^{\rightarrow}) \wedge q^{\leftarrow}$*

$$(\nabla(q^{\rightarrow}) \succcurlyeq \nabla(p^{\rightarrow})) \xrightarrow{pc} (p \sqsubseteq q) \xrightarrow{pc}$$
$$\forall X.\, (p \texttt{ says } X \xrightarrow{pc'} q \texttt{ says } X)$$

Proof term.

$$\begin{aligned}
&\lambda(pf_1 : (\nabla(q^{\rightarrow}) \succcurlyeq \nabla(p^{\rightarrow})))[pc].\\
&\quad \lambda(pf_2 : (p \sqsubseteq q))[pc].\\
&\qquad \texttt{assume } pf_1 \texttt{ in assume } pf_2 \texttt{ in}\\
&\qquad\quad \Lambda X.\, \lambda(x : p \texttt{ says } X)[pc'].\, \texttt{bind } x' = x \texttt{ in } (\eta_q\, x')
\end{aligned}$$

The principal $pc = \ell^{\rightarrow} \wedge \nabla(p^{\rightarrow}) \wedge q^{\leftarrow}$ restricts delegation (hand-off) to contexts with the integrity of $\nabla(p^{\rightarrow}) \wedge q^{\leftarrow}$. The two arguments are proofs of authority with acts-for types: a proof of $\nabla(q^{\rightarrow}) \succcurlyeq \nabla(p^{\rightarrow})$ and a proof of $p \sqsubseteq q$. The *pc* ensures that the proofs have sufficient integrity to be used in `assume` terms since it has the integrity of both $\nabla(p^{\rightarrow})$ and $q^{\leftarrow}$. Note that low-integrity or confidential delegation values must first be bound via `bind` before the above term may be applied. Thus the *pc* would reflect the protection level of both arguments. Principals $\ell^{\rightarrow}$ and pc' are unconstrained. As demonstrated in Section 4.4.2, FLAC programmers may instantiate these principals is a variety of ways to enforce different properties.

Dynamic hand-off terms give FLAC programs a level of expressiveness and security not offered by other authorization logics. Observe that pc' may be chosen independently of the other principals. This means that although the pc prevents low-integrity principals from creating hand-off terms, a high-integrity principal may create a hand-off term and provide it to an arbitrary principal. Hand-off terms in FLAC, then, are similar to capabilities since even untrusted principals may use them to change the protection level of values. Unlike in most capability systems, however, the propagation of hand-off terms can be constrained using information flow policies.

Terms that have types of the form in Proposition 1 illustrate a subtlety of enforcing information flow in an authorization mechanism. Because these terms relabel information from one protection level to another protection level, the transformed information implicitly depends on the proofs of authorization. FLAC ensures that the information security of these proofs is protected—like that of all other values—even as the policies of other information are being modified. Hence, authorization proofs cannot be used as a side channel to leak information.

4.6 Semantic security properties of FLAC

4.6.1 Delegation invariance

FLAC programs dynamically extend trust relationships, enabling new flows of information. Nevertheless, well-typed programs have end-to-end semantic properties that enforce strong information security. These properties derive primarily from FLAC's control of the delegation context. The ASSUME rule ensures that only high-integrity proofs of authorization can extend the delegation context, and furthermore that such extensions occur only in high-integrity contexts.

That low-integrity contexts cannot extend the delegation context turns out to be a

crucial property. This property allows us to state a useful invariant about the evaluation of FLAC programs. Recall that `assume` terms evaluate to `where` terms in the FLAC semantics. Thus, FLAC programs typically compute values containing a hierarchy of nested `where` terms. The terms record the values whose types were used to extend the delegation context during type checking.

For a well-typed FLAC program, we can prove that certain trust relationships could not have been added by the program. Characterizing these relationships requires a concept of the minimal authority required to cause one principal to act for another. Although similar, this idea is distinct from the voice of a principal. Consider the relationship between a and $a \wedge b$. The voice of $a \wedge b$, $\nabla(a \wedge b)$, is sufficient integrity to add a delegation $a \wedge b$ to a so that $a \succcurlyeq a \wedge b$. Alternatively, having only the integrity of $\nabla(b)$ is sufficient to add a delegation $a \succcurlyeq b$, which also results in $a \succcurlyeq a \wedge b$. To precisely characterize which trust relationships can not be added by program, we need to identify this minimal integrity $\nabla(b)$ given the pair of principals a and $a \wedge b$. The following definitions are in service of this goal.

The first definition formalizes the idea that two principals are considered equivalent in a given context if they act for each other.

Definition 8 (Principal Equivalence)**.** We say that two principals p and q are *equivalent* in Π; *pc*, denoted Π; *pc*; *pc* $\Vdash p \equiv q$, if

$$\Pi; \textit{pc}; \textit{pc} \Vdash p \succcurlyeq q \text{ and } \Pi; \textit{pc}; \textit{pc} \Vdash q \succcurlyeq p.$$

Next, we define the *factorization* of two principals in a given context. For two principals, p and q, their factorization involves representing q as the conjunction of two principals $q_s \wedge q_d$ such that $p \succcurlyeq q_s$ in the desired context. Note that p need not act for q_d.

Definition 9 (Factorization)**.** A $(\Pi; \textit{pc})$-*factorization* of an ordered pair of principals (p, q) is a tuple (p, q_s, q_d) such that Π; *pc*; *pc* $\Vdash q \equiv q_s \wedge q_d$ and Π; *pc*; *pc* $\Vdash p \succcurlyeq q_s$.

A factorization is *static* if $\Pi = \varnothing$ (and thus $\mathcal{L} \vDash p \succcurlyeq q_s$).

Finally, the minimal factorization of p and q is a q_s and q_d such that q_s has greater authority and q_d has less authority than any other factorization of p and q in the same context.

Definition 10 (Minimal Factorization)**.** A $(\Pi; \mathit{pc})$-factorization (p, q_s, q_d) of (p, q) is *minimal* if for any $(\Pi; \mathit{pc})$-factorization (p, q'_s, q'_d) of (p, q),

$$\Pi; \mathit{pc}; \mathit{pc} \Vdash q_s \succcurlyeq q'_s \text{ and } \Pi; \mathit{pc}; \mathit{pc} \Vdash q'_d \succcurlyeq q_d$$

The minimal factorization (p, q_s, q_d) of p and q for a given Π and *pc* identifies the authority necessary to cause p to act for q. Because q_s is the principal with the greatest authority such that $p \succcurlyeq q_s$ and $q \equiv q_s \wedge q_d$, then speaking for q_d is sufficient authority to cause p to act for q since adding the delegation $p \succcurlyeq q_d$ would imply that $p \succcurlyeq q$. This intuition also matches with the fact that $\Pi; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q_d$ if and only if $q_d = \bot$, which is the case if and only if $\Pi; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q$. Observe also that minimal $(\Pi; \mathit{pc})$-factorizations are also trivially unique up to equivalence.

Since the q_d component of minimal factorization can be thought of as the "gap" in authority between two principals, we use q_d to define the notion of principal *subtraction*.

Definition 11 (Principal Subtraction)**.** Let (p, q_s, q_d) bet the minimal $(\Pi; \mathit{pc})$-factorization of (p, q). We define $q - p$ in $\Pi; \mathit{pc}$ to be q_d. That is, $\Pi; \mathit{pc}; \mathit{pc} \Vdash q - p \equiv q_d$. Note that $q - p$ is not defined outside of a judgement context.

Lemma 2 proves that minimal factorizations exist for all contexts and principals, so principal subtract is well defined.

Lemma 2 (Minimal Factorizations Exist)**.** *For any context* $(\Pi; \mathit{pc})$ *and principals* p, q*, there exists a minimal* $(\Pi; \mathit{pc})$*-factorization of* (p, q).

Proof. Given (p, q), we first let $q_s = p \vee q$. By definition, $\Pi; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq p \vee q$, and for all factorizations (p, q'_s, q'_d), $\Pi; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q'_s$ and $\Pi; \mathit{pc}; \mathit{pc} \Vdash q \succcurlyeq q'_s$, so $\Pi; \mathit{pc}; \mathit{pc} \Vdash q_s \succcurlyeq q'_s$.

Now let $D = \{r \in \mathcal{L} \mid \Pi; \mathit{pc}; \mathit{pc} \Vdash q \equiv q_s \wedge r\}$. All principals in $\mathcal{L}$ can be represented as a finite set of meets and joins of names in $\mathcal{N}$ so q and q_s are finite. Π is also finite, adding only finitely-many dynamic equivalences, so D is finite up to equivalence. Moreover, $q \in D$ trivially, so D is non-empty and thus we can define $q_d = \bigvee D$.

Now let (p, q'_s, q'_d) be any factorization of (p, q). We must show that $\Pi; \mathit{pc}; \mathit{pc} \Vdash q'_d \succcurlyeq q_d$. First we see that $\Pi; \mathit{pc}; \mathit{pc} \Vdash q \equiv q_s \wedge q'_d$. $\Pi; \mathit{pc}; \mathit{pc} \Vdash q_s \succcurlyeq q'_s$ gives us $\Pi; \mathit{pc}; \mathit{pc} \Vdash q_s \wedge q'_d \succcurlyeq q$ and the definitions of q_s and q'_d as parts of a factorization of (p, q) give us the other direction. Therefore, by construction, $q'_d \in D$, so by the definition of $\vee$ and q_d, $\Pi; \mathit{pc}; \mathit{pc} \Vdash q'_d \succcurlyeq q_d$. Thus we see that (p, q_s, q_d) is a minimal $\Pi; \mathit{pc}$-factorization of (p, q). □

We can now state precisely which trust relationships may change in a given information flow context.

Lemma 3 (Delegation Invariance)**.** *Let* $\Pi; \Gamma; \mathit{pc} \vdash e : s$ *such that* $e \longrightarrow e'$ `where` v*. Then there exist* $r, t \in \mathcal{L}$ *and* $\Pi' = \Pi, \langle r^\pi \succcurlyeq t^\pi \mid \mathit{pc}\rangle$ *such that* $\Pi; \Gamma; \mathit{pc} \vdash v : (r^\pi \succcurlyeq t^\pi)$ *and* $\Pi'; \Gamma; \mathit{pc} \vdash e' : s$*. Moreover, for all principals* p *and* q *if* $\Pi; \mathit{pc}; \mathit{pc} \nVdash \mathit{pc} \succcurlyeq \nabla(q^\pi) - \nabla(p^\pi)$*, then*

$$\Pi'; \mathit{pc}; \mathit{pc} \nVdash p^\pi \succcurlyeq q^\pi.$$

Proof. See Appendix B.1. □

First, Lemma 3 says that at each step of evaluation, there exists a Π' such that e' is well typed. More importantly, this Π' has a useful invariant. If *pc* does not speak for the authority required to cause q^π to delegate to p^π, then Π and Π' must agree on the trust relationship of p^π and q^π.

4.6.2 Noninterference

Lemma 3 is critical for our proof of *noninterference*, a result that states that public and trusted output of a program cannot depend on restricted (secret or untrustworthy) information. Our proof of noninterference for FLAC programs relies on a proof of subject reduction under a bracketed semantics, based on the proof technique of Pottier and Simonet [70]. This technique is relatively standard, so we omit it here, but complete proofs are found in Appendix B.1.

In other noninterference results based on bracketed semantics, including [70], noninterference follows almost directly from the proof of subject reduction. This is because the subject reduction proof shows that evaluating a term cannot change its type. In FLAC, however, subject reduction alone is insufficient; evaluation may enable flows from secret or untrusted inputs to public and trusted types.

To see how, suppose e is a well-typed program according to $\Pi; \Gamma, x : s; \mathit{pc} \vdash e : s'$. Furthermore, let H be a principal such that $\Pi; \mathit{pc} \vdash H \leq s$ and $\Pi; \mathit{pc} \nvdash H \leq s'$. In other words, x is a "high" variable (more restrictive; secret and untrusted), and e evaluates to a "low" result (less restrictive; public and trusted). In [70], executions that differ only in secret or untrusted inputs must evaluate to the same value, since otherwise the value would not be well typed. In FLAC, however, if the *pc* has sufficient integrity, then an `assume` term could cause $\Pi'; \mathit{pc} \vdash H \leq s'$ to hold in a delegation context Π' of a subterm of e. The key to proving our result relies on using Lemma 3 to constrain the assumptions that can be added to Π'. Thus noninterference in FLAC is dependent on H and its relationship to *pc* and the type s'.

For more precision, Theorem 2 describes noninterference on confidentiality and integrity separately. It states that for some principal H^{π} that flows to s but not ℓ `says bool`, if *pc* cannot cause $H^{\leftarrow}$ to speak for $\nabla(H^{\rightarrow})$ for confidentiality, or $\ell^{\leftarrow}$ for integrity, then an execution of e that differs only in the value of s-typed inputs, the

computed values must be equal.[10]

Theorem 2 (Noninterference)**.** *Let* $\Pi; \Gamma, x : s; pc \vdash e : \ell$ `says bool`*. If there exists some* H *and* π *such that*

1. $\Pi; pc \vdash H^\pi \leq s$
2. $\Pi; pc; pc \not\vdash H^\pi \sqsubseteq \ell^\pi$
3. *(a) if* $\pi = \rightarrow$ *then* $\Pi; pc; pc \not\vdash pc \succcurlyeq \nabla(H^\rightarrow) - H^\leftarrow$

   (b) if $\pi = \leftarrow$ *then* $\Pi; pc; pc \not\vdash pc \succcurlyeq (\ell - H)^\leftarrow$

then for all v_1, v_2 *with* $\Pi; \Gamma; pc \vdash v_i : s$*, if* $e[x \mapsto v_i] \longrightarrow^* v_i'$*, then* $v_1' = v_2'$*.*

Proof. By Lemma 3 and subject reduction on a bracketed semantics. See Appendix B.1 for details. □

Condition 1 identifies s as a "high" type—at least as restricted as H. Condition 2 identifies ℓ `says bool` as a "low" type, to which information labeled H should not flow. Conditions 3a and 3b identify *pc* as having integrity compared to the difference between $H^\leftarrow$ and the voice $H^\rightarrow$ or $\ell^\leftarrow$. Given these conditions, if e evaluates to v_1' when $x = v_1$ and v_2' when $x = v_2$, then $v_1' = v_2'$.

Noninterference is a key tool for obtaining many of the security properties we seek. For instance, noninterference is essential for verifying the properties of commitment schemes discussed in Section 4.4.1. The proofs of these properties are described in Appendix B.2.

[10] It is standard for noninterference proofs in languages with higher-order functions to restrict their results to non-function types (cf. [3, 70, 102]). In this paper, we prove noninterference for boolean types, encoded as `bool` $=$ (`unit` $+$ `unit`). With an appropriate equivalence relation on terms, this noninterference result can be lifted to more general types.

4.6.3 Robust declassification

Using our noninterference result, we obtain a more general semantic security property for FLAC programs. That property, *robust declassification* [96], requires disclosures of secret information to be independent of low-integrity information. Robust declassification permits some confidential information to be disclosed to an attacker, but attackers can influence neither the decision to disclose information nor the choice of what information is disclosed. Therefore, robust declassification is a more appropriate security condition than noninterference when programs are intended to disclose information.

Programs and contexts that meet the requirements of Theorem 2 trivially satisfy robust declassification since no information is disclosed. In higher-integrity contexts where the pc speaks for $H^{\rightarrow}$ (and thus may influence its trust relationships), FLAC programs exhibit robust declassification.

Following Myers *et al.* [64], we extend our set of terms with a "hole" term $[\bullet]$ representing portions of a program that are under the control of an attacker. We extend the type system with the following rule for holes with lambda-free types:

$$[\textsc{Hole}] \qquad \frac{\Pi; pc \vdash H^{\leftarrow} \leq t \qquad \Pi; pc; pc \Vdash H^{\leftarrow} \succcurlyeq \nabla(pc)}{\Pi; \Gamma; pc \vdash [\bullet] : t}$$

We write $e[\vec{\bullet}]$ to denote a program e with holes. Let an *attack* be a vector $\vec{a}$ of terms and $e[\vec{a}]$ be the program where a_i is substituted for $\bullet_i$. An attack $\vec{a}$ is a *fair attack* [96] on a well-typed program with holes $e[\vec{\bullet}]$ if the program $e[\vec{a}]$ is also well typed. Unfair attacks give the attacker enough power to break security directly, without exploiting existing declassifications. Fair attacks represent the power of the attacker over low-integrity portions of the program.

Theorem 3 (Robust declassification)**.** *Let $e[\vec{\bullet}]$ be a program such that*

1. $\Pi; \Gamma, x : s, \Gamma'; pc \vdash e[\vec{\bullet}] : \ell$ `says bool`

2. $\Pi; pc; pc \not\Vdash pc \not\succcurlyeq (\ell - h)^{\leftarrow}$.

Then for all attacks $\vec{a_1}$ *and* $\vec{a_2}$ *and all inputs* v *such that* $\Pi; \Gamma, x : s, \Gamma'; pc \vdash e[\vec{a_i}] : \ell$ `says bool` *and* $\Pi; \Gamma; pc \vdash v : s$*, if* $e[\vec{a_i}][x \mapsto v] \longrightarrow^* v_i'$*, then* $v_1' = v_2'$.

Proof. By Lemma 3 and a generalization of Theorem 2 under attacks. See Appendix B.1 for details. □

Our formulation of robust declassification in some sense more general than previous definitions since it permits some endorsements, albeit restricted to untrusted principals that cannot influence the trust relationships of $\ell^{\leftarrow}$, the integrity of the result. Previous definitions of robust declassification [64, 96] forbid endorsement altogether; *qualified robustness* [64] permits endorsement but offers only possibilistic security.

CHAPTER 5

FLAME: FLOW-LIMITED AUTHORIZATION WITH MONADIC EFFECTS.

This chapter presents Flame, a Haskell library for enforcing flow-limited authorization based on the Flow-Limited Authorization Calculus. Embedding the FLAC type system into an existing language requires an expressive type system in the target language. In particular, to create flow-safe authorization mechanisms in the style of FLAC, we need the ability to represent authorizations at the type level based on evidence terms. While a fully-fledged dependently typed language would provide more than enough expressiveness, we determined that Haskell's type system, along with some recent extensions (for example, [68, 95]), would be sufficient for our needs. These extensions make some dependently typed programming possible. Several of our implementation strategies were inspired by the exploration of dependently typed programming in Haskell by Lindley and McBride [53].

Flame also expands on the FLAC programming model to make building programs easier. While FLAC's type system supports polymorphic types, it does not have type-level variables that range over principals. This simplifies the formal presentation, but would require, for instance, a separate definition of the commitment scheme operations for each pair of principals that wished to use them. Thus, to be at all practical, we want Flame to support types that are polymorphic in the principals they refer to.

To motivate the kinds of programs we want to secure with Flame, consider the partial Haskell program in Figure 5.1 that protects a secret phrase with a password. After obtaining input from the user, the program calls `checkPass` to compare it to the password. If `checkPass` returns true, the secret is printed, otherwise an error is printed.

Is this program secure? It depends on the specification of `checkPass`. The expected implementation is probably the following:

```
checkPass guess = guess == "mypasswd"
```

```
secret :: String
secret = "mysecret"

inputPass :: IO String
inputPass = getLine

checkPass :: String -> Bool
checkPass guess = [          ]

main :: IO ()
main = do pass <- inputPass
          if checkPass guess then
            putStrLn secret
          else
            putStrLn "Incorrect password."
```

Figure 5.1: A Haskell program that protects a secret phrase with a password. The security of the program depends on whether returning `True` from `checkPass` means that the user is authorized to see the secret.

So `checkPass` returning `True` indicates that the user is authorized to see the secret phrase. However, another implementation might cause the above program to output the secret phrase inappropriately:

```
checkPass guess = guess /= "mypasswd"
```

Although this example is simplistic, it illustrates a common problem. The `checkPass` function is used as an authorization mechanism, but the interface and semantics of that mechanism are implicit. Misunderstanding the interface or semantics can lead to violations of security. The return value of `checkPass` is used to encode authorization as a Boolean value, but nothing prevents the programmer from writing a program that prints the secret without proper authorization.

A better approach would be to represent the authorization as a first-class value and prevent the disclosure of `secret` unless this evidence of authorization is provided. FLAC provides a core programming model for building such programs; The goal of

```
data Prin =
  Top
  | Bot
  | Name  String
  | Conj  Prin Prin
  | Disj  Prin Prin
  | Conf  Prin
  | Integ Prin
```

(a) The principal data type for run-time principals

```
data KPrin =
  KTop
  | KBot
  | KName Symbol
  | KConj  KPrin KPrin
  | KDisj  KPrin KPrin
  | KConf  KPrin
  | KInteg KPrin
  | KVoice KPrin
```

(b) The principal kind for type-level principals.

Figure 5.2: Flame principals

Flame is to use FLAC as a basis for secure programming in Haskell.

There are several challenges to realizing this goal. First among these challenges is how FLAC terms and types should be encoded in Haskell. Finding the right encoding in Haskell for FLAC delegation terms $\langle p \succcurlyeq q \rangle$ is challenging since the type is dependent on the value: $\langle p \succcurlyeq q \rangle$ has type $(p \succcurlyeq q)$. Haskell does not directly support dependent types. Another challenge is building a mechanism for Haskell that enforces the acts-for constraints the FLAC type system places on secure computation while still permitting new flows to be enabled via `assume`. Finally, whereas FLAC is a pure functional language, Flame must support effectful Haskell programs that perform input and output, mutate memory, and leverage foreign function interfaces.

Flame addresses these challenges by using an assortment of pre-existing extensions to Haskell that allow limited forms of dependently-typed programming. In addition, Flame provides an extension to the Haskell constraint solver for checking acts-for constraints.

5.1 Run-time and type-level principals

To embed FLAC-style types into Haskell, a first requirement is to represent principals both at run time and compile time. Figure 5.2a shows Flame's data type for representing run-time principals. Primitive principals are represented by a `String` and created using the constructor `Name`. The special principals `Top` and `Bot` correspond to $\top$ and $\bot$, respectively. Conjunctions (`Conj`), disjunctions (`Disj`), and authority projections (`Conf` and `Integ`) are similarly constructed. As in FLAC, we omit ownership projections (Section 3.2.2) for simplicity.

To represent these principals at the type level, Flame defines a principal *kind*. In the same way types classify terms, a kind classifies types. So defining a principal kind is a way of defining a new class of types just for principals. In Haskell, all terms (the expressions actually evaluated at run time) must have types in the `*` kind. This means all types in other kinds are uninhabited (no term exists with that type) and not represented at run time. However, types in the `*` kind may be *parameterized* by types of other kinds since these parameters are not represented at run time. Therefore, using types parameterized by principals in our principal kind allows us to enforce policies at compile type without incurring run-time overhead.

The principal kind is defined using the DataKinds [95] extension to GHC. DataKinds creates a kind by "promoting" the constructors in the `KPrin` data type, shown in Figure 5.2b. Each data constructor in `KPrin` becomes a type constructor for types in kind `KPrin`. These constructors can be used to construct type-level principals and instantiate type variables. For example, the following data type

```
data T (p::KPrin) a
```

has a type parameter that ranges over the principal kind, letting us construct types like `T KTop Int` and `T (KConf KBot) String`.

```
type (⊤)   = KTop
type (⊥)   = KBot
type N s   = KName s
type C p   = KConf p
type I p   = KInteg p
type p ∧ q = KConj p q
type p ∨ q = KDisj p q
type (∇) p = KVoice p

type p ⊔ q = (C p ∧ C q) ∧ (I p ∨ I q)
type p ⊓ q = (C p ∨ C q) ∧ (I p ∧ I q)

type Public    = C KBot
type Secret    = C KTop
type Trusted   = I KTop
type Untrusted = I KBot
type PT        = Public ∧ Trusted
type SU        = Secret ∧ Untrusted
```

Figure 5.3: Some useful type synonyms for Flame principals

We also define several type synonyms, presented in Figure 5.3, both for convenience and to bring Flame's type-level principal representation as close as possible to FLAC's.

The primary difference between `Prin` and `KPrin` is in the `Name` and `KName` constructors for primitive principals. `KName` principals are created using a member of the `Symbol` kind, which are the symbols known at compile time. The `Name` constructor creates a principal from any string value, and may not be known at compile time. For instance, string literals are known at compile time, so we can create a type-level principal `KName "Alice"` and a run-time principal `Name "Alice"`. For a string `arg` provided on the command-line, however, we can only create `Name arg`.

This restriction does not mean that type-level principals must always be written in concrete terms. Like other Haskell types, members of the `KPrin` kind may be represented abstractly using type variables. Suppose `n` is a type variable that ranges over the

`Symbol` kind. Then `KName n` is a type-level principal that is polymorphic in its name `n`.

Bridging the gap between run time and compile time

Being able to represent principals at the type level is helpful, but not actually sufficient: in many cases we want to associate a type-level principal with a run-time principal. We could do this informally by making sure that the dynamic principals in our computations always correspond to the types we use to label them, but any error that results in a discrepancy between the run-time principal and the type-level principal would make our verification mechanism unsound.

For this reason, Flame provides its own mechanism that associates a run-time principal with an existentially quantified, type-level principal. Only Flame library code is permitted to create and manipulate these associations, ensuring that run-time principals cannot become disassociated from their type-level representation. Following Lindley and McBride [53], we use the GHC extensions PolyKinds and GADT to define a data type for existential quantification, and then use this to promote a `Prin` to an instance of a special *singleton* type.

A singleton type is a type that has a unique inhabitant. For instance, the unit type `()` is a singleton type since it has only one inhabitant, also written `()`. Singleton types are useful for emulating dependently typed programming since they provide a link between values and types. By knowing that a function has a unit return type, we know the value returned must be `()`.

By generalizing this idea, we can provide a link between run-time and type-level principals. Figure 5.4 defines `SPrin`, a generalized algebraic data type (GADT) for principal singletons, along with additional notation definitions analogous to those for `KPrin`. In general, we use the convention that `SPrin` operations are prepended with an asterisk `*`.

```
data SPrin :: KPrin -> * where
  STop   :: SPrin KTop
  SBot   :: SPrin KBot
  SName  :: forall (n :: Symbol). Proxy n -> SPrin (KName n)
  SConj  :: SPrin p -> SPrin q -> SPrin (KConj p q)
  SDisj  :: SPrin p -> SPrin q -> SPrin (KDisj p q)
  SConf  :: SPrin p -> SPrin (KConf p)
  SInteg :: SPrin p -> SPrin (KInteg p)
  SVoice :: SPrin p -> SPrin (KVoice p)

(*→) p = SConf p
(*←) p = SInteg p

(*▽) p   = SVoice p
(*∧) p q = SConj p q
(*∨) p q = SDisj p q
(*⊔) p q = ((p*→) *∧ (q*→)) *∧ ((p*←) *∨ (q*←))
(*⊓) p q = ((p*→) *∨ (q*→)) *∧ ((p*←) *∧ (q*←))
```

Figure 5.4: A GADT defining singleton types for each member of KPrin and functions for notational convenience.

Each principal p in `KPrin` is uniquely represented by a value of type `SPrin p`. Therefore, if we can associate a run-time principal with the appropriate singleton, we would have a mechanism for reasoning about a value in type-level computations. Writing a function to make this association is tricky: given a `Prin` value, what type of `SPrin` should the function return? We know that there is *some* SPrin that represents the `Prin`, we just don't know which one it is at compile time.

The solution is to associate the `Prin` value with an *existentially qualified* `SPrin`. The data type `Ex`, defined in Figure 5.5, is a general data type for existential quantification. Next, the `promote` function associates a witness of `SPrin` for each `Prin` value.

Once we have the existentially quantified `SPrin` returned by `promote`, we can always extract the witness to get the specific `SPrin` associated with a principal. However, recall that we also want to ensure that we can easily maintain the association between the run-time principal and its type-level representation when desired. For this purpose,

```
data Ex (p :: k -> *) where
  Ex :: p i -> Ex p

promote :: Prin -> Ex SPrin
promote p =
  case p of
    Top         ->  Ex STop
    Bot         ->  Ex SBot
    (Name str)  ->  case someSymbolVal str of
                      SomeSymbol n -> Ex (SName n)
    (Conj p q)  ->  case promote p of
                      Ex p' -> case promote q of
                                 Ex q' -> Ex (SConj p' q')
    (Disj p q)  ->  case promote p of
                      Ex p' -> case promote q of
                                 Ex q' -> Ex (SDisj p' q')
    (Conf p)    ->  case promote p of Ex p' -> Ex (SConf p')
    (Integ p)   ->  case promote p of Ex p' -> Ex (SInteg p')
```

Figure 5.5: Promoting run-time principals to the type level. An SPrin witness is created for each run-time principal. Client code may extract this witness by pattern matching on the return value.

Flame provides the data type `DPrin`, which is essentially a `Prin` and `SPrin` pair in which the `SPrin` is the associated singleton for the `Prin` value. To ensure `DPrin` values cannot be accidentally (or maliciously) constructed with incorrect elements, `DPrin` values may only be created using the `withPrin` function, shown in Figure 5.6. Given a run-time principal and a function that accepts a polymorphic `DPrin` parameter, `withPrin` extracts the witness `p'`, pairs it with its run-time value, and provides it to the function.

Flame additionally defines algebraic operators on `DPrin` values, shown in Figure 5.7 that preserve the associations between the run-time principal and the singleton witness.

5.2 Expressing and solving acts for constraints

Flame uses *qualified types* to express information flow constraints. A qualified type $C \Rightarrow \tau$ is a type τ qualified by a context C. The context contains constraints that must

```
data DPrin p = UnsafeAssoc { dyn :: Prin, st :: SPrin p }

(<=>) :: Prin -> SPrin p -> DPrin p
p <=> sp = UnsafeAssoc p sp

withPrin :: Prin -> (forall p . DPrin p -> a) -> a
withPrin p f = case promote p of
                 Ex p' -> f (p <=> p')
```

Figure 5.6: Associating run-time and type-level principals with `withPrin`.

hold for the term to have type τ. Most Haskell programs use qualified types in some form. For instance, a function may require its arguments to be member of a particular type class to ensure that certain operations are defined.

Figure 5.8 illustrates such a function. The constraint `Eq a` requires that any instantiation of the type variable `a` be a member of the `Eq` type class. All members of the `Eq` type class define a function == for equality comparisons. By requiring `a` to be a member `Eq`, the compiler can assume that == is defined for any instantiation of `a`.

Flame defines *acts-for constraints*, a new type of constraint that may be specified in the context of a qualified type. These constraints specify the conditions required for a term to have type τ. to For instance, the qualified type $p \succcurlyeq q \Rightarrow \tau$ specifies that a term has type τ only if principal p acts for q, where p and q are type-level principals.

Acts-for constraints are useful for constraining information flows in a modular way. For instance, a function with type $p \succcurlyeq q \Rightarrow \tau \rightarrow \tau'$ may only be applied in a context where $p \succcurlyeq q$. Therefore, the function may assume $p \succcurlyeq q$ holds and permit q's information to flow to p within the function. This means we can use the context C of a qualified type $C \Rightarrow \tau$ as an analogue of the Π context in the FLAC typing rules.

The acts-for operation $\succcurlyeq$ is defined as an *empty closed type family*, which is an approach sometimes used (for example, Gundry [36]) to define new type-level constants that can be processed specially by a GHC compiler plug-in.

```
( ⊤ ) :: DPrin (⊤)
( ⊤ ) = Top <=> STop

( ⊥ ) :: DPrin (⊥)
( ⊥ ) = Bot <=> SBot

(^→) :: DPrin p  -> DPrin (C p)
(^→) p  = Conf (dyn p) <=> SConf (st p)

(^←) :: DPrin p  -> DPrin (I p)
(^←) p = Integ (dyn p) <=> SInteg (st p)

( ∧ ) :: DPrin p -> DPrin q -> DPrin (p ∧ q)
( ∧ ) p q = Conj (dyn p) (dyn q) <=> SConj (st p) (st q)

( ∨ ) :: DPrin p -> DPrin q -> DPrin (p ∨ q)
( ∨ )  p q  = Disj (dyn p) (dyn q) <=> SDisj (st p) (st q)

( ⊔ ) :: DPrin p -> DPrin q -> DPrin (p ⊔ q)
( ⊔ )  p q  = (Conj (Conf (Conj (dyn p) (dyn q)))
                           (Integ (Disj (dyn p) (dyn q))))
            <=> ((st p) *⊔ (st q))

( ⊓ ) :: DPrin p -> DPrin q -> DPrin (p ⊓ q)
( ⊓ ) p q  = (Conj (Conf (Disj (dyn p) (dyn q)))
                  (Integ (Conj (dyn p) (dyn q))))
           <=> ((st p) *⊓ (st q))

( ∇ ) :: DPrin p -> DPrin ((∇) p)
( ∇ ) p = voiceOf (dyn p) <=> SVoice (st p)
```

Figure 5.7: Additional `DPrin` operations.

```
f :: Eq a => a -> a -> String
f x y = if x == y then
          "Equal"
        else
          "Not equal"
```

Figure 5.8: The constraint `Eq a` requires that `a` be an instance of the type class `Eq`, which defines the operation ==.

```
type family (⋡) (p :: KPrin) (q :: KPrin) :: Constraint where
```

"Empty" because acts-for constraints are left abstract in the language and handled by a compiler plug-in (otherwise there would be instances defined after the `where` keyword), and "closed" because no other instances of in the type may be defined. That the type family is empty is not critical. Support in GHC for empty closed type families is relatively recent. For versions that do not support it, is it easy to create a single instance in the family using one or more axioms of the FLAM algebra. This means that constraints using only one of these axioms may be solved without appealing to our plug-in. Expressing all of the FLAM algebra in this way is not possible, hence the need for a compiler plug-in.

Flame also supports information flow constraints of the form $p \sqsubseteq q$. As discussed in Chapter 3, the FLAM principal algebra enables information flow constraints to be represented as acts-for constraints. Therefore, Flame defines the $\sqsubseteq$ operator as a type synonym for two acts-for constraints:

```
type (⊑) (p :: KPrin) (q :: KPrin) = ((C q ⋡ C p) , (I p ⋡ I q))
```

In other words, $p \sqsubseteq q$ requires that the confidentiality of q acts for the confidentiality of p, and the integrity of p acts for the integrity of q.

5.2.1 An algorithm for solving actsFor constraints

In this section we describe Flame's algorithm for finding proofs of acts-for relationships given a set of delegations between FLAM principals. This algorithm can be seen as a simplified version of the FLAM proof-search algorithm presented in Section 3.5. This algorithm only considers local delegation sets and assumes all delegations have the same label. The primary purpose of this simplified algorithm is for checking acts-for constraints in the Flame constraint solver. The algorithm assumes the set of delegations

and the desired result have the same label, but can be used as the core component of a more general search like that in 3.5. Flame provides two almost-identical implementations of this algorithm: one for solving run-time queries, and one for solving static constraints.

The goal of our algorithm is to find a proof that a principal p acts for another principal q, given some set of delegations. The general approach is to build a graph where each principal is a node and each delegation is an edge from a principal to its *superior*, the principal it delegates to. This graph is then used to answer acts-for queries by translating them to reachability queries on the graph.

The algorithm begins with a set of delegations and a query. When solving static constraints, these delegations are collected during type inference from the constraint context of qualified types. For run-time queries, the delegations are provided explicitly by querying code.

Given a set of delegations, each represented by a pair of principals $(\texttt{Prin}, \texttt{Prin})$, the first step is to convert the principals to FLAM normal form. A normalized principal has the form $p^{\rightarrow} \wedge q^{\leftarrow}$ where p and q are in "join of meets" form (for example, $(p_1 \vee p_2) \wedge (p_3 \vee p_4)$). The rewriting rules are essentially the same as those used in the FLAM formalization (see Appendix A.2). Our implementation is also based on the normalization code found in [8].

The next step is to expand the set of delegations into an equivalent set of simpler delegations. We split each delegation into a confidentiality delegation and an integrity delegation. Since the principals in each delegation have been normalized, this transformation is easy: $p_1^{\rightarrow} \wedge p_2^{\leftarrow} \succcurlyeq q_1^{\rightarrow} \wedge q_2^{\leftarrow}$ becomes $p_1^{\rightarrow} \succcurlyeq q_1^{\rightarrow}$ and $p_2^{\leftarrow} \succcurlyeq q_2^{\leftarrow}$. By keeping these confidentiality and integrity delegations separate from each other, we can answer the confidentiality and integrity components of an acts-for query separately.

We want to represent these delegation sets as two graphs where each node corre-

sponds to a principal, and an edge exists between nodes p and q if $p^\pi \succcurlyeq q^\pi$, where π is $\rightarrow$ for the confidentiality delegation set and $\leftarrow$ for the integrity delegation set. Ideally, p and q would always be primitive principals, since this would reduce the size of the graphs. Unfortunately, some delegations cannot be reduced to this form. For instance, $n_1^\pi \wedge n_2^\pi \succcurlyeq n_3^\pi$ cannot be reduced further; neither can $n_1^\pi \succcurlyeq n_2^\pi \vee n_3^\pi$.

Instead, we take the approach of simplifying the delegations as much as possible, and then adding special conjunction or disjunction vertices to the graph as needed. Simplified delegations all have the form $n_1 \wedge ... \wedge n_j \succcurlyeq n'_1 \vee ... \vee n'_k$. To obtain this form for a normalized (and split) delegation $p^\pi \succcurlyeq q^\pi$, we first distribute the join over p to convert from join-of-meet form to a meet of joins. In other words, $(n_1 \vee n_2) \wedge (n_3 \vee n_4)$ becomes $(n_1 \wedge n_3) \vee (n_1 \wedge n_4) \vee (n_2 \wedge n_3) \vee (n_2 \wedge n_4)$. For each disjunct $(n_1 \wedge ... \wedge n_j)$ of this meet, we create a new delegation $(n_1 \wedge ... \wedge n_j)^\pi \succcurlyeq q^\pi$. Then for each conjunct $(n'_1 \vee ... \vee n'_k)$ of q (still in join-of-meet form), we create $n_1 \wedge ... \wedge n_j \succcurlyeq n'_1 \vee ... \vee n'_k$. Once all delegations are in simplified form, we construct a graph with a vertex for each principal appearing in a delegation, and an edge for each delegation from the delegating principal to its superior.

We would like to use this graph to answer acts-for queries like $p \succcurlyeq q$ by determining if p is reachable from q, and therefore one of q's superiors. However, as its stands, some of q's superiors may not be reachable from p. This is because the relationships between conjunction and disjunction principals and their components is not represented in the graphs. Figure 5.9 illustrates some of these missing relationships. Principal $p \wedge q$ by definition acts for both p and q, and principals p and q each act for $p \vee q$. These relationships are based on the structure of the lattice, but are not yet represented in the graph. Furthermore, suppose p and q each delegate to r. Then the conjunction $p \wedge q$, by definition, also delegates to r.

These relationships derive from the structural relationships between principals, but

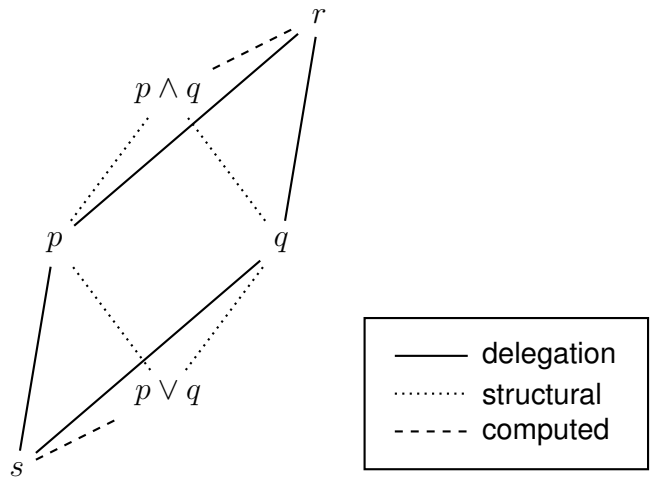


Figure 5.9: Computed relationships between principals.
Direct delegations imply additional relationships between principals that are structurally related. Flame's constraint checker iterates until the set of *computed* relationships reaches a fixed point.

also from the delegations between them. For instance, the set of superiors for $p \wedge q$ is the intersection of the superiors of p and q; the set of inferiors of $p \vee q$ is the intersection of inferiors of p and q. Computing these relationships from an initial delegation set may introduce new relationships that require further computation to propagate. For instance, if $r \succcurlyeq t$ (in addition to the relationships in Figure 5.9), then after $r \succcurlyeq p \wedge q$ is computed, the graph will still not represent the relationship $r \succcurlyeq p \wedge q \wedge t$. Consequently, we need to iterate the computation of these relationships until it reaches a fixed point. Since the set of principals is finite, such a fixed point always exists.

Once a fixed point is reached, we can answer an acts-for query $p \succcurlyeq q$ as follows. First, we normalize the principals and split into confidentiality and integrity queries $p'^{\pi} \succcurlyeq q'^{\pi}$ where p' and q' are normalized and split principals. We handle each of these queries separately—if both queries are true, then $p \succcurlyeq q$ is also true. Each p' and q' is represented as a join of meets of the form $(p'_i \vee ... \vee p'_{i+j})$ and $(q'_k \vee ... \vee q'_{k+l})$, where each p'_x and q'_y are primitive principals. For each $(q'_k \vee ... \vee q'_{k+l})$, we need to find some $(p'_i \vee ... \vee p'_{i+j})$ such that $(p'_i \vee ... \vee p'_{i+j})^{\pi} \succcurlyeq (q'_k \vee ... \vee q'_{k+l})^{\pi}$. Such a relationship holds

if for every disjunct p'_x for $x \in [i, i+j]$, there is some disjunct q'_y for $y \in [k, k+l]$ such that $p'^{\pi}_x \succcurlyeq q'^{\pi}_y$.

Since p'_x and q'_y are primitive principals, we can determine if $p'^{\pi}_x \succcurlyeq q'^{\pi}_y$ using the graphs we constructed. First, if $p'_x = \top$ or $q'_y = \bot$, we are done since $\top^{\pi} \succcurlyeq q^{\pi}$ for all q and $p^{\pi} \succcurlyeq \bot^{\pi}$ for all p. Otherwise, let G be the graph representing the confidentiality delegation set if we are processing the confidentiality query, or the graph for the integrity delegation set if we are processing the integrity query. If p'_x is reachable from q'_y in G, then $p'^{\pi}_x \succcurlyeq q'^{\pi}_y$.

5.2.2 Creating new delegations

We have discussed how acts-for constraints are specified and checked, but not how they are satisfied by client code. Some acts-for constraints are always satisfiable: those that require no delegations to derive. For instance, the following constraints always hold (for all p and q) since they derive from the axioms and rules of the FLAM principal algebra:

$$p \succcurlyeq p \qquad \top \succcurlyeq p \qquad p \succcurlyeq \bot \qquad p \succcurlyeq p^{\rightarrow} \qquad p \succcurlyeq p^{\leftarrow} \qquad p \wedge q \succcurlyeq p \qquad p \succcurlyeq p \vee q$$

More interesting relationships require evidence—a delegation. For example, suppose we have an expression e with the qualified type $\texttt{Alice} \succcurlyeq \texttt{Bob} \Rightarrow \tau$ how should the constraint $\texttt{Alice} \succcurlyeq \texttt{Bob}$ be satisfied? FLAC provides lexically-scoped extensions of the delegation context Π via `assume`. In FLAC, such a constraint could be satisfied with an acts-for term and an assume term $\texttt{assume}\ \langle \texttt{Alice} \succcurlyeq \texttt{Bob} \rangle\ \texttt{in}\ e$. So we would like a mechanism that permits us to *locally* satisfy (within some lexical scope) some or all of the acts-for constraints in a qualified type based on a term.

It turns out that our desired mechanism is an instance of the *configurations problem* as defined by Kiselyov and Shan [44]. The configurations problem is the problem of supporting multiple run-time configurations safely by ensuring statically that users cannot accidentally mix them up. In our setting the "configuration" is the principal lattice

that results from adding new delegations to the delegation context. Central to the security of Flame is that it ensure (statically) that programs only contain authorized flows as specified by this principal lattice.

Following Seipp [79], we use Kmett's `reflection` library [47], an implementation of Kiselyov and Shan's approach, to define the core mechanism we need to locally satisfy acts-for constraints. First, we define a type class `Pi` that represents the delegation context. This type class has no operations, and is merely a mechanism for collecting delegations statically:

```
class Pi del where
```

Delegation between principals is represented with the `AFType` data type. `AFType` values are constructed from a pair of principal singletons—a superior `sup` and an inferior `inf`—whose parameters are reflected in the type.

```
data AFType (p :: KPrin) (q :: KPrin) =
    AFType { sup :: SPrin p, inf :: SPrin q}
```

We then define an instance of `ReifiableConstraint` so that we can represent membership in `Pi` as a value. A reifiable constraint is a constraint that can be represented as a value. This type class is defined by Seipp and Kmett as follows.

```
class ReifiableConstraint p where
  data Def (p :: * -> Constraint) (a :: *) :: *
  reifiedIns :: Reifies s (Def p a) :- p (Lift p a s)
```

The associated data type `Def` is the type of the values representing these constraints, and `reifiedIns` is a proof that, for a constraint `p (Lift p a s)`, it is sufficient to satisfy the constraint `Reifies s (Def p a)`.

```
instance ReifiableConstraint Pi where
  data Def Pi (AFType p q) = Del { sup_ :: SPrin p, inf_ :: SPrin q}
  reifiedIns = Sub Dict
```

The data type `Def` is the type of the values that represent instances of `Pi`. Constructor `Del` lets us construct new delegations given principal singletons representing the superior and inferior principal. The following notational definitions make using delegations somewhat more intuitive.

```
type (:≽) p q  = Def Pi (AFType p q)
(≽) :: SPrin p -> SPrin q -> (p :≽ q)
(≽) p q = Del p q
```

We can now define an unrestricted (and therefore dangerous) version of `assume`. This implementation is a just a specialization of Seipp's `using` function, also found in Kmett's auxiliary reflection library [46]. More details about this function and the concepts it relies upon may be found in Seipp's tutorial [79] and in the Functional Pearl by Kiselyov and Shan's [44] it is based upon.

```
unsafeAssume :: forall a p q. (p :≽ q) -> ((p ≽ q) => a) -> a
unsafeAssume d m = reify d $ \(_ :: Proxy s) ->
  let replaceProof :: Reifies s (Def Pi (AFType p q)) :- (p ≽ q)
      replaceProof = trans proof reifiedIns
        where proof = unsafeCoerceConstraint ::
                        Pi (Lift Pi (AFType p q) s) :- (p ≽ q)
  in m \\ replaceProof
```

Given a delegation of type `(p :≽ q)` and a term with qualified type `((p ≽ q) => a)`, `unsafeAssume` returns a term of type `a`. In other words, the constrain `(p ≽ q)` is satisfied using the delegation as evidence. This is done by *reifying* the delegation `d` as an instance of `Pi` over the term `m`. The term `replaceProof` asserts that we can use this instance to satisfy the constraint `(p ≽ q)`. From `reifiedIns`, we can safely replace `Pi (Lift Pi (AFType p q) s)` with `Reifies s (Def p a)`, allow the reification of `d` via the `reify` function from the reflection library. The term `proof` goes a step further and declares (via `unsafeCoerceConstraint`) that replacing the constraint `(p ≽ q)` with the constraint `Pi (Lift Pi (AFType p q) s`. This allows the constraint in the qualified type to be satisfied by the reified delegation.

The usage of the unsafe function `unsafeCoerceConstraint` is justified since we are explicitly defining an additional way to satisfy acts-for constraints. Resolving an acts-for constraint using an instance of `Pi` is analogous to deriving an acts-for relationship in FLAC using the R-ASSUME rule defined in Section 4.3.

The `unsafeAssume` function is unsafe for a different reason, though, since it permits acts-for constraints to be satisfied without restriction. This makes it unsafe for client code to use this function directly. In Section 5.3, we present a safe interface to `unsafeAssume` that uses flow-limited authorization to ensure only authorized delegations can enable new flows.

5.3 Enforcing information flow control with acts-for constraints

The Flame library uses acts-for constraints on type-level principals to enforce information flow constraints on sensitive computations. The basic approach is to wrap sensitive information in an abstract data type that includes a level representing the confidentiality and integrity of the information. Flame provides operations on these abstract data types that form monad-like type classes similar to the `bind` and η_p operations of FLAC presented in Chapter 4. Any computation that uses the wrapped information must prove that the constraints are satisfied.

In Haskell, side-effecting computation occurs in the `IO` monad. The bind and return operations of the `IO` monad sequence the IO actions that occur in a Haskell program. Because Haskell expressions are evaluated lazily, IO actions are only processed if they are "used" by the program. For example, consider the following program.

```
name :: String
name = "Alice"

main :: IO ()
main = let sayhi   = hPutStr stdout "Hello " in
       let sayname = hPutStr stdout name in
       do sayhi
          sayname
```

This program binds two `IO` actions to the variables `sayhi` and `sayname`, and then applies them in sequence to print "Hello Alice". The function `putStr`, which prints a string to `stdout`, has type `String -> IO ()`. Haskell's `do` syntax is sugar for monadic operations; here, it applies the expressions in sequence.

The two `IO` actions bound to `sayhi` and `sayname` are only executed because they are composed in the `IO` computation that is returned by the `main` function. Suppose we omitted `sayname`:

```
main :: IO ()
main = let sayhi   = hPutStr stdout "Hello " in
       let sayname = hPutStr stdout name in
       sayhi
```

This program only prints "Hello", which might surprise those accustomed to eager evaluation and non-pure languages. The `sayname` variable is never used, so the computation it represents is never evaluated; hence no side effects occur. Using the `IO` monad for all side-effecting computation is thus a mechanism for both sequencing the ordering of side effects and specifying at the type level which computations may have side effects.[1]

The `Lbl` abstract data type associates values and pure computations with a label representing the confidentiality and integrity of the value or the result of the computation.

```
data Lbl (l::KPrin) a
```

The type parameter `l` has kind `KPrin` and `a` is a type variable for the type of the protected value. There are two primary operations on `Lbl`: `label` and `bind`.

[1] Haskell does provide "unsafe" functions like `unsafePerformIO` that permit `IO` actions without representing them in the type. We assume programs using Flame do not use these functions. Safe Haskell [84] is an approach to enforcing these assumptions.

```
label :: a -> Lbl l a
bind :: (l ⊑ l') => Lbl l a -> (a -> Lbl l' b) -> Lbl l' b
```

The `label` operation creates new labeled values whereas `bind` is used to perform computations on them. Given a labeled value of type `Lbl l a` and a function from values of type `a` to labeled values of type `Lbl l' b`, `bind` unwraps the labeled value and applies the function. The constraint `(l ⊑ l')` indicates that the label of the result must be at least as restrictive as `l`.

We can use the above core operations to define another useful operation for labeled values. The `relabel` operation takes a value labeled at `l` and returns a value labeled at `l'`, where `(l ⊑ l')`.

```
relabel :: (l ⊑ l') => Lbl l a -> Lbl l' a
relabel x = bind x label
```

The `Lbl` data type is designed for enforcing information security in pure computations. For example, consider the labeled factorial function below.

```
factorial :: Lbl l Int -> Lbl l Int
factorial ln = bind ln $ \n ->
                 label $ fact n
  where fact 0 = 1
        fact n = n * fact (n - 1)
```

This function binds a labeled value `ln`, then computes the factorial of it using the helper function `fact`.

In fact, the above pattern is generalizable to any pure function. This operation is often called `fmap`, for "functorial map" since type constructors (in this case `Lbl l`) having such operations are functors.

```
fmap :: (a -> b) -> Lbl l a -> Lbl l b
fmap f ln = bind ln $ \n ->
                 label $ f n
```

`Lbl` operations prevent `IO` operations from using protected values since `IO` actions remain wrapped in the `Lbl` data type. Flame provides no mechanism for extracting the

```
lift   :: Lbl l a -> IFC e pc l a

apply  :: (pc ⊑ pc') =>
          IFC e pc l a
          -> (Lbl l a -> IFC e pc' l' b)
          -> IFC e pc' l' b

ebind  :: (l ⊑ l', l ⊑ pc) =>
          Lbl l a
          -> (a -> IFC e pc l' b)
          -> IFC e pc' l' b
```

Figure 5.10: Core IFC operations.

IO computation from the Lbl data type, so the side effects will never be executed. The following program fragment contains an IO action in a labeled computation.

```
secret :: Lbl Alice String
secret = label "Alice's secret"

main = IO ()
main = let leak = bind secret $
                        \s -> label $ hPutStr stdout s in
       ??? -- no way to use leak to get a value in IO
```

Whereas the variables sayhi and sayname in the previous examples have type IO (), the variable leak has type Lbl Alice (IO ()). The Lbl operations provided by Flame require that any computation that uses leak (via bind) must return a labeled type Lbl l a, where Alice ⊑ l. Consequently, there is no way for main to use leak to return a value of type IO ().

Most programs, of course, have side effects—including those that process sensitive information. The Lbl data type and its operations enforce information security for pure computation, but they do not support these programs. Therefore, most computation in Flame occurs in IFC, an abstract data type that is similar in spirit to Lbl, but supports secure computations with side effects.

The type IFC e pc l a associates a label l with the result of a computation of

type a, similar to the Lbl l a type. IFC also associates a label pc that bounds the confidentiality and integrity of side-effects, for some effect e. Flame provides three core operations for computing with IFC: lift, ebind, and apply, whose types are presented in Figure 5.10. The lift operation lifts a (pure) labeled term into the IFC data type. Whereas Lbl only required a single computation operation for pure computation (bind), IFC provides both ebind and apply. These two operations separate the concerns of composing effectful computations (apply) and using labeled values in effectful computation (ebind).

The apply operation corresponds to the FLAC APP rule for function application. It applies an IFC value to an effectful function on Lbl values. The ebind operation, for "effectful bind," enables effectful computation with labeled values. Given a labeled term of type Lbl l a and a function with effects in e of type a -> IFC e pc l' b, ebind returns the result of applying the function to the unlabeled value. The constraints l $\sqsubseteq$ l' and l $\sqsubseteq$ pc ensure that both the label of the result (l') and the label bounding side-effects (pc) protect the labeled value.

Using the above operators, we can derive monadic operators similar to those in FLAC. These operators are presented in Figure 5.11. Flame's protect function corresponds to the monadic unit η operator in FLAC. Given a term of any type, protect labels the term and lifts it into IFC. The reprotect operation is a function analogous to the relabel operation on Lbl types. It allows the side-effect label (pc) and the result label (l) to be relabeled to more restrictive policies.

The use function corresponds to a FLAC bind term. Its constraints implement FLAC's BINDM typing rule. Given a protected value of type IFC e pc l a and a function on a with return type IFC e pc' l' b, use returns the result of applying the function, provided that l $\sqsubseteq$ l' and (pc $\sqcup$ l) $\sqsubseteq$ pc'.

These operations illustrate the core difference between enforcing information flow

```
protect :: a -> IFC e pc l a
protect = lift . label

reprotect :: (l ⊑ l', pc ⊑ pc') => m n pc l a -> m n pc' l' a
reprotect x = apply x $ \x' -> ebind x' (protect :: a -> m n SU l' a)

use :: (l ⊑ l', (pc ⊔ l) ⊑ pc') =>
       IFC e pc l a
       -> (a -> IFC e pc' l' b)
       -> IFC e pc' l' b
use x f = apply x $ \x' -> ebind x' f
```

Figure 5.11: Derived IFC operations

control in a pure setting and in an effectful one. In a pure setting, we need only ensure the result label protects the security of information the computation depends upon. In an effectful setting, we must ensure that composing effectful computations preserves the security as well.

The pc bound is used to constrain what effects may occur in an IFC computation. For example, the Flame provides the following binding for hPutStr:

```
hPutStr :: (pc ⊑ l) => IFCHandle l -> String -> IFC IO pc SU ()
```

The type IFCHandle l represents a file handle with label l, meaning that information read from or written to this handle should be protected with confidentiality and integrity l. The effect parameter e is instantiated with the IO effect since hPutStr outputs a string. The constraint pc ⊑ l establishes this label as an upper bound for the pc of the IFC type the call to hPutStr occurs within. The result has unit type so, for convenience, the label on the result is given the most restrictive label SU to simplify satisfying the constraints of the any enclosing computation. Flame provides similar wrappers for other common functions in the System.IO module. A selection of these is listed in Appendix C.1.

Examining the implementations of reprotect and use illustrates how the more

primitive `apply` and `ebind` operations are used to manipulate labeled data in Flame. Although both `use` and `reprotect` are based on `apply` and `ebind`, `use` is more restrictive than `reprotect` since it requires both `pc` and `l` to flow to `pc'`. In `use`, satisfying these two constraints is sufficient to satisfy the constraints of `apply` and `ebind` which are used to implement `use`. Whereas `use` may invoke an arbitrary function `f`, in `reprotect`, the bound value is immediately passed to `protect`. This function is effectively pure since it is typed with pc `SU`, the most restrictive information flow policy. Since any label flows to `SU`, the constraints for `reprotect` are less restrictive than `use`.

Perhaps the most significant difference between `Lbl` and `IFC` is that `IFC` provides a mechanism for extracting the result of an effectful computation from the `IFC` data type. The function `runIFC` takes an `IFC` term and returns the effectful computation with a labeled result.

```
runIFC :: IFC e pc l a -> e (Lbl l a)
```

Now we can output Alice's secret to file handle that protects Alice's information.

```
type Alice = KName "Alice"
alice :: SName Alice
alice = SName (Proxy :: Proxy "Alice")

sec :: Lbl Alice String
sec = label "Alice's secret"

stdout :: IFCHandle Alice
stdout = mkStdout alice

main :: IO (Lbl l' ())
main = runIFC $ ebind sec $ hPutStrLnx alice stdout
```

The expression `hPutStrLnx alice stdout` has type `String -> IFC IO Alice SU ()`. The function `hPutStrLnx` is just a version of `hPutStrLn` that allows us to specify the `pc` of the computation explicitly.[2]

[2]The Flame constraint solver does not currently instantiate type variables to satisfy bounds conditions, so an explicit label is sometimes necessary to instantiate type variables. Extending the solver to find instantiations of type variables that satisfy all acts-for constraints in a context would remove the necessity of

5.3.1 Safely enabling new flows

In Section 5.2.2 we presented `unsafeAssume`, which extends the delegation context with new delegations, thereby enabling new flows. We now describe how we can use acts-for constraints and the above abstractions to provide a safer interface to this mechanism.

The `assume` function restricts an extension of the delegation context to protected computations `IFC e pc l a`. Furthermore, for an extension `(p ≽ q)`, it requires that `pc` speak for `q`, and that the voice of `p`'s confidentiality speak for `q`'s confidentiality. These constraints correspond to those in ASSUME.

```
assume :: (pc ≽ (▽) q, (▽) (C p) ≽ (▽) (C q)) =>
          (p :≽ q) -> ((p ≽ q) => IFC e pc l a) -> IFC e pc l a
assume = unsafeAssume
```

The Flame library exports only this more restrictive version of `assume`, ensuring that all new flows are authorized.

5.4 Secure programming with Flame

Monadic operations are use extensively in Haskell programs. To ease the syntactic burden of explicit bind and return operations, Haskell provides syntactic sugar for them in the form of its "do-syntax". This syntax enable more concise expression of monadic computation. Unfortunately, since neither `IFC` nor `Lbl` form a `Monad` instance, this syntax is not directly available for Flame code.

We can, however, redefine the operations that do-syntax desugars to by using the extension `RebindableSyntax`. This approach has some drawbacks since it requires choosing a single domain to use do-syntax for: either `Monad` instances or Flame. However, this choice can be made for each file or even for each do block. We anticipate the

specifying the pc explicitly. We expect that the "greatest lower bounds" algorithm used by Jif's solver [65] should be relatively straightforward to port to Flame.

effects do-syntax is most commonly used for, like `IO`, will often be wrapped by the `IFC` transformer anyway (for example, `IFC IO pc l a`), making the right choice obvious. For files that use Flame extensively, we provide the module `Flame.Prelude` which exports the following definitions.

```
return = protect
(>>=) = use
m >> a = apply m $ \_ -> a
```

With these bindings, the following Flame program

```
do hPutStrLnx pc stdout "What is your name?"
   name <- hGetLinex pc stdin
   hPutStrLnx pc stdout "Hello " ++ name
```

desugars to

```
apply (hPutStrLnx pc stdout "What is your name?") $ \_ ->
use (hGetLinex pc stdin) $ \name ->
hPutStrLnx pc stdout "Hello " ++ name
```

Using do-syntax with Flame is more concise, but also makes using Flame more natural for experienced Haskell programmers. While the types of the rebound operations are more complex, the operations are analogous to their `Monad` counterparts and help leverage the programmers familiarity with `Monad` to properly use Flame's abstractions.

Figure 5.12 presents a new version of the password checker from Figure 5.1 written in Flame. Flame makes explicit the connection between the authorization check performed by `checkPass` and the disclosure of `secret`. Instead of encoding authorization as a Boolean return value, `checkPass` returns *evidence* of the authorization in the form of two delegations which are used to enable the flow from Alice to the user. It is impossible to leak the secret in an unauthorized way since without the delegations returned by `checkPass`, the secret may not flow to the user's console.

The password checker uses a design pattern useful for many situations. The entry point of the program, shown in Figure 5.13, constructs a principal for the user invoking the program, in this case using a Unix system call to obtain the effective username

```
secret :: Lbl Alice String
secret = label "mysecret"

checkPass :: SPrin client
        -> String
        -> IFC IO (I Alice) (I Alice)
           (Maybe (Voice client :≽ Voice Alice, client :≽ Alice))
checkPass client guess =
   {- Declassify the comparison with the password -}
   assume ((bottom*←) ≽ (alice*←)) $
   assume ((bottom*→) ≽ (alice*→)) $ do
   pwd <- liftx (alice*←) password
   protect $
     if pwd == guess then
       Just $ ((*▽) client ≽ (*▽) alice, client ≽ alice)
     else
       Nothing

secure_main :: DPrin user
        -> IFCHandle (I user)
        -> IFCHandle (C user)
        -> IFC IO ((C user) ∧ (I Alice)) SU ()
secure_main userprin stdin stdout =
  let user = (st userprin) in
  let pc = (user*→) *∧ (alice*←) in
  do pass <- inputPass user stdin stdout
     mdel <- checkPass user pass
     case mdel of
       Just (vdel,del) ->
         {- Use the granted authority to print Alice's secret -}
         assume vdel $ assume del $ do
           secret' <- liftx pc secret
           hPutStrLnx pc stdout secret'
       Nothing ->
         hPutStrLnx pc stdout "Incorrect password."
```

Figure 5.12: A Flame version of the password checker example. In this version, `checkPass` returns *evidence* of authorization which is used to enable the flow from Alice to the user.

```
main :: IO ()
main = do
  username <- getUserName
  withPrin (Name username) $ \user ->
    let user' = (st user) in
    let stdin  = mkStdin (user'*←) in
    let stdout = mkStdout (user'*→) in do
      _ <- runIFC $ secure_main user stdin stdout
      return ()
```

Figure 5.13: This Flame entry point creates a principal for the user running the program and uses it to create wrapped `IFCHandle`s for `stdin` and `stdout`.

of the current process. This principal is then used to create `IFCHandle` wrappers for `stdin` and `stdout`. The principal and handles are then passed to a *secure entry point* `secure_main` which characterizes the information flow constraints placed on the program.

The precise signature for `secure_main` is chosen by the developer to characterize the initial context of the program. In this case, the program is checking the password on behalf of Alice, so the entry point has Alice's integrity. Labeling `secure_main` with Alice's integrity is a form of endorsement (from Alice) stating that the password checker may act on her behalf. A more sophisticated design pattern might use code signatures [58] or leverage other Unix access control features like `setuid` bits to ensure this endorsement is authorized. Note however, that this implicit endorsement does not extend to the *input* to the program. The signature permits an untrusted user to invoke `secure_main` from the command line, but any input the user provides remains untrusted.

5.4.1 Flow-limited authorization with Macaroons

Macaroons are bearer credentials that generalize token-based mechanisms like session cookies [10]. A macaroon may have one or more caveats that attenuate the authority

granted to the bearer. This makes it easy to authorize clients in a decentralized way. For example, Alice can share a single photo from an online album by placing a caveat on the macaroon for the album that limits the possessor's access to the shared photo.

We discussed in Chapter 4 how mechanisms like macaroons can introduce information security vulnerabilities when used improperly. Using Flame, we have built an interface to a popular macaroons library, libmacaroons [32], that prevents these vulnerabilities using flow-limited authorization. In the process of building this interface, we also built Haskell bindings for libmacaroons, which are useful on their own.

Figures 5.14 and 5.15 present our Haskell API for libmacaroons. The functions for creating and manipulating macaroons are shown in Figure 5.14, along with their specifications. A macaroon contains a location "hint" for the service issuing the macaroon, a *key identifier*, a list of *caveats*, and a signature . Macaroon are initially created with the `create` function, which accepts the location, a root key, and the identifier for that key. Caveats may then be added with the `addFirstPartyCaveat` and `addThirdPartyCaveat`. A first-party caveat is a caveat that is dispatched by the issuer of the macaroon; a third-party caveat is dispatched by obtaining a macaroon from another service and binding it to the caveated macaroon using the `prepareForRequest` function.

The functions for verifying macaroons are shown in Figure 5.15. A service authorizes a request by creating a verifier with `createVerifier`. The service then adds to this verifier relevant predicates satisfied by the request using `satisfyExact`. For example, if the request is for a photo with id number 42, the service might add the predicate *"photo_id = 42"*.

A macaroon is verified by calling `verify` with the verifier, the macaroon, a list of dispatch macaroons (for third-party caveats), and the key associated the key identifier of the macaroon. This function verifies the macaroon's signature and checks that all

```
{-- Create a new macaroon. --}
create :: ByteString --^ location
       -> ByteString --^ key
       -> ByteString --^ key identifier
       -> (Macaroon, ReturnCode)
{-- Create a copy of a macaroon --}
copy :: Macaroon -> (Macaroon, ReturnCode)
{-- Get a user-friendly description of a macaroon. --}
inspect :: Macaroon -> (String, ReturnCode)
{-- Validate a macaroon's format --}
validate   :: Macaroon -> Bool
{-- Extract a macaroon's location string --}
location   :: Macaroon -> ByteString
{-- Extract a macaroon's key identifier string --}
identifier :: Macaroon -> ByteString
{-- Extract a macaroon's signature string --}
signature :: Macaroon -> ByteString
{-- List a macaroon's third-party caveats. --}
thirdPartyCaveats :: Macaroon
                  -> ([(ByteString, ByteString)], ReturnCode)
{-- Serialize a macaroon --}
serialize :: Macaroon --^ Macaroon to serialize
          -> MacaroonFormat --^ Serialization format
          -> (ByteString, ReturnCode)
{-- Deserialize a macaroon --}
deserialize :: ByteString -> (Macaroon, ReturnCode)
{-- Add a first-party caveat. --}
addFirstPartyCaveat :: Macaroon
                    -> ByteString --^ the caveat
                    -> (Macaroon, ReturnCode)
{-- Add a third-party caveat. --}
addThirdPartyCaveat :: Macaroon
                    -> ByteString --^ location
                    -> ByteString --^ key
                    -> ByteString --^ key identifier
                    -> (Macaroon, ReturnCode)
{-- Bind a macaroon to dispatch a third-party caveat. --}
prepareForRequest :: Macaroon --^ root macaroon
                  -> Macaroon --^ macaroon to bind
                  -> (Macaroon, ReturnCode)
```

Figure 5.14: libmacaroons bindings for macaroon creation and manipulation.

```
{-- Create a macaroon verifier --}
verifierCreate :: IO Verifier
{-- Verify a macaroon --}
verify :: Verifier
       -> Macaroon    --^ macaroon to verify
       -> ByteString  --^ macaroon key
       -> [Macaroon]  --^ dispatch macaroons
       -> IO (Bool, ReturnCode)
{-- Satisfy potential caveats with an exact string. --}
satisfyExact :: Verifier
             -> ByteString
             -> IO (Bool, ReturnCode)
{-- Satisfy potential caveats with a function. --}
satisfyGeneral :: Verifier
               -> (String -> IO Bool)  --^ a function that checks
                                       --^ if a caveat is satisfied
               -> IO (Bool, ReturnCode)
```

Figure 5.15: libmacaroons bindings for macaroon verification.

caveats of the macaroon are satisfied. If, for instance, the macaroon has a caveat that requires *"photo_id = 42"*, then `verify` succeeds if the verifier has a predicate that matches this string. Each first-party caveat of the macaroon must have a predicate that dispatches it, and each third-party caveat must have a dispatch macaroon that is verified recursively.

Dispatching caveats by comparing strings is efficient, but some predicates are not easily expressed by an exact string. For this situation, libmacaroons provides `satisfyGeneral` which accepts a function that decides whether a given caveat is accepted. For example, a macaroon may have a timeout caveat that says it expires after a certain time, expressed by the string *"time < 2016-09-16 16:28"*. To dispatch this caveat, a service might use `satisfyGeneral` to add a function like `checkTime`, shown in Figure 5.16, which compares the current time to the expiration time.[3] If the expiration time has passed, or if the caveat is unrelated to the timeout caveat format, `checkTime`

[3]The `checkTime` function is adapted from a similar python example in the libmacaroons tutorial [32].

```
checkTime :: String -> IO Bool
checkTime caveat =
  if "time < " `isPrefixOf` caveat then
    case getTime caveat of
      Just when -> do
        now <- getCurrentTime
        return (utctDay now < when)
      Nothing -> False
  else False
 where
   getTime = parseTimeM True defaultTimeLocale
               "%Y-%m-%d %H:%M" $ drop 7
```

Figure 5.16: `checkTime`: A predicate function for dispatching timeout caveats.

returns False. Note that returning False doesn't mean the macaroon cannot be verified, only that this function cannot dispatch this caveat. The caveat may still be dispatched by an exact predicate or a different predicate function.

Securing libmacaroons with Flame

One difference worth noting between the functions in Figures 5.14 and 5.15 is that the macaroon functions are pure: all macaroons are immutable values. In contrast, a verifier may be imperatively updated by `satisfyExact` and `satisfyGeneral`, hence the verification functions are in the `IO` monad.

Flame's design makes it particularly easy to incorporate pure code with little or no change. In this case, creating a Flame-enabled library for libmacaroons required creating bindings for only the macaroon verification functions. The API for these bindings is presented in Figure 5.17. All of the other libmacaroons bindings can be used by Flame programs directly.

Note that using the libmacaroons bindings for the functions in Figure 5.17 is not dangerous. Using the verification functions would generate results in the `IO` monad which cannot be extracted from a protected computation. Flame bindings extend the

```
verifierCreate :: SPrin pc
               -> SPrin l
               -> IFC IO pc' pc' (IFCVerifier pc l)

verify :: (pc ⊑ pc') =>
          IFCVerifier pc' l
          -> Macaroon
          -> Lbl l' ByteString
          -> [Macaroon]
          -> IFC IO pc l (Bool, ReturnCode)

satisfyExact :: (pc ⊑ l) =>
                IFCVerifier pc' l
                -> ByteString
                -> IFC IO pc l (Bool, ReturnCode)

satisfyGeneral :: (pc ⊑ l) =>
                  IFCVerifier pc' l
                  -> (String -> IFC IO pc' l Bool)
                  -> IFC IO pc l (Bool, ReturnCode)
```

Figure 5.17: Flame API for libmacaroons verification functions.

trusted computing base, and so can make principled use of the `IFC` type's underlying constructors to lift the `IO` actions into the `IFC` computation type.

Similar to the `IFCHandle` type used to wrap `stdin` and `stdout` in Section 5.4, Flame wraps the verifier in a type, `IFCVerifier`, that is used to restrict side effects. Like `IFCHandle`, `IFCVerifier` has a parameter to limit the restrictiveness of the predicates added to the verifier. Because these predicates are used to dispatch caveats during verification, the result of verification may reveal information about them. The parameter `l` is used to bound the information revealed by storing a string with `satisfyExact` or a function with `satisfyGeneral`. It also bounds the information revealed by executing the functions added with `satisfyGeneral`.

Note also that `satisfyGeneral` functions may have side effects. Executing this code during verification may reveal information about the context that `verify` is called

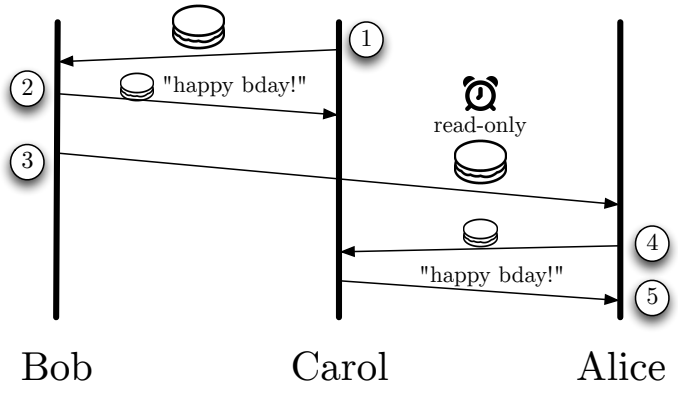


Figure 5.18: Embargoing secret messages with macaroons. Bob adds caveats to the macaroon he receives from Carol to give Alice access only on or after her birthday.

within. Hence, `pc` is used to bound this context, ensuring all predicate verification functions have side-effects that are safe to execute in this context.

Example: embargoed secret messages

To demonstrate Flame macaroons in action, we'll expand on our earlier password example. In this version, Bob uses Carol to host messages that he shares with his friends. He wants to post a birthday message for Alice, but allow her to see it only on or after her birthday.

Figure 5.18 illustrates the intended sequence of messages exchanged between Bob, Carol, and Alice. In (1), Carol gives Bob a macaroon for the hosted message. Using this macaroon in (2), Bob updates the message to be a birthday message for Alice. Next, Bob adds a caveat giving Alice read-only access, but not until the date of her birthday. Bob gives this macaroon to Alice in message (3). When presented to Carol by Alice in (4), Carol checks whether the current date is the same as or later than the date in the caveat. If so, she gives Bob's birthday message to Alice in message (5).

There are two secrets of concern here. First is the secret message for Alice hosted by Carol. Bob and Carol are permitted to read the message, but Alice should only be

able to read it with the macaroon from Bob, which can only be verified on or after her birthday. The other secret is Alice's birthday. The macaroon Bob issues to Alice has a caveat based on Alice's birthday, so by presenting that macaroon to Carol, Alice reveals her birthday to Carol.

Credentials-based systems like Macaroons are designed to protect the first kind of secret, the message for Alice, by preventing those without appropriate credentials from accessing the secret. They do not protect the second kind of secret, Alice's birthday. The reason is that they do not constrain what information a credential may derive from. This means that a credential intended for Alice to present to Carol can depend on secret information (Alice's birthday) without Alice's knowledge, creating a covert channel.

We have implemented this example in Flame using `IFCHandle`s to represent communication channels between Alice, Bob, and Carol, and an `IFCRef`[4] to store the updatable message. The channels restrict the confidentiality and integrity of messages exchanged between principals. The complete source is found in Appendix C.2.

For instance, Figure 5.19 illustrates `carolUpdateMessage`, which implements the code invoked by message (2) in Figure 5.18. Carol receives a macaroon from a client, creates a verifier with dispatch predicates, and attempts to verify the macaroon. Input from the client is labeled `(C (p ∨ Carol) ∧ I p)`, so it has the integrity of the client, `p`. The message has type `IFCRef Carol String`, which means it is a mutable reference to a `String` with Carol's confidentiality and integrity. Therefore, information that derives from the client's input cannot flow to the message unless the macaroon is verified (line 13) and an `assume` term delegates Carol's integrity to the client (line 15).

Figure 5.20 presents `bobOutputMacaroon`, which creates a new macaroon with caveats for Alice using the macaroon from Carol. This macaroon is provided to Alice over an output channel with confidentiality `Alice ∨ Bob ∨ Carol` since the maca-

[4] `IFCRef` is a wrapper for the System.IO mutable reference type `IORef`. Appendix C.1 lists the API for working with `IFCRef` values.

```
1  carolUpdateMessage :: SPrin p
2                     -> IFCHandle (C (p ∨ Carol) ∧ I p)
3                     -> IFCRef Carol String
4                     -> IFC IO Carol SU ()
5  carolUpdateMessage p from_p message =
6    do (mac, err) <- inputMac
7       if err /= MacaroonSuccess then do
8         error "Could not deserialize macaroon."
9       else do
10        v <- verifierCreatex pc pc l
11        satisfyExactx pc v "op: update"
12        satisfyGeneralx pc v (checkTimeAfter pc l)
13        (res, _) <- verifyx pc v mac cKey []
14        if res then
15           assume ((p*←) ≽ (carol*←)) $ do
16             msg <- hGetLinex pc from_p
17             writeIFCRefx pc message msg
18        else
19          error "Could not verify macaroon."
20   where pc = carol
21         l  = carol
22         inputMac :: IFC IO Carol Carol (Macaroon, ReturnCode)
23         inputMac = assume ((p*←) ≽ (carol*←)) $ do
24                      mac <- hGetLinex pc from_p
25                      return $ deserialize . pack $ mac
```

Figure 5.19: Update message. Carol verifies the macaroon before permitting information with the client's integrity to flow to the label of the message.

roon is for Alice to present to Carol. It would be insecure for Alice to present a more restrictively labeled macaroon to Carol. However, because the caveat added by Bob on line 8 depends on `birthday` (accessed on line 21), this function may be insecure.

Suppose Alice permits Bob to know her birthday, but not Carol. Then `birthday` would represent this in Flame with a labeled value.

```
birthday :: Lbl (I Alice ∧ C (Alice ∨ Bob)) Day
birthday = label $ fromGregorian 2016 8 20
```

This label causes the compiler to reject `bobOutputMacaroon` since it cannot satisfy the constraint that `C (Alice ∨ Bob ∨ Carol) ≽ C (Alice ∨ Bob)`. If `birthday` is

```
1  bobOutputMacaroon :: IFCHandle (C (Bob ∨ Carol) ∧ I Carol)
2                    -> IFCHandle (C (Alice ∨ Bob ∨ Carol) ∧ I Bob)
3                    -> IFC IO (C (Alice ∨ Bob ∨ Carol) ∧ I Bob) SU ()
4  bobOutputMacaroon fromCarol toAlice =
5    do day <- getBirthday
6       (mac, err) <- inputMac
7       let (mac1, MacaroonSuccess) =
8             addFirstPartyCaveat mac (after day)
9           (mac2, MacaroonSuccess) =
10            addFirstPartyCaveat mac1 "op: read"
11          (serialized, MacaroonSuccess) =
12            serialize mac2 MacaroonV1 in
13          hPutStrLnx pc toAlice $ unpack serialized
14   where after day = pack $ "time >= "
15                           ++ formatTime
16                                defaultTimeLocale "%Y-%m-%d" day
17         pc = ((alice *∨ bob *∨ carol)*→) *∧ (bob*←)
18         getBirthday :: IFC IO (C (Alice ∨ Bob ∨ Carol) ∧ I Bob)
19                               (C (Alice ∨ Bob ∨ Carol) ∧ I Bob) Day
20         getBirthday = assume ((alice*←) ≽ (bob*←)) $
21                         liftx pc $ relabel birthday
22         inputMac :: IFC IO (C (Alice ∨ Bob ∨ Carol) ∧ I Bob)
23                            (C (Alice ∨ Bob ∨ Carol) ∧ I Bob)
24                            (Macaroon, ReturnCode)
25         inputMac = assume ((carol*←) ≽ (bob*←)) $
26                    assume ((*∇) (alice *∨ carol) ≽ (*∇) bob) $
27                    assume ((alice *∨ carol) ≽ bob) $ do
28                     mac <- hGetLinex bob fromCarol
29                     return $ deserialize . pack $ mac
```

Figure 5.20: A macaroon with caveats for Alice. Bob creates a macaroon with caveats for Alice. Because of the caveat added on line 8, this function is only secure if Alice permits Carol to know her birthday.

instead labeled

```
birthday :: Lbl (I Alice ∧ C (Alice ∨ Bob ∨ Carol)) Day
birthday = label $ fromGregorian 2016 8 20
```

then the compiler accepts the implementation of `bobOutputMacaroon` since all information flow constraints are satisfied.

CHAPTER 6

RELATED WORK

Flow-limited authorization builds upon work on authorization as well as work in information flow control—previously somewhat disjoint areas of security research. In this chapter, we highlight some of the most relevant research from these areas.

The connection between delegation and information flow policy downgrades, here called the delegation loophole, is identified in [39] and further developed in [83]. These papers also discuss secret trust relationships, and thus have similar threat models to FLAM. We are not aware of previous work addressing poaching attacks.

Broberg *et al.* [17] identify classes of flows which specific information flow models may consider secure or insecure. Delegation loopholes are an example of a *time-transitive flow* in their terminology. FLAM considers these flows insecure since they permit attackers to influence how information is relabeled. FLAM also considers poaching attacks to be unsafe since attackers may obtain information not directly released to them, which undermines the effectiveness of revocation. These flows are not completely characterized by the classes presented in [17], but share some characteristics with the *direct-release* class of flows.

FLAM's bounded derivation rules place information flow constraints on which delegations may be used to derive judgments. This differs from previous approaches (for example, Rx roles [83] and Flume capability groups [48]), which give a single information flow bound for all trust relationships of a principal. As recognized by Bandhakavi *et al.* [9], a single bound is too restrictive since it must also protect delegations made by other principals. So, when the bound of principal p is more restrictive than the bound of principal q, either q cannot delegate to p or p's bound must be downgraded (as in [83]), even though p might not trust q. RTI [9], like FLAM, overcomes these restrictions by tracking information flow at the level of delegations and ignoring relationships that ex-

ceed information flow bounds. However, since relabeling is not robust in RTI, it remains vulnerable to the delegation loophole and poaching attacks. FLAM's flows-to relation is more consistent with decentralized information flow control principles: the authorization of a flow depends only on those principals who speak for the policies in question.

Label algebras [61] abstract the structure and semantics of the security policies of several DIFC systems. It might appear that a clever encoding of FLAM contexts (specifically, $pc; \ell$) as label algebra *authorities* might serve to represent FLAM as a label algebra. However, such an encoding would be too abstract to represent conditions such as robust authority or even robust downgrading, so delegation loopholes and poaching attacks cannot be addressed within this framework. For instance, the noninterference lemma given in [61] for an example language admits non-robust declassification, even without changes to the trust configuration.

Many models and mechanisms have been suggested for expressive, decentralized authorization and trust management [4, 12, 13, 30, 31, 37, 50, 51, 78]. Few consider the information security of the authorization policies or the authorization process. For instance, Birgisson *et al.* [13] note that, under certain conditions, an attacker could use malicious credentials to probe for private information such as group membership. Such an attack is possible in many frameworks.

Some prior approaches have sought to reason about the information security of authorization mechanisms. Becker [11] discusses *probing attacks* that leak confidential information to an attacker. Garg and Pfenning [34] present a logic that ensures assertions made by untrusted principals cannot influence the truth of statements made by other principals. Bryans *et al.* [18] compare noninterference and opacity as security conditions for confidential policies.

FLAM ensures queries of the trust configuration satisfy robust authorization, so probing attacks cannot reveal confidential information. Opacity is a *possibilistic* notion

of security, meaning that an authorization decision may depend on secret information, provided that the same result could derive from public information. Possibilistic security conditions are often inadequate in settings with attackers that have (or can acquire) additional knowledge, perhaps through additional queries.

Some languages and systems for authorization or access control have combined aspects of information security and authorization (for example, [9, 39, 59, 60, 83, 91]) in dynamic settings. However, all of these are susceptible to the security vulnerabilities discussed in Section 2 that arise from the interaction of information flow and authorization.

DCC [2, 3] has been used to model both authorization and information flow, but not simultaneously. DCC programs are type-checked with respect to a static security lattice, whereas FLAC programs can introduce new trust relationships during evaluation, enabling more general applications.

Boudol [14] defines terms that enable or disable flows for a lexical scope, similar to `assume` terms, but does not restrict their usage. Rx [83] and RTI [9] use labeled roles to represent information flow policies. The integrity of a role restricts who may change policies. However, information flow in these languages is not robust [64]: attackers may indirectly affect how flows change when authorized principals modify policies.

Some authorization systems use access control policies to protect sensitive credentials. In trust negotiation [92, 93, 100], principals iteratively exchange credentials protected by access control policies, withholding sensitive credentials until sufficient trust has been established. Minami and Kotz [59, 60] encrypt authorization proofs based on access control policies to protect the confidentiality and integrity of authorization results, though they ignore side-channels. Because access control policies are not compositional, they are insufficient for controlling the propagation of sensitive credentials: the rules for disclosure may vary arbitrarily between principals. FLAM unifies principals

and information flow control policies, which are inherently compositional, and enforces end-to-end security of trust relationships.

Some type systems proposed for information flow control encode authorized policy downgrades directly in data types (for example, [16, 66]) or with respect to privileges granted to code (e.g. [80]). This removes some of the need for an underlying authorization mechanism, permitting developers to model trust relations using the type system or structure of the program. Such type systems are in a sense too low-level to be directly vulnerable to delegation loopholes or poaching attacks, but the authorization mechanisms they encode may still be vulnerable. FLAM can provide guidance for a way to obtain robust authorization in these systems.

Previous work has studied information flow control with higher-order functions and side effects. In the SLam calculus [38], implicit flows due to side effects are controlled via *indirect reader* annotations on types. Zdancewic and Myers [97] and Flow Caml [70] control implicit flows via *pc* annotations on function types. FLAC also controls side effects via a *pc* annotation, but here the side effects are changes in trust relationships that define which flows are permitted. Tse and Zdancewic [87] also extend DCC with a program-counter label but for a different purpose: their *pc* tracks information about the protection context, permitting more terms to be typed.

DKAL* [42] is an executable specification language for authorization protocols, simplifying analysis of protocol implementations. FLAC may be used as a specification language, but FLAC offers stronger guarantees regarding the information security of specified protocols. Errors in DKAL* specifications could lead to vulnerabilities. For instance, DKAL* provides no intrinsic guarantees about confidentiality, which could lead to authorization side channels or probing attacks. FLAC helps programmers verify that their specifications are consistent with their assumptions about trust relationships between principals, and the information flow contexts their specifications will be used

within. Furthermore, the FLAC type system ensures errors can't cause information to be released or corrupted unless they exist in high-integrity contexts and explicitly relabel the information.

The Jif programming language [62, 65] supports dynamically computed labels through a simple dependent type system. Jif also supports dynamically changing trust relationships through operations on principal objects [24]. Because the signatures of principal operations (for example, to add a new trust relationship) are missing the constraints imposed by FLAC, authorization can be used as a covert channel. FLAC shows how to close these channels in languages like Jif.

Dependently-typed languages are often expressive enough to encode authorization policies, information flow policies, or both. The F* [82] type system is capable of enforcing information flow and authorization policies. Typing rules like those in FLAC could probably be encoded within its type system, but so could incorrect, insecure rules. Thus, FLAC contributes a model for encodings that enforce strong information security.

Aura [43] embeds a DCC-based proof language and type system in a dependently-typed general-purpose functional language. As in DCC, Aura programs may derive new authorization proofs using existing proof terms and a monadic bind operator. However, Aura does not track information flow. In Aura, Alice can sign an assertion P about program state using the `say` operator, producing a value with type `Alice says` P. A malicious principal may be able to influence Alice's decision to sign an assertion or the contents of that assertion. For this reason, Aura is ill-suited for reasoning about the end-to-end information security properties of dynamic authorization mechanisms.

Enforcing information flow control with Haskell's type classes has been explored in previous work. Li and Zdancewic [52] first proposed using arrows [41] to enforce information flow in Haskell. Subsequent work has mostly focused on monad-based mechanisms. Crary *et al.* [25] use a monad parameterized by a result label as well as an effect

label, similar to `IFC`'s `l` and `pc` parameters, respectively. Devriese and Piessens [26] use an adaptation of Kmett's *parameterized monads* [45] to enforce information flow policies in Haskell. In particular, they decouple the definitions of bind, `(>>=)` and sequence, `(>>)`, which is important for precise enforcement of information flow. For example, the expression `m >>= f` passes the value of `m` to the function `f`, whereas `m >> f` discards the value, processing only the side effects. Thus, the label on the result of `f` must be at least as restrictive as `m` for `(>>=)`, but not for `(>>)`.

SecLib [74] uses a monad to enforce simple information flow policies. Downgrading is constrained by restricting access to constructors that act a capabilities for declassification or endorsement.

LIO [81] is a Haskell library that enforces information flow control dynamically using a monad that maintains a floating label at run time that is analogous to Flame's static *pc* label. Using sensitive data raises this label, and side-effects are limited to ensure information security is preserved, similar to the IO API provided by Flame. LIO is limited to IO effects, but LMonad [69] extends LIO to constrain information flow on arbitrary monads. HLIO [19] enforces both static and dynamic information flow control, and uses many of the same techniques as Flame for dependent programming in Haskell.

The formal results presented in LIO [81] and HLIO [19] do not permit policies to be downgraded. LIO implementations mediate downgrading with capabilities that permit new flows to be enabled. These systems do not offer a semantic security condition in the presence of downgrading capabilities. Waye *et al.* [89] present approaches to controlling downgrading in DCLabels, a label model often used in LIO. One of these approaches, *robust privileges*, is conjectured to enforce a property analogous to robust declassification and qualified robustness in the DLM. Flame controls downgrading using flow-limited authorization, modeled formally in FLAC, which enforces noninterference and robust declassification.

CHAPTER 7

CONCLUSION

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

Flow-limited authorization is a simple, coherent, and powerful way to address a set of fundamental, interconnected security issues. The Flow-Limited Authorization Model, or FLAM, unifies principals with information flow policies through a novel principal algebra. It supports integrated reasoning about both authorization and information flow control so that delegations are trusted only when appropriate and kept secret when necessary; further, authorization side channels are explicitly controlled. A key insight is that relabeling information flow policies is really a downgrading operation that can be made secure by preventing untrusted principals from influencing relabeling decisions.

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

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

The Flow-Limited Authorization Calculus, or FLAC, builds upon this work, leveraging FLAM as the theoretical framework for an authorization logic and secure program-

ming model. Existing security models do not account fully for the interactions between authorization and information flow. The result is that both the implementations and the uses of authorization mechanisms can lead to insecure information flows that violate confidentiality or integrity. The security of information flow mechanisms can also be compromised by dynamic changes in trust.

FLAC integrates these two security paradigms, controlling the interactions between dynamic authorization and secure information flow. FLAC offers strong guarantees and can serve as the foundation for building software that implements and uses authorization securely. Further, FLAC can be used to reason compositionally about secure authorization and secure information flow, guiding the design and implementation of future security mechanisms. It also enables new DIFC analogues for existing security mechanisms like role-based access control, bearer credentials, and commitment schemes.

We have instantiated the FLAC model for secure programming in Haskell. Flame is a library that enforces flow-limited authorization for Haskell programs by modeling information flow and authorization as monadic effects. We use Flame to integrate diverse existing authorization mechanisms into the FLAC model, from password checkers to bearer credential schemes like Macaroons. Our experience indicates that Haskell programmers can secure their programs with Flame without significantly deviating from their familiar design patterns.

APPENDIX A

FLAM APPENDIX

A.1 FLAM acts-for proof search algorithm

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

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

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

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

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

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

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

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

    progressCondition ← progressCondition ∨ ruleConditions

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

```
49
50    return new ProofSearchResult(type ← PRUNED, data ← progressCondition)
```

A.2 Normalization algorithm

We present the normalization algorithm as a set of rewriting rules that show how to combine normal-form principals to obtain another normal-form principal.

$$\mathrm{norm}^{J^{\rightarrow}}(*, \rightarrow) = J^{\rightarrow}$$

$$\mathrm{norm}^{J^{\rightarrow}}(*, \leftarrow) = \bot$$

$$\mathrm{norm}^{J^{\leftarrow}}(*, \rightarrow) = \bot$$

$$\mathrm{norm}^{J^{\leftarrow}}(*, \leftarrow) = J^{\leftarrow}$$

$$\mathrm{norm}^{J_1{}^{\rightarrow} \wedge J_2{}^{\leftarrow}}(*, \rightarrow) = J_1{}^{\rightarrow}$$

$$\mathrm{norm}^{J_1{}^{\rightarrow} \wedge J_2{}^{\leftarrow}}(*, \leftarrow) = J_2{}^{\leftarrow}$$

$$\mathrm{norm}^{J}(*, \rightarrow) = J^{\rightarrow}$$

$$\mathrm{norm}^{J}(*, \leftarrow) = J^{\leftarrow}$$

$$\mathrm{norm}^{:}(P, J^{\rightarrow}) = \mathrm{norm}^{\mathrm{norm}^{:}(P,J)}(*, \rightarrow)$$

$$\mathrm{norm}^{:}(P, J^{\leftarrow}) = \mathrm{norm}^{\mathrm{norm}^{:}(P,J)}(*, \leftarrow)$$

$$\mathrm{norm}^{:}(J^{\rightarrow}, k) = J^{\rightarrow} : k$$

$$\mathrm{norm}^{:}(J^{\leftarrow}, k) = J^{\leftarrow} : k$$

$$\mathrm{norm}^{:}(J^{\rightarrow}, T : O) = (J^{\rightarrow}) : (T : O)$$

$$\mathrm{norm}^{:}(J^{\leftarrow}, T : O) = (J^{\leftarrow}) : (T : O)$$

$$\mathrm{norm}^{:}(J_1{}^{\rightarrow} \wedge J_2{}^{\leftarrow}, P) = \mathrm{norm}^{\wedge}(\mathrm{norm}^{:}(J_1{}^{\rightarrow}, P), \mathrm{norm}^{:}(J_2{}^{\leftarrow}, P))$$

$$\mathrm{norm}^{:}(J_1 \wedge J_2, P) = \mathrm{norm}^{\wedge}(\mathrm{norm}^{:}(J_1, P), \mathrm{norm}^{:}(J_2, P))$$

$$\mathrm{norm}^{:}(P, J_1{}^{\rightarrow} \wedge J_2{}^{\leftarrow}) = \mathrm{norm}^{\wedge}(\mathrm{norm}^{:}(P, J_1{}^{\rightarrow}), \mathrm{norm}^{:}(P, J_2{}^{\leftarrow}))$$

$$\mathrm{norm}^{:}(P, J_1 \wedge J_2) = \mathrm{norm}^{\wedge}(\mathrm{norm}^{:}(P, J_1), \mathrm{norm}^{:}(P, J_2))$$

$$\mathrm{norm}^{:}(M_1 \vee M_2, P) = \mathrm{norm}^{\vee}(\mathrm{norm}^{:}(M_1, P), \mathrm{norm}^{:}(M_2, P))$$

$$\mathrm{norm}^{:}(P, M_1 \vee M_2) = \mathrm{norm}^{\vee}(\mathrm{norm}^{:}(P, M_1), \mathrm{norm}^{:}(P, M_2))$$

$$\mathrm{norm}^{:}(T : O, k) = (T : O) : k$$

$$\mathrm{norm}^{:}(T_1 : O_1, T_2 : O_2) = (T_1 : O_1) : (T_2 : O_2)$$

$$\mathrm{norm}^{:}(k, T : O) = k : (T : O)$$

$$\mathrm{norm}^{:}(k_1, k_2) = k_1 : k_2$$

$$
\begin{aligned}
\texttt{norm}^{\wedge}(J^{\rightarrow}, k) &= \texttt{norm}^{\wedge}(k, J^{\rightarrow}) = (J \wedge k)^{\rightarrow} \wedge k^{\leftarrow} \\
\texttt{norm}^{\wedge}(J_1{}^{\rightarrow}, J_2{}^{\rightarrow}) &= (J_1 \wedge J_2)^{\rightarrow} \\
\texttt{norm}^{\wedge}(J^{\rightarrow}, J^{\leftarrow}) &= \texttt{norm}^{\wedge}(J^{\leftarrow}, J^{\rightarrow}) = J \\
\texttt{norm}^{\wedge}(J_1{}^{\rightarrow}, J_2{}^{\leftarrow}) &= \texttt{norm}^{\wedge}(J_2{}^{\leftarrow}, J_1{}^{\rightarrow}) = J_1{}^{\rightarrow} \wedge J_2{}^{\leftarrow} \\
\texttt{norm}^{\wedge}(J^{\rightarrow}, T : O) &= \texttt{norm}^{\wedge}(T : O, J^{\rightarrow}) \\
&= (J \wedge (T : O))^{\rightarrow} \wedge (T : O)^{\leftarrow} \\
\texttt{norm}^{\wedge}(J^{\rightarrow}, J_1{}^{\rightarrow} \wedge J_2{}^{\leftarrow}) &= \texttt{norm}^{\wedge}(J_1{}^{\rightarrow} \wedge J_2{}^{\leftarrow}, J^{\rightarrow}) \\
&= (J \wedge J_1)^{\rightarrow} \wedge J_2{}^{\leftarrow} \\
\texttt{norm}^{\wedge}(J_1{}^{\rightarrow}, J_2) &= \texttt{norm}^{\wedge}(J_2, J_1{}^{\rightarrow}) = (J_1 \wedge J_2)^{\rightarrow} \wedge J_2{}^{\leftarrow} \\
\texttt{norm}^{\wedge}(J^{\rightarrow}, M) &= \texttt{norm}^{\wedge}(M, J^{\rightarrow}) \\
&= (J \wedge M)^{\rightarrow} \wedge M^{\leftarrow} \\
\texttt{norm}^{\wedge}(J^{\leftarrow}, k) &= \texttt{norm}^{\wedge}(k, J^{\leftarrow}) = k^{\rightarrow} \wedge (J \wedge k)^{\leftarrow} \\
\texttt{norm}^{\wedge}(J_1{}^{\leftarrow}, J_2{}^{\leftarrow}) &= (J_1 \wedge J_2)^{\leftarrow} \\
\texttt{norm}^{\wedge}(J^{\leftarrow}, T : O) &= \texttt{norm}^{\wedge}(T : O, J^{\leftarrow}) \\
&= (T : O)^{\rightarrow} \wedge (J \wedge (T : O))^{\leftarrow} \\
\texttt{norm}^{\wedge}(J^{\leftarrow}, J_1{}^{\rightarrow} \wedge J_2{}^{\leftarrow}) &= \texttt{norm}^{\wedge}(J_1{}^{\rightarrow} \wedge J_2{}^{\leftarrow}, J^{\leftarrow}) \\
&= J_1{}^{\rightarrow} \wedge (J \wedge J_2)^{\leftarrow} \\
\texttt{norm}^{\wedge}(J_1{}^{\leftarrow}, J_2) &= \texttt{norm}^{\wedge}(J_2, J_1{}^{\leftarrow}) = J_2{}^{\rightarrow} \wedge (J_1 \wedge J_2)^{\leftarrow} \\
\texttt{norm}^{\wedge}(J^{\leftarrow}, M) &= \texttt{norm}^{\wedge}(M, J^{\leftarrow}) \\
&= M^{\rightarrow} \wedge (J \wedge M)^{\leftarrow} \\
\texttt{norm}^{\wedge}(T : O, k) &= \texttt{norm}^{\wedge}(k, T : O) = (T : O) \wedge k \\
\texttt{norm}^{\wedge}(T_1 : O_1, T_2 : O_2) &= (T_1 : O_1) \wedge (T_2 : O_2) \\
\texttt{norm}^{\wedge}(J_1{}^{\rightarrow} \wedge J_2{}^{\rightarrow}, k) &= \texttt{norm}^{\wedge}(k, J_1{}^{\rightarrow} \wedge J_2{}^{\rightarrow}) \\
&= (J_1 \wedge k)^{\rightarrow} \wedge (J_2 \wedge k)^{\leftarrow} \\
\texttt{norm}^{\wedge}(J, k) &= \texttt{norm}^{\wedge}(k, J) = J \wedge k \\
\texttt{norm}^{\wedge}(J_1{}^{\rightarrow} \wedge J_2{}^{\rightarrow}, T : O) &= \texttt{norm}^{\wedge}(T : O, J_1{}^{\rightarrow} \wedge J_2{}^{\rightarrow}) \\
&= (J_1 \wedge (T : O))^{\rightarrow} \wedge (J_2 \wedge (T : O))^{\leftarrow} \\
\texttt{norm}^{\wedge}(J, T : O) &= \texttt{norm}^{\wedge}(T : O, J) = J \wedge (T : O)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{norm}^{\wedge}(J_1{}^{\rightarrow} \wedge J_2{}^{\rightarrow}, M) &= \texttt{norm}^{\wedge}(M, J_1{}^{\rightarrow} \wedge J_2{}^{\rightarrow}) \\
&= (J_1 \wedge M)^{\rightarrow} \wedge (J_2 \wedge M)^{\leftarrow} \\
\texttt{norm}^{\wedge}(J, M) &= \texttt{norm}^{\wedge}(M, J) = J \wedge M \\
\texttt{norm}^{\wedge}(M, k) &= \texttt{norm}^{\wedge}(k, M) = M \wedge k \\
\texttt{norm}^{\wedge}(M, T : O) &= \texttt{norm}^{\wedge}(T : O, M) = M \wedge (T : O) \\
\texttt{norm}^{\wedge}(M_1, M_2) &= M_1 \wedge M_2 \\
\texttt{norm}^{\vee}(J^{\rightarrow}, k) &= \texttt{norm}^{\vee}(k, J^{\rightarrow}) \\
&= (\texttt{norm}^{\vee}(J, k))^{\rightarrow} \\
\texttt{norm}^{\vee}(J_1{}^{\rightarrow}, J_2{}^{\rightarrow}) &= (\texttt{norm}^{\vee}(J_1, J_2))^{\rightarrow} \\
\texttt{norm}^{\vee}(J_1{}^{\rightarrow}, J_2{}^{\leftarrow}) &= \texttt{norm}^{\vee}(J_2{}^{\leftarrow}, J_1{}^{\rightarrow}) = \bot \\
\texttt{norm}^{\vee}(J^{\rightarrow}, T : O) &= \texttt{norm}^{\vee}(T : O, J^{\rightarrow}) \\
&= (\texttt{norm}^{\vee}(J, T : O))^{\rightarrow} \\
\texttt{norm}^{\vee}(J_1{}^{\rightarrow}, J_2{}^{\rightarrow} \wedge J_3{}^{\leftarrow}) &= \texttt{norm}^{\vee}(J_2{}^{\rightarrow} \wedge J_3{}^{\leftarrow}, J_1{}^{\rightarrow}) \\
&= (\texttt{norm}^{\vee}(J_1, J_2))^{\rightarrow} \\
\texttt{norm}^{\vee}(J_1{}^{\rightarrow}, J_2) &= \texttt{norm}^{\vee}(J_2, J_1{}^{\rightarrow}) = (\texttt{norm}^{\vee}(J_1, J_2))^{\rightarrow} \\
\texttt{norm}^{\vee}(J^{\rightarrow}, M) &= \texttt{norm}^{\vee}(M, J^{\rightarrow}) = (\texttt{norm}^{\vee}(J, M))^{\rightarrow} \\
\texttt{norm}^{\vee}(J^{\leftarrow}, k) &= \texttt{norm}^{\vee}(k, J^{\leftarrow}) \\
&= (\texttt{norm}^{\vee}(J, k))^{\leftarrow} \\
\texttt{norm}^{\vee}(J_1{}^{\leftarrow}, J_2{}^{\leftarrow}) &= (\texttt{norm}^{\vee}(J_1, J_2))^{\leftarrow} \\
\texttt{norm}^{\vee}(J^{\leftarrow}, T : O) &= \texttt{norm}^{\vee}(T : O, J^{\leftarrow}) \\
&= (\texttt{norm}^{\vee}(J, T : O))^{\leftarrow} \\
\texttt{norm}^{\vee}(J_1{}^{\leftarrow}, J_2{}^{\rightarrow} \wedge J_3{}^{\leftarrow}) &= \texttt{norm}^{\vee}(J_2{}^{\rightarrow} \wedge J_3{}^{\leftarrow}, J_1{}^{\leftarrow}) \\
&= (\texttt{norm}^{\vee}(J_1, J_3))^{\leftarrow} \\
\texttt{norm}^{\vee}(J_1{}^{\leftarrow}, J_2) &= \texttt{norm}^{\vee}(J_2, J_1{}^{\leftarrow}) = (\texttt{norm}^{\vee}(J_1, J_2))^{\leftarrow} \\
\texttt{norm}^{\vee}(J^{\leftarrow}, M) &= \texttt{norm}^{\vee}(M, J^{\leftarrow}) = (\texttt{norm}^{\vee}(J, M))^{\leftarrow} \\
\texttt{norm}^{\vee}(T : O, k) &= \texttt{norm}^{\vee}(k, T : O) = (T : O) \vee k \\
\texttt{norm}^{\vee}(T_1 : O_1, T_2 : O_2) &= (T_1 : O_1) \vee (T_2 : O_2) \\
\texttt{norm}^{\vee}(J_1{}^{\rightarrow} \wedge J_2{}^{\rightarrow}, k) &= \texttt{norm}^{\vee}(k, J_1{}^{\rightarrow} \wedge J_2{}^{\rightarrow}) \\
&= (\texttt{norm}^{\vee}(J_1, k))^{\rightarrow} \wedge (\texttt{norm}^{\vee}(J_2, k))^{\leftarrow}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{norm}^{\vee}(J_1 \wedge J_2, k) &= \mathtt{norm}^{\vee}(k, J_1 \wedge J_2) \\
&= \mathtt{norm}^{\vee}(J_1, k) \wedge \mathtt{norm}^{\vee}(J_2, k) \\
\mathtt{norm}^{\vee}({J_1}^{\rightarrow} \wedge {J_2}^{\rightarrow}, T : O) &= \mathtt{norm}^{\vee}(T : O, {J_1}^{\rightarrow} \wedge {J_2}^{\rightarrow}) \\
&= (\mathtt{norm}^{\vee}(J_1, T : O))^{\rightarrow} \wedge (\mathtt{norm}^{\vee}(J_2, T : O))^{\leftarrow} \\
\mathtt{norm}^{\vee}(J_1 \wedge J_2, T : O) &= \mathtt{norm}^{\vee}(T : O, J_1 \wedge J_2) \\
&= \mathtt{norm}^{\vee}(J_1, T : O) \wedge \mathtt{norm}^{\vee}(J_2, T : O) \\
\mathtt{norm}^{\vee}({J_1}^{\rightarrow} \wedge {J_2}^{\rightarrow}, J) &= \mathtt{norm}^{\vee}(J, {J_1}^{\rightarrow} \wedge {J_2}^{\rightarrow}) \\
&= (\mathtt{norm}^{\vee}(J_1, J))^{\rightarrow} \wedge (\mathtt{norm}^{\vee}(J_2, J))^{\leftarrow} \\
\mathtt{norm}^{\vee}(J_1 \wedge J_2, J) &= \mathtt{norm}^{\vee}(J, J_1 \wedge J_2) \\
&= \mathtt{norm}^{\vee}(J_1, J) \wedge \mathtt{norm}^{\vee}(J_2, J) \\
\mathtt{norm}^{\vee}(T : O, M) &= \mathtt{norm}^{\vee}(M, T : O) = (T : O) \vee M \\
\mathtt{norm}^{\vee}(T_1 : O_1, T_2 : O_2) &= (T_1 : O_1) \vee (T_2 : O_2) \\
\mathtt{norm}^{\vee}({J_1}^{\rightarrow} \wedge {J_2}^{\rightarrow}, M) &= \mathtt{norm}^{\vee}(M, {J_1}^{\rightarrow} \wedge {J_2}^{\rightarrow}) \\
&= (\mathtt{norm}^{\vee}(J_1, M))^{\rightarrow} \wedge (\mathtt{norm}^{\vee}(J_2, M))^{\leftarrow} \\
\mathtt{norm}^{\vee}(J_1 \wedge J_2, M) &= \mathtt{norm}^{\vee}(M, J_1 \wedge J_2) \\
&= \mathtt{norm}^{\vee}(J_1, M) \wedge \mathtt{norm}^{\vee}(J_2, M) \\
\mathtt{norm}^{\vee}(M, k) &= \mathtt{norm}^{\vee}(k, M) = M \vee k \\
\mathtt{norm}^{\vee}(M, T : O) &= \mathtt{norm}^{\vee}(T : O, M) = M \vee (T : O) \\
\mathtt{norm}^{\vee}(M_1, M_2) &= M_1 \vee M_2
\end{aligned}
$$

APPENDIX B

FLAC APPENDIX

B.1 Proofs of FLAC noninterference and robustness

Lemma 4 (Soundness)**.** *If* $e \longrightarrow^* e'$ *then* $\lfloor e \rfloor_1 \longrightarrow \lfloor e' \rfloor_1$ *and* $\lfloor e \rfloor_2 \longrightarrow \lfloor e' \rfloor_2$.

Proof. By inspection of the rules in Figure 4.5 and Figure B.1. □

Lemma 5 (Completeness)**.** *If* $\lfloor e \rfloor_1 \longrightarrow^* v_1$ *and* $\lfloor e \rfloor_2 \longrightarrow^* v_2$*, then there exists some* v *such that* $e \longrightarrow^* v$.

Proof. Assume $\lfloor e \rfloor_1 \longrightarrow^* v_1$ and $\lfloor e \rfloor_2 \longrightarrow^* v_2$. The extended set of rules in Figure B.1 always move brackets out of subterms, and therefore can only be applied a finite number of times. Therefore, by Lemma 4, if e diverges, either $\lfloor e \rfloor_1$ or $\lfloor e \rfloor_2$ diverge; this contradicts our assumption.

It remains to be shown that if the evaluation of e gets stuck, either $\lfloor e \rfloor_1$ or $\lfloor e \rfloor_2$ gets stuck. This is easily proven by induction on the structure of e. Therefore, since we assumed $\lfloor e \rfloor_i \longrightarrow^* v_i$, then e must terminate. Thus, there exists some v such that $e \longrightarrow^* v$. □

Lemma 6 (Substitution)**.** *If* $\Pi; \Gamma, x : s'; pc \vdash e : s$ *and* $\Pi; \Gamma; pc \vdash v : s'$ *then* $\Pi; \Gamma; pc \vdash e[x \mapsto v] : s$.

Proof. By induction on the derivation of $\Pi; \Gamma, x : s'; pc \vdash e : s$. □

Lemma 7 (Type substitution)**.** *If* $\Pi; \Gamma, X; pc \vdash e : s$ *then* $\Pi; \Gamma; pc \vdash e[X \mapsto s'] : s[X \mapsto s']$.

Proof. By induction on the derivation of $\Pi; \Gamma, X; pc \vdash e : s$. □

Lemma 8 (Projection)**.** *If* $\Pi; \Gamma; pc \vdash e : s$ *then* $\Pi; \Gamma; pc \vdash \lfloor e \rfloor_i : s$

Syntax extensions

$$\begin{array}{rcl} v & ::= & \ldots \mid (v \mid v) \\ e & ::= & \ldots \mid (e \mid e) \end{array}$$

Typing extensions

[BRACKET]
$$\frac{\Pi; \Gamma; \mathit{pc}' \vdash e_1 : s \qquad \Pi; \Gamma; \mathit{pc}' \vdash e_2 : s \qquad \Pi; \mathit{pc}; \mathit{pc} \Vdash (H \sqcup \mathit{pc})^\pi \sqsubseteq \mathit{pc}'^\pi \qquad \Pi; \mathit{pc} \vdash H^\pi \leq s}{\Pi; \Gamma; \mathit{pc} \vdash (e_1 \mid e_2) : s}$$

Evaluation extensions

[B-STEP]
$$\frac{e_i \longrightarrow e_i' \qquad e_j = e_j' \qquad \{i, j\} = \{1, 2\}}{(e_1 \mid e_2) \longrightarrow (e_1' \mid e_2')}$$

[B-APP] $(v_1 \mid v_2)\ v \longrightarrow (v_1\ \lfloor v \rfloor_1 \mid v_2\ \lfloor v \rfloor_2)$ [B-TAPP] $(v_1 \mid v_2)\ s \longrightarrow (v_1\ s \mid v_2\ s)$

[B-UNPAIR] $\mathtt{proj}_i\ (v_1 \mid v_2) \longrightarrow (\mathtt{proj}_i\ v_1 \mid \mathtt{proj}_i\ v_2)$

[B-CASE]
$$\begin{array}{l} \mathtt{case}\ (v_1 \mid v_2)\ \mathtt{of}\ \mathtt{inj}_1(x).\ e_1 \mid \mathtt{inj}_2(x).\ e_2 \longrightarrow \\ \quad (\mathtt{case}\ v_1\ \mathtt{of}\ \mathtt{inj}_1(x).\ \lfloor e_1 \rfloor_1 \mid \mathtt{inj}_2(x).\ \lfloor e_2 \rfloor_1 \\ \quad \mid \mathtt{case}\ v_2\ \mathtt{of}\ \mathtt{inj}_1(x).\ \lfloor e_1 \rfloor_2 \mid \mathtt{inj}_2(x).\ \lfloor e_2 \rfloor_2) \end{array}$$

[B-UNITM] $\eta_\ell\ (v_1 \mid v_2) \longrightarrow (\eta_\ell\ v_1 \mid \eta_\ell\ v_2)$

[B-BINDM] $\mathtt{bind}\ x = (v_1 \mid v_2)\ \mathtt{in}\ e \longrightarrow (\mathtt{bind}\ x = v_1\ \mathtt{in}\ \lfloor e \rfloor_1 \mid \mathtt{bind}\ x = v_2\ \mathtt{in}\ \lfloor e \rfloor_2)$

[B-ASSUME] $\mathtt{assume}\ (v_1 \mid v_2)\ \mathtt{in}\ e \longrightarrow (\mathtt{assume}\ v_1\ \mathtt{in}\ \lfloor e \rfloor_1 \mid \mathtt{assume}\ v_2\ \mathtt{in}\ \lfloor e \rfloor_2)$

Figure B.1: Extensions for bracketed semantics

Proof. By induction on the derivation of $\Pi; \Gamma; \mathit{pc} \vdash e : s$. □

Lemma 9 (Values)**.** *If* $\Pi; \Gamma; \mathit{pc} \vdash v : s$*, then* $\Pi; \Gamma; \mathit{pc}' \vdash v : s$ *for any* pc'.

Proof. By induction on the derivation of $\Pi; \Gamma; \mathit{pc} \vdash e : s$. □

Lemma 10 (Robust transitivity)**.** *If* $\Pi; \mathit{pc}; \ell \Vdash p \succcurlyeq q$ *and* $\Pi; \mathit{pc}; \ell \Vdash q \succcurlyeq r$*, then* $\Pi; \mathit{pc}; \ell \Vdash p \succcurlyeq r$.

Proof. This is a consequence of the FLAM's Principal Factorization Lemma, Lemma 1 in Section 3.4, and verified in Coq. □

Lemma 11 (Voices)**.** *If* $\Pi; \mathit{pc}; \ell \Vdash p \succcurlyeq q$ *then* $\Pi; \mathit{pc}; \ell \Vdash \nabla(p) \succcurlyeq \nabla(q)$.

Proof. By induction on the derivation of $\Pi; \mathit{pc}; \ell \Vdash p \succcurlyeq q$. $\mathcal{L} \vDash p \succcurlyeq q$ implies $\Pi; \mathit{pc}; \ell \Vdash \nabla(p) \succcurlyeq \nabla(q)$ (verified in Coq), and each $\langle p \succcurlyeq q \mid \ell \rangle \in \Pi$ has $\Pi; \mathit{pc}; \ell \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow})$, so $\langle p \succcurlyeq q \mid \ell \rangle \in \Pi$ implies $\Pi; \mathit{pc}; \ell \Vdash \nabla(p) \succcurlyeq \nabla(q)$. The remaining cases are trivial. □

Lemma 12 (*pc* reduction)**.** *If* $\Pi; \Gamma; \mathit{pc}' \vdash e : s$ *and* $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \sqsubseteq \mathit{pc}'$*, then* $\Pi; \Gamma; \mathit{pc} \vdash \lfloor e \rfloor_i : s$.

Proof. By induction on the derivation of $\Pi; \Gamma; \mathit{pc}' \vdash e : s$ and Lemma 10. Note that BRACKET does not preserve this property, hence the projection of e is necessary. □

Theorem 4 (Subject reduction)**.** *Suppose* $\Pi; \Gamma; \mathit{pc} \vdash e : s$ *and* $\lfloor e \rfloor_i \longrightarrow \lfloor e' \rfloor_i$*. If* $i \in \{1, 2\}$ *then assume* $\Pi; \mathit{pc}; \mathit{pc} \Vdash H \sqsubseteq \mathit{pc}$*. Then* $\Pi; \Gamma; \mathit{pc} \vdash e' : s$.

Proof.

Case (E-APP). e is $(\lambda(x : s')[\mathit{pc}'].\, e')\ v$, so by APP we have $\Pi; \Gamma; \mathit{pc} \vdash v : s'$ and $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \sqsubseteq \mathit{pc}'$ and by LAM we have $\Pi; \Gamma, x : s'; \mathit{pc}' \vdash e' : s$. Then by Lemma 9 we have $\Pi; \Gamma; \mathit{pc}' \vdash v : s'$, and by Lemma 6 we obtain $\Pi; \Gamma; \mathit{pc}' \vdash e'[x \mapsto v] : s$.

Case (E-TAPP). e is $(\Lambda X.\, e)\ s'$ and s is $s[X \mapsto s']$, so by TAPP we have $\Pi; \Gamma; \mathit{pc} \vdash e : \forall X.\, s$. Then by TLAM, we have $\Pi; \Gamma, X; \mathit{pc} \vdash e : s$ and by Lemma 7 we obtain $\Pi; \Gamma; \mathit{pc} \vdash e[X \mapsto s] : s[X \mapsto s']$.

Case (E-CASE). e is

$$(\texttt{case}\ (\texttt{inj}_1\ v)\ \texttt{of}\ \texttt{inj}_1(x).\, e_1 \mid \texttt{inj}_2(x).\, e_2)$$

By INJ we have $\Pi; \Gamma; \mathit{pc} \vdash v : s_1$, and CASE gives us $\Pi; \Gamma; \mathit{pc} \vdash e_1 : s$. Therefore, by Lemma 6 we have $\Pi; \Gamma; \mathit{pc} \vdash e_1[x \mapsto v] : s$.

Case (E-BINDM). e is $\texttt{bind}\ x = (\eta_\ell\ v)\ \texttt{in}\ e'$ so by BINDM we have $\Pi; \Gamma; \mathit{pc} \vdash (\eta_\ell\ v) : \ell\ \texttt{says}\ s'$ and $\Pi; \Gamma; \mathit{pc} \sqcup \ell \vdash e' : s$. Rule UNITM and Lemma 9 give us $\Pi; \Gamma; \mathit{pc} \sqcup \ell \vdash v : s'$. Therefore, by Lemma 6 we have $\Pi; \Gamma; \mathit{pc} \sqcup \ell \vdash e'[x \mapsto v] : s$.

Case (E-ASSUME). e is $\texttt{assume}\ v\ \texttt{in}\ e''$ and e' is $e''\ \texttt{where}\ v$, Let $\Pi' = \Pi, \langle p \succcurlyeq q \mid \mathit{pc}\rangle$. By ASSUME we have $\Pi; \Gamma; \mathit{pc} \vdash v : (p \succcurlyeq q)$ and $\Pi'; \Gamma; \mathit{pc} \vdash e'' : s$. Therefore, by WHERE (choosing $\mathit{pc}' = \mathit{pc}$) we have $\Pi; \Gamma; \mathit{pc} \vdash (e''\ \texttt{where}\ v) : s$.

Case (E-EVAL). e is $E[e]$. By induction, $\Pi; \Gamma; \mathit{pc} \vdash e' : s'$. Therefore, $\Pi; \Gamma; \mathit{pc} \vdash E[e'] : s$.

Case (W-*). We prove the case for W-APP here. e is $(v''\ \texttt{where}\ v)\ v'$ and e' is $v''\ v'\ \texttt{where}\ v$. By APP and WHERE we have

$$\Pi, \langle p \succcurlyeq q \mid \mathit{pc}'\rangle; \Gamma; \mathit{pc}' \vdash v'' : s' \xrightarrow{\mathit{pc}''} s$$

$$\Pi; \Gamma; \mathit{pc} \vdash v' : s' \qquad \Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \sqsubseteq \mathit{pc}'' \qquad \Pi; \Gamma; \mathit{pc} \vdash v : (p \succcurlyeq q)$$

$$\Pi; \mathit{pc}'; \mathit{pc}' \Vdash \mathit{pc}' \sqsubseteq \mathit{pc} \qquad \Pi; \mathit{pc}'; \mathit{pc}' \Vdash \mathit{pc}' \succcurlyeq \nabla(q)$$

$$\text{and } \Pi; \mathit{pc}'; \mathit{pc}' \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow})$$

From this, we obtain $\Pi, \langle p \succcurlyeq q \mid \mathit{pc}'\rangle; \Gamma; \mathit{pc}' \vdash v''v' : s$ via APP and Lemma 10. Then we get $\Pi; \Gamma; \mathit{pc} \vdash v''\ v'\ \texttt{where}\ v : s$ via WHERE. The remaining cases follow similarly to the case for W-APP, but using the relevant typing rule for the underlying term (e.g., UNPAIR, or CASE, *et cetera*) instead of APP.

Case (B-STEP). e is $(e_1 \mid e_2)$. Assume without loss of generality that $e_1 \longrightarrow e_1'$ and $e_2 = e_2'$. By BRACKET, $\Pi; \Gamma; \mathit{pc} \vdash e_1 : s$. By induction, $\Pi; \Gamma; \mathit{pc} \vdash e_1' : s$, thus BRACKET gives us $\Pi; \Gamma; \mathit{pc} \vdash (e_1' \mid e_2') : s$.

Case (B-APP). e is $(v_1 \mid v_2)\ v$. By APP we have $\Pi; \Gamma; \mathit{pc} \vdash (v_1 \mid v_2) : s' \xrightarrow{\mathit{pc}'} s$ and $\Pi; \Gamma; \mathit{pc} \vdash v : s'$, and by BRACKET, we have $\Pi; \mathit{pc} \vdash H \leq (s' \xrightarrow{\mathit{pc}'} s)$. By P-FUN, we have $\Pi; \mathit{pc} \vdash H \leq s$. By Lemma 8, we have $\Pi; \Gamma; \mathit{pc} \vdash v_i : (s' \xrightarrow{\mathit{pc}'} s)$. Therefore, by APP and BRACKET, we have $\Pi; \Gamma; \mathit{pc} \vdash (v_1\ \lfloor v \rfloor_1 \mid v_2\ \lfloor v \rfloor_2) : s$.

Case (B-TAPP). e is $(v_1 \mid v_2)\ s'$. By TAPP we have $\Pi; \Gamma; \mathit{pc} \vdash (v_1 \mid v_2) : \forall X.\, s$, and by BRACKET, we have $\Pi; \mathit{pc} \vdash H \leq (\forall X.\, s)$. By P-TFUN, we have $\Pi; \mathit{pc} \vdash H \leq s$. By Lemma 8, we have $\Pi; \Gamma; \mathit{pc} \vdash v_i : (\forall X.\, s)$. Therefore, by TAPP and BRACKET, we

have $\Pi; \Gamma; pc \vdash (v_1\ s' \mid v_2\ s') : s$.

Case (B-UNPAIR). e is $\mathtt{proj}_i\ (v_1 \mid v_2)$ and e' is $(\mathtt{proj}_i\ v_1 \mid \mathtt{proj}_i\ v_2)$. By UNPAIR, $\Pi; \Gamma; pc \vdash \mathtt{proj}_i\ (v_1 \mid v_2) : s$, and by BRACKET, we have $\Pi; pc \vdash H \leq s$. Then by Lemma 8, we have $\Pi; \Gamma; pc \vdash \mathtt{proj}_i\ v_i : s$, so UNPAIR and BRACKET give us $\Pi; \Gamma; pc \vdash (\mathtt{proj}_i\ v_1 \mid \mathtt{proj}_i\ v_2) : s$.

Case (B-CASE). e is

$$(\mathtt{case}\ (v_1 \mid v_2)\ \mathtt{of}\ \mathtt{inj}_1(x).\ e_1 \mid \mathtt{inj}_2(x).\ e_2)$$

and e' is

$$\begin{aligned}(&\mathtt{case}\ v_1\ \mathtt{of}\ \mathtt{inj}_1(x).\ \lfloor e_1 \rfloor_1 \mid \mathtt{inj}_2(x).\ \lfloor e_2 \rfloor_1 \\ &\mid \mathtt{case}\ v_2\ \mathtt{of}\ \mathtt{inj}_1(x).\ \lfloor e_1 \rfloor_2 \mid \mathtt{inj}_2(x).\ \lfloor e_2 \rfloor_2)\end{aligned}$$

By BRACKET, for some pc' we have $\Pi; \Gamma; pc' \vdash v_i : s_i$ and $\Pi; pc; pc \Vdash (H \sqcup pc)^\pi \sqsubseteq pc'^\pi$. By CASE and Lemma 8, we have $\Pi; pc \vdash pc' \leq s$, therefore Lemma 10 gives us $\Pi; pc \vdash H \leq s$. We also have $\Pi; \Gamma; pc \vdash \lfloor e_1 \rfloor_i : s$ and $\Pi; \Gamma; pc \vdash \lfloor e_2 \rfloor_i : s$ for $i \in \{1, 2\}$. Therefore, by CASE we have $\Pi; \Gamma; pc \vdash \lfloor e' \rfloor_i : s$, and by BRACKET, we have $\Pi; \Gamma; pc \vdash e' : s$.

Case (B-UNITM). s is $\ell\ \mathtt{says}\ s$, e is $\eta_\ell\ (v_1 \mid v_2)$, and e' is $(\eta_\ell\ v_1 \mid \eta_\ell\ v_2)$ By UNITM, $\Pi; \Gamma; pc \vdash (v_1 \mid v_2) : s$, and by BRACKET, we have $\Pi; pc \vdash H \leq s$. Then by P-LBL1, we have $\Pi; pc \vdash H \leq \ell\ \mathtt{says}\ s$. Therefore UNITM and BRACKET give us $\Pi; \Gamma; pc \vdash (\eta_\ell\ v_1 \mid \eta_\ell\ v_2) : \ell\ \mathtt{says}\ s$.

Case (B-BINDM). e is $\mathtt{bind}\ x = (v_1 \mid v_2)\ \mathtt{in}\ e''$, and e' is

$$(\mathtt{bind}\ x = v_1\ \mathtt{in}\ \lfloor e'' \rfloor_1 \mid \mathtt{bind}\ x = v_2\ \mathtt{in}\ \lfloor e'' \rfloor_2)$$

By BINDM and BRACKET, for some pc' we have $\Pi; \Gamma; pc' \vdash v_i : s'$ and $\Pi; pc; pc \Vdash (H \sqcup pc)^\pi \sqsubseteq pc'^\pi$. Also, by BINDM and Lemma 10, we have $\Pi; pc \vdash H \leq s$. Then,

using Lemma 8, we have $\Pi; \Gamma; \mathit{pc} \vdash \texttt{bind}\ x = v_i\ \texttt{in}\ \lfloor e'' \rfloor_i : s$, so BRACKET gives us $\Pi; \Gamma; \mathit{pc} \vdash (\texttt{bind}\ x = v_1\ \texttt{in}\ \lfloor e'' \rfloor_1 \mid \texttt{bind}\ x = v_2\ \texttt{in}\ \lfloor e'' \rfloor_2) : s$.

Case (B-ASSUME). e is $\texttt{assume}\ (\langle p_1 \succcurlyeq q_1 \rangle \mid \langle p_2 \succcurlyeq q_2 \rangle)\ \texttt{in}\ e''$, and e' is $(\texttt{assume}\ v_1\ \texttt{in}\ \lfloor e'' \rfloor_1 \mid \texttt{assume}\ v_2\ \texttt{in}\ \lfloor e'' \rfloor_2)$. By ASSUME and BRACKET, for some pc' we have $\Pi; \Gamma; \mathit{pc}' \vdash v_i : (p \succcurlyeq q)$ and $\Pi; \mathit{pc}; \mathit{pc} \Vdash (H \sqcup \mathit{pc})^\pi \sqsubseteq \mathit{pc}'^\pi$. By ASSUME and Lemma 8, we have $\Pi; \mathit{pc} \vdash \mathit{pc}' \leq s$, therefore Lemma 10 gives us $\Pi; \mathit{pc} \vdash H \leq s$. We also have $\Pi; \Gamma; \mathit{pc} \vdash v_i : (p \succcurlyeq q)$ and $\Pi, \langle p \succcurlyeq q \mid \mathit{pc} \rangle; \Gamma; \mathit{pc} \vdash \lfloor e'' \rfloor_i : s$ for $i \in \{1, 2\}$. Therefore, by ASSUME we have $\Pi; \Gamma; \mathit{pc} \vdash \texttt{assume}\ v_i\ \texttt{in}\ \lfloor e'' \rfloor_i : s$, and by BRACKET, we have $\Pi; \Gamma; \mathit{pc} \vdash e' : s$.

□

We extend the result of Lemma 1 to minimal factorizations.

Lemma 13 (Delegation Factorization)**.** *If* $\Pi; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q$ *and* (p, q_s, q_d) *is the minimal static factorization of* (p, q)*, then* $\Pi; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q_d$ *and*

$$\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \succcurlyeq \nabla(q_d)$$

Proof. A Coq-verified proof in [7] showed that there is some static factorization (p, q'_s, q'_d) such that $\Pi; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q'_d$ and $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \succcurlyeq \nabla(q'_d)$. By the definition of minimal factorization, $\mathcal{L} \vDash q'_d \succcurlyeq q_d$, so $\mathcal{L} \vDash \nabla(q'_d) \succcurlyeq \nabla(q_d)$, and by transitivity on static acts-for relationships, $\Pi; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q_d$ and $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \succcurlyeq \nabla(q_d)$. □

Lemma 3 (Delegation Invariance)**.** *Let* $\Pi; \Gamma; \mathit{pc} \vdash e : s$ *such that* $e \longrightarrow e'$ `where` v*. Then there exist* $r, t \in \mathcal{L}$ *and* $\Pi' = \Pi, \langle r^\pi \succcurlyeq t^\pi \mid \mathit{pc} \rangle$ *such that* $\Pi; \Gamma; \mathit{pc} \vdash v : (r^\pi \succcurlyeq t^\pi)$ *and* $\Pi'; \Gamma; \beta; \mathit{pc} \vdash e' : s$*. Moreover, for all principals* p *and* q *if* $\Pi; \mathit{pc}; \mathit{pc} \nVdash \mathit{pc} \succcurlyeq \nabla(q^\pi) - \nabla(p^\pi)$*, then*

$$\Pi'; \mathit{pc}; \mathit{pc} \nVdash p^\pi \succcurlyeq q^\pi.$$

Proof. Let (p, q_s, q_d) be the minimal $(\Pi; \mathit{pc})$-factorization of (p, q) (so $\Pi; \mathit{pc}; \mathit{pc} \Vdash q_d \equiv q - p$). First we note that $q_d \neq \bot$ and thus $\Pi; \mathit{pc}; \mathit{pc} \nVdash p \succcurlyeq q_d$. Now assume for contradiction that $\Pi'; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q$.

We claim that $\Pi; \mathit{pc}; \mathit{pc} \Vdash t^\pi \succcurlyeq q_d$. Assume for contradiction that this is false. By transitivity $\Pi'; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q_d$, and since $\Pi; \mathit{pc}; \mathit{pc} \nVdash p \succcurlyeq q_d$, any derivation of this must use R-ASSUME with the delegation $\langle r^\pi \succcurlyeq t^\pi \mid \mathit{pc} \rangle$. This means that $\Pi; \mathit{pc}; \mathit{pc} \Vdash q_d \equiv t^\pi \wedge q'_d$ for some q'_d where $\Pi; \mathit{pc}; \mathit{pc} \nVdash t^\pi \succcurlyeq q'_d$. Assume without loss of generality that $\Pi; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q'_d$. Otherwise the same argument would give us that $\Pi; \mathit{pc}; \mathit{pc} \Vdash q'_d \equiv t^\pi \wedge q''_d$, so $q_d = t^\pi \wedge q''_d$ so we can let $q'_d = q''_d$. Since all representations are finite and with each iteration we remove at least one term from q'_d, we can only do this finitely many times until eventually $\Pi; \mathit{pc}; \mathit{pc} \Vdash p \succcurlyeq q'_d$ (possibly because $q'_d = \bot$). However, this means that $(p, q_s \wedge q'_d, t^\pi)$ is a valid $(\Pi; \mathit{pc})$-factorization of (p, q). Since $\Pi; \mathit{pc}; \mathit{pc} \nVdash t^\pi \succcurlyeq q'_d$ by assumption, it is also the case that $\Pi; \mathit{pc}; \mathit{pc} \nVdash t^\pi \succcurlyeq t^\pi \wedge q'_d$, which contradicts the assumption that (p, q_s, q_d) is minimal and thus $\Pi; \mathit{pc}; \mathit{pc} \Vdash t^\pi \succcurlyeq q_d$.

If $\pi = \rightarrow$, then by WHERE with $\mathit{pc}' = \mathit{pc}$, $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \succcurlyeq \nabla(t^\rightarrow)$ and by Lemma 11, $\Pi; \mathit{pc}; \mathit{pc} \Vdash \nabla(t^\rightarrow) \succcurlyeq \nabla(q_d^\rightarrow)$. Thus, by transitivity, $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \succcurlyeq \nabla(q_d^\rightarrow)$ which contradicts our assumption. Similarly, if $\pi = \leftarrow$, by WHERE with $\mathit{pc}' = \mathit{pc}$, $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \succcurlyeq t^\leftarrow$, so by transitivity, $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \succcurlyeq q_d^\leftarrow$ which again contradicts our assumption. □

Theorem 2 (Noninterference)**.** *Let* $\Pi; \Gamma, x : s; \mathit{pc} \vdash e : \ell$ `says bool`*. If there exists some* H *and* π *such that*

1. $\Pi; \mathit{pc} \vdash H^\pi \leq s$
2. $\Pi; \mathit{pc}; \mathit{pc} \nVdash H^\pi \sqsubseteq \ell^\pi$
3. *(a) if* $\pi = \rightarrow$ *then* $\Pi; \mathit{pc}; \mathit{pc} \nVdash \mathit{pc} \succcurlyeq \nabla(H^\rightarrow) - H^\leftarrow$

   (b) if $\pi = \leftarrow$ *then* $\Pi; \mathit{pc}; \mathit{pc} \nVdash \mathit{pc} \succcurlyeq (\ell - H)^\leftarrow$

then for all v_1, v_2 *with* $\Pi; \Gamma; pc \vdash v_i : s$*, if* $e[x \mapsto v_i] \longrightarrow^* v_i'$*, then* $v_1' = v_2'$.

Proof. To prove this, we employ the bracketed semantics. By Lemma 9 $\Pi; \Gamma; \top^{\rightarrow} \vdash v_i : s$ and thus, for any H, $\Pi; \Gamma; pc \vdash (v_1 \mid v_2) : s$.

We now examine the term $e[x \mapsto (v_1 \mid v_2)]$. By Lemma 6, $\Pi; \Gamma; pc \vdash e[x \mapsto (v_1 \mid v_2)] : \ell$ `says bool`, and by assumption $e[x \mapsto v_i] \longrightarrow^* v_i$. Thus by Lemma 5 there is some v' such that $e[x \mapsto (v_1 \mid v_2) H^{\pi}] \longrightarrow^* v'$, and moreover, by Theorem 4 $\Pi; \Gamma; pc \vdash v' : \ell$ `says bool`.

We will show that $v_1' = v_2'$ by showing that v' does not contain a bracket term. First we note that since $\Pi; pc; pc \not\vdash H^{\pi} \sqsubseteq \ell^{\pi}$, v' cannot itself be a bracketed term. Similarly, it cannot be the case that $v' = (\eta_{\ell}\ w)$ where w is bracketed. The only other option is that $v' = w$ `where` u where w contains a bracket term. We will prove this is not the case by induction on the number of $\texttt{where}_{\pi}$ clauses.

For the base case with zero clauses, we have already shown this. Now we assume that $v' = w$ `where` u and w is not itself a $\texttt{where}_{\pi}$ clause containing a bracketed term. There are two cases to consider, depending on π.

Case ($\pi = \rightarrow$). In this case condition 2 means $\Pi; pc; pc \not\vdash \ell^{\rightarrow} \not\succcurlyeq H^{\rightarrow}$. Since $\Pi; pc; pc \not\vdash pc \not\succcurlyeq \nabla(H^{\rightarrow}) - H^{\leftarrow}$, Lemma 3 gives us that there is some Π' such that $\Pi'; \Gamma; pc \vdash u : \ell$ `says bool` and $\Pi'; pc; pc \not\vdash \ell^{\rightarrow} \not\succcurlyeq H^{\rightarrow}$. Thus w cannot itself be a bracket term (or $(\eta_{\ell}\ w')$), and by the inductive hypothesis it is not a `where` clause containing a bracket, thus proving that v' contains no bracketed terms.

Case ($\pi = \leftarrow$). In this case condition 2 means $\Pi; pc; pc \not\vdash H^{\leftarrow} \not\succcurlyeq \ell^{\leftarrow}$. Since $\Pi; pc; pc \not\vdash pc \not\succcurlyeq (\ell - H)^{\leftarrow}$, Lemma 3 and the same argument as the previous case show that w contains no bracketed terms.

Thus we have that v' contains no bracketed terms, so

$$v_1' = \lfloor v' \rfloor_1 = \lfloor v' \rfloor_2 = v_2'.$$

□

To prove FLAC enforces robust declassification, we first prove a stronger property we call noninterference under attacks. This property demonstrates that attacks by low integrity principals cannot create interfering flows of information. This result uses the following property of principal subtraction.

Lemma 14. *For all principals p, q, and r,*

$$\Pi; \mathit{pc}; \mathit{pc} \Vdash (p - q) \land (q - r) \succcurlyeq p - r.$$

Proof. Let $\overline{p}_d$ such that $\Pi; \mathit{pc}; \mathit{pc} \Vdash \overline{p}_d \equiv (p - q) \land (q - r)$. Now consider the following expression

$$\begin{aligned}
\Pi; \mathit{pc}; \mathit{pc} \Vdash [\overline{p}_d \lor p] \land (p \lor r) &\equiv p \lor (\overline{p}_d \land r) \\
&\equiv p \lor [(p - q) \land (r \land (q - r))] \\
&\succcurlyeq p \lor [(p - q) \land q] \\
&\succcurlyeq p \lor p \\
&\equiv p.
\end{aligned}$$

Thus we see that for any $(\Pi; \mathit{pc})$-factorization $(r, p \lor r, p_d)$ of (r, p), $\Pi; \mathit{pc}; \mathit{pc} \Vdash \overline{p_d} \lor p \succcurlyeq p_d$ which means $\Pi; \mathit{pc}; \mathit{pc} \Vdash \overline{p_d} \succcurlyeq p_d$. In particular, since $(p - r)$ represents the minimal such factorization, $\Pi; \mathit{pc}; \mathit{pc} \Vdash (p - q) \land (q - r) \succcurlyeq p - r$. □

Lemma 15 (Noninterference under attacks)**.** *Let $e[\vec{\bullet}]$ be a program such that*

1. $\Pi; \Gamma; \mathit{pc} \vdash e[\vec{\bullet}] : \ell$ `says bool`
2. *For all* $\langle p \succcurlyeq q \mid \mathit{pc}' \rangle \in \Pi$, $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc}' \sqsubseteq \mathit{pc}$
3. $\Pi; \mathit{pc}; \mathit{pc} \nVdash H^\pi \sqsubseteq \ell^\pi$
4. *(a) if* $\pi = \rightarrow$ *then* $\Pi; \mathit{pc}; \mathit{pc} \nVdash \mathit{pc} \succcurlyeq \nabla(H^\rightarrow) - H^\leftarrow$

   (b) if $\pi = \leftarrow$ *then* $\Pi; \mathit{pc}; \mathit{pc} \nVdash \mathit{pc} \succcurlyeq (\ell - H)^\leftarrow$

Then for all attacks $\vec{a_1}$ *and* $\vec{a_2}$ *such that* $\Pi; \Gamma; pc \vdash e[\vec{a_i}] : \ell$ `says bool`*, if* $e[\vec{a_i}] \longrightarrow^* v_i$*, then* $v_1 = v_2$.

Proof. Without loss of generality, assume $\vec{a_i}$ contains only one term, which we will refer to as a_i. Since $\vec{a_1}$ and $\vec{a_2}$ must have the same, finite number of terms n, if this were not the case we could construct a sequence of attacks $\vec{b_0}, \ldots, \vec{b_n}$ such that $\vec{b_i}$ includes the first i terms of $\vec{a_1}$ and the rest of the terms from $\vec{a_2}$. As each term must type check independently by HOLE, each of the $\vec{b_j}$s are valid attacks, each differ from the previous by at most one term, and $\vec{a_1} = \vec{b_n}$ and $\vec{a_2} = \vec{b_0}$. Thus it suffices to show that if at most one term differs, the attacks still produce the same result.

By HOLE, there is some Π', Γ', β', pc', and type τ such that

- $\Pi' \supseteq \Pi$,
- $\Gamma' \supseteq \Gamma$,
- $\Pi'; pc'; pc' \Vdash \beta' \sqsubseteq \beta$,
- $\Pi'; pc'; pc' \Vdash pc \sqsubseteq pc'$,
- $\Pi'; \Gamma'; \beta'; pc' \vdash [a_i]_{H^\pi} : \tau$,
- $\Pi'; pc' \vdash H^{\leftarrow} \leq \tau$.

Define $\mathrm{grd}(\tau)$ as the most restrictive principal protected by type τ. Specifically, the principal p such that $\varnothing; pc \vdash p \leq \tau$ and for all principals p' such that $\varnothing; pc \vdash p' \leq \tau$, we have $\mathcal{L} \vDash p' \sqsubseteq p$.

Let $\hat{H} = \mathrm{grd}(\tau)^\pi \wedge H^{\pi'}$ where $\pi' \neq \pi$ is the opposite projection. We note that $\Pi; \Gamma; pc \vdash e[a_i] : \ell$ `says bool` and $\Pi; pc; pc \Vdash \hat{H}^\pi \sqsubseteq \mathrm{grd}(\tau)^\pi$ trivially, and we claim that Condition 4 holds with $\hat{H}$ in place of H. We prove this in two cases.

Case (π = $\rightarrow$). First we claim that $\Pi; pc; pc \Vdash pc \succcurlyeq \nabla(\mathrm{grd}(\tau)^\rightarrow) - \nabla(H^\rightarrow)$. By Condition 2 and R-WEAKEN, know that $\Pi; pc; pc \Vdash pc \succcurlyeq \nabla(\mathrm{grd}(\tau)^\rightarrow) - \nabla(H^\rightarrow)$ if

and only if $\Pi; \mathit{pc}'; \mathit{pc}' \Vdash \mathit{pc} \succcurlyeq \nabla(\mathrm{grd}(\tau)^{\rightarrow}) - \nabla(H^{\rightarrow})$. Moreover, since $\Pi'; \mathit{pc}'; \mathit{pc}' \Vdash H^{\rightarrow} \succcurlyeq \mathrm{grd}(\tau)^{\rightarrow}$, finite iterated application of Lemma 3 gives us that $\Pi; \mathit{pc}'; \mathit{pc}' \Vdash \mathit{pc} \succcurlyeq \nabla(\mathrm{grd}(\tau)^{\rightarrow}) - \nabla(H^{\rightarrow})$. Therefore Condition 4, Lemma 14, and the definition of $\hat{H}$ give us

$$\Pi; \mathit{pc}; \mathit{pc} \nVdash \mathit{pc} \succcurlyeq \nabla(\hat{H}^{\rightarrow}) - \hat{H}^{\leftarrow}.$$

Case ($\pi = \leftarrow$). By the same argument as above noting that HOLE gives us $\Pi'; \mathit{pc}'; \mathit{pc}' \Vdash H^{\leftarrow} \succcurlyeq \mathrm{grd}(\tau)^{\leftarrow}$, $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \succcurlyeq (H - \mathrm{grd}(\tau))^{\leftarrow}$. Thus by Lemma 14, $\Pi; \mathit{pc}; \mathit{pc} \nVdash \mathit{pc} \succcurlyeq \ell^{\leftarrow} - \mathrm{grd}(\tau)^{\leftarrow}$.

Therefore, if we let x be a fresh variable in $e[\vec{\bullet}]_{H^{\pi}}$, then we can let $e' = e[x]_{H^{\pi}}$, and $\Pi; \Gamma, x : \tau; \beta; \mathit{pc} \vdash e' : \ell$ `says bool`. If we let w_i such that $a_i \longrightarrow^* w_i$, then $e'[x \mapsto w_i] \longrightarrow^* v_i$ and by Theorem 2, $v_1 = v_2$. □

Theorem 3 (Robust declassification)**.** *Let $e[\vec{\bullet}]$ be a program such that*

1. $\Pi; \Gamma, x : s, \Gamma'; \mathit{pc} \vdash e[\vec{\bullet}] : \ell$ `says bool`
2. $\Pi; \mathit{pc}; \mathit{pc} \nVdash \mathit{pc} \succcurlyeq (\ell - h)^{\leftarrow}$.

Then for all attacks $\vec{a_1}$ and $\vec{a_2}$ and all inputs v such that $\Pi; \Gamma, x : s, \Gamma'; \mathit{pc} \vdash e[\vec{a_i}] : \ell$ `says bool` and $\Pi; \Gamma; \mathit{pc} \vdash v : s$, if $e[\vec{a_i}][x \mapsto v] \longrightarrow^* v_i'$, then $v_1' = v_2'$.

Proof. Let $e'[\vec{\bullet}] = e[\vec{\bullet}][x \mapsto v]$, and let $\vec{a_i'} = \vec{a_i}[x \mapsto v]$. By Lemma 6, $\Pi; \Gamma; \mathit{pc} \vdash e'[\vec{a_i'}] : \ell$ `says bool`. Thus by Lemma 15 $v_1' = v_2'$. □

B.2 Commitment scheme verification

To prove the desired properties of commitment schemes for boolean values, let $s =$ `bool` and recall:

$\Gamma_{cro} =$ `commit`, `receive`, `open`, $x : p^{\rightarrow}$ `says` $s, y : p \wedge q^{\leftarrow}$ `says` s

- q **cannot `receive` a value that hasn't been committed.** Let $H = p^{\rightarrow} \wedge q^{\leftarrow}$. For any e and $\Gamma_{cro}; \mathit{pc}_q \vdash e : p \wedge q^{\leftarrow}$ `says` `bool`, observe that $\Pi; \mathit{pc}_q \vdash H \leq p^{\rightarrow}$ `says` `bool`, $\Pi; \mathit{pc}_q; \mathit{pc}_q \not\vdash H^{\rightarrow} \sqsubseteq p^{\rightarrow}$, $\Pi; \mathit{pc}_q; \mathit{pc}_q \not\vdash H^{\leftarrow} \sqsubseteq (p \wedge q)^{\leftarrow}$, and $\Pi; \mathit{pc}_q; \mathit{pc}_q \not\vdash \mathit{pc}_q \succcurlyeq \nabla(H^{\rightarrow}) \wedge (p \wedge q)^{\leftarrow}$. Therefore, by Theorem 2, if $e[x \mapsto v_1] \longrightarrow^* v_1'$ and $e[x \mapsto v_2] \longrightarrow^* v_2'$, then $v_1' \simeq v_2'$.

- q **cannot learn a value that hasn't been opened.** Let $H = p^{\rightarrow} \wedge q^{\leftarrow}$. For any e, ℓ, and $\Gamma_{cro}; \mathit{pc}_q \vdash e : \ell \sqcap q^{\rightarrow}$ `says` `bool`, Observe that both $\Pi; \mathit{pc}_q \vdash H \leq p^{\rightarrow}$ `says` `bool` and $\Pi; \mathit{pc}_q \vdash H \leq p \wedge q^{\rightarrow}$ `says` `bool`. Therefore, Theorem 2 applies as above for both x and y. Thus if $e[x \mapsto v_1] \longrightarrow^* v_1'$ and $e[x \mapsto v_2] \longrightarrow^* v_2'$, then $v_1' \simeq v_2'$. and if $e[x \mapsto v_1] \longrightarrow^* v_1''$ and $e[x \mapsto v_2] \longrightarrow^* v_2''$, then $v_1'' \simeq v_2''$.

- p **cannot `open` a value that hasn't been received.** Let $H = p^{\rightarrow} \wedge p^{\leftarrow}$. For any e and $\Gamma_{cro}; \mathit{pc}_p \vdash e : p^{\leftarrow} \wedge q$ `says` `bool`, observe that $\Pi; \mathit{pc}_p \vdash H \leq p^{\rightarrow}$ `says` `bool`, $\Pi; \mathit{pc}_p; \mathit{pc}_p \not\vdash H^{\rightarrow} \sqsubseteq q^{\rightarrow}$, $\Pi; \mathit{pc}_p; \mathit{pc}_p \not\vdash H^{\leftarrow} \sqsubseteq (p \wedge q)^{\leftarrow}$, and $\Pi; \mathit{pc}_p; \mathit{pc}_p \not\vdash \mathit{pc}_p \succcurlyeq \nabla(H^{\rightarrow}) \wedge (p \wedge q)^{\leftarrow}$. Therefore, by Theorem 2, if $e[x \mapsto v_1] \longrightarrow^* v_1'$ and $e[x \mapsto v_2] \longrightarrow^* v_2'$, then $v_1' \simeq v_2'$.

APPENDIX C

FLAME APPENDIX

C.1 Flame IO API

```
data IFCHandle (l::KPrin) = NewHdl { unsafeUnwrap :: System.IO.Handle }

mkStdout :: SPrin out -> IFCHandle out
mkStderr :: SPrin err -> IFCHandle err
mkStdin  :: SPrin in_ -> IFCHandle in_

hFlush :: (pc ⊑ l) => IFCHandle l -> IFC IO pc SU ()

hPrint :: (Show a, pc ⊑ l) => IFCHandle l -> a -> IFC IO pc SU ()

hPutChar :: (pc ⊑ l) => IFCHandle l -> Char -> IFC IO pc SU ()

hPutStr :: (pc ⊑ l) => IFCHandle l -> String -> IFC IO pc SU ()

hPutStrLn :: (pc ⊑ l) => IFCHandle l -> String -> IFC IO pc SU ()

hGetChar :: IFCHandle l -> IFC IO pc l Char

hGetLine :: IFCHandle l -> IFC IO pc l String

data IFCRef (l::KPrin) a = IFCRef { unsafeUnwrap :: Data.IORef a}

newIFCRef :: (pc ⊑ l) => SPrin l -> a -> IFC IO pc pc (IFCRef l a)

writeIFCRef :: (pc ⊑ l) => IFCRef l a -> a -> IFC IO pc SU ()

readIFCRef :: IFCRef l a -> IFC IO pc (pc ⊔ l) a
```

C.2 Haskell source: embargoed secret messages with macaroons

```
{−# LANGUAGE TypeOperators, PostfixOperators #−}
{−# LANGUAGE DataKinds #−}
{−# LANGUAGE OverloadedStrings #−}
{−# LANGUAGE RebindableSyntax #−}
{−# OPTIONS_GHC −fplugin Flame.Type.Solver #−}
```

```
import Prelude hiding ( return, (>>=), (>>)
                      , print, putStr, putStrLn, getLine)
import qualified Prelude as P ( return, (>>=), (>>) )
import Data.List
import Data.Proxy
import Data.String
import Data.Hex
import Data.Maybe
import Flame.Time
import Data.Time as T
import Data.ByteString.Char8 (ByteString, pack, unpack, empty)

import Flame.Prelude
import Flame.Macaroons
import qualified Macaroons as M
import Flame.Data.Principals
import Flame.Type.Principals
import Flame.Type.IFC
import Flame.Type.TCB.IFC
import Flame.IO
import qualified System.IO as SIO
import Data.IORef as SIO
import Flame.IFCRef as Ref
import Data.Functor.Identity

{- Static principals -}
alice = SName (Proxy :: Proxy "Alice")
type Alice = KName "Alice"

bob = SName (Proxy :: Proxy "Bob")
type Bob = KName "Bob"

carol = SName (Proxy :: Proxy "Carol")
type Carol = KName "Carol"

{- Alice's birthday (shared with Bob and Carol) -}
birthday :: Lbl (I Alice ∧ C (Alice ∨ Bob ∨ Carol)) Day
birthday = label $ fromGregorian 2016 9 3

{- The macaroon key -}
cKey :: Lbl Carol ByteString
cKey = label "secret"
```

```
cKeyName :: ByteString
cKeyName = "key"

carolLoc = "loc://carol"

carolOutputMacaroon :: IFCHandle ((C (Bob ∨ Carol)) ∧ (I Carol))
                    -> IFC IO ((C (Bob ∨ Carol)) ∧ (I Carol)) SU ()
carolOutputMacaroon toBobFromCarol =
    assume ((*∇) bob ≽ (*∇) carol) $
    assume ((bob*→) ≽ (carol*→)) $ do
      mac <- liftx (carol*←) carolmac
      let (serialized, MacaroonSuccess) = serialize mac MacaroonV1 in
        hPutStrLnx pc toBobFromCarol $ unpack serialized
  where
    pc = ((bob *∨ carol)*→) *∧ (carol*←)
    carolmac :: Lbl Carol Macaroon
    carolmac = bind cKey $ \key ->
               case create carolLoc key cKeyName of
                 (m, MacaroonSuccess) -> label m
                 _ -> error "Error creating macaroon"

checkTimeAfter :: SPrin pc -> SPrin l -> String -> IFC IO pc l Bool
checkTimeAfter pc l caveat =
  if "time >= " `isPrefixOf` caveat then
    case (parseTimeM True defaultTimeLocale "%Y-%m-%d" $ drop 7 caveat) of
      Just when -> do
        now <- getCurrentTimex pc
        return $ (utctDay now) >= when
      Nothing -> protect False
  else protect False

carolUpdateMessage :: SPrin p
                   -> IFCHandle (C (p ∨ Carol) ∧ I p)
                   -> IFCRef Carol String
                   -> IFC IO Carol SU ()
carolUpdateMessage p from_p message =
  do (mac, err) <- inputMac
     if err /= MacaroonSuccess then do
       error "Could not deserialize macaroon."
     else do
       v <- verifierCreatex pc pc l
       satisfyExactx pc v "op: update"
```

```
      satisfyGeneralx pc v (checkTimeAfter pc l)
      (res, _) <- verifyx pc v mac cKey []
      if res then
         assume ((p*←) ≽ (carol*←)) $ do
           msg <- hGetLinex pc from_p
           writeIFCRefx pc message msg
      else
        error "Could not verify macaroon."
 where pc = carol
       l  = carol
       inputMac :: IFC IO Carol Carol (Macaroon, ReturnCode)
       inputMac = assume ((p*←) ≽ (carol*←)) $ do
                   mac <- hGetLinex pc from_p
                   return $ deserialize . pack $ mac

bobOutputMacaroon :: IFCHandle (C (Bob ∨ Carol) ∧ I Carol)
                  -> IFCHandle (C (Alice ∨ Bob ∨ Carol) ∧ I Bob)
                  -> IFC IO (C (Alice ∨ Bob ∨ Carol) ∧ I Bob) SU ()
bobOutputMacaroon fromCarol toAlice =
  do day <- getBirthday
     (mac, err) <- inputMac
     let (mac1, MacaroonSuccess) = addFirstPartyCaveat mac (after day)
         (mac2, MacaroonSuccess) = addFirstPartyCaveat mac1 "op: read"
         (serialized, MacaroonSuccess) = serialize mac2 MacaroonV1 in
         hPutStrLnx pc toAlice $ unpack serialized
  where after day = pack $ "time >= "
                           ++ formatTime defaultTimeLocale
                                         "%Y-%m-%d" day
        pc = ((alice *∨ bob *∨ carol)*→) *∧ (bob*←)
        getBirthday :: IFC IO (C (Alice ∨ Bob ∨ Carol) ∧ I Bob)
                              (C (Alice ∨ Bob ∨ Carol) ∧ I Bob) Day
        getBirthday = assume ((alice*←) ≽ (bob*←)) $
                        liftx pc $ relabel birthday
        inputMac :: IFC IO (C (Alice ∨ Bob ∨ Carol) ∧ I Bob)
                           (C (Alice ∨ Bob ∨ Carol) ∧ I Bob)
                           (Macaroon, ReturnCode)
        inputMac = assume ((carol*←) ≽ (bob*←)) $
                   assume ((*∇) (alice *∨ carol) ≽ (*∇) bob) $
                   assume ((alice *∨ carol) ≽ bob) $ do
                    mac <- hGetLinex bob fromCarol
                    return $ deserialize . pack $ mac

carolOutputMessage :: SPrin p
```

```
                  -> IFCHandle (C (p ∨ Carol) ∧ I p)
                  -> IFCHandle (C (p ∨ Carol) ∧ I Carol)
                  -> IFCRef Carol String
                  -> IFC IO Carol SU ()
carolOutputMessage p from_p to_p message =
  do (mac, err) <- inputMac
     if err /= MacaroonSuccess then do
       error "Could not deserialize macaroon."
     else do
       v <- verifierCreatex pc pc l
       satisfyExactx pc v "op: read"
       satisfyGeneralx pc v (checkTimeAfter pc l)
       (res, _) <- verifyx pc v mac cKey []
       if res then
         assume ((*∇) p ≽ (*∇) carol) $
          assume (p ≽ carol) $ do
           msg <- readIFCRefx carol message
           hPutStrLnx carol to_p msg
       else
         error "Cannot verify macaroon."
  where pc = carol
        l  = carol
        inputMac :: IFC IO Carol Carol (Macaroon, ReturnCode)
        inputMac = assume ((p*←) ≽ (carol*←)) $ do
                     mac <- hGetLinex carol from_p
                     return $ deserialize . pack $ mac

main :: IO (Lbl SU ())
main = do
  {- create a protected reference for bob's message -}
  lmsg <- runIFC $ newIFCRefx publicTrusted carol "no message"
  let message = unlabelPT lmsg in do
    {- carol sends bob a macaroon -}
    runIFC $ carolOutputMacaroon toBobFromCarol
    {- carol sends bob a macaroon -}
    runIFC $ carolUpdateMessage bob fromBobToCarol message
    {- bob gets macaroon from carol, creates caveats, and sends alice a macaroon -}
    runIFC $ bobOutputMacaroon fromCarolToBob toAliceFromBobForCarol
    {- bob gets macaroon from carol, creates caveats, and sends alice a macaroon -}
    runIFC $ carolOutputMessage alice fromAliceToCarol toAliceFromCarol message
 where
   {- Channel: Carol -> Bob -}
   toBobFromCarol  = mkStdout (((bob *∨ carol)*→) *∧ (carol*←))
```

```
fromCarolToBob  = mkStdin  (((bob *∨ carol)*→) *∧ (carol*←))
{− Channel: Bob −> Carol −}
toCarolFromBob  = mkStdout  (((bob *∨ carol)*→) *∧ (bob*←))
fromBobToCarol  = mkStdin  (((bob *∨ carol)*→) *∧ (bob*←))
{− Channel: Bob −> Alice −}
toAliceFromBobForCarol = mkStdout (((alice *∨ bob *∨ carol)*→) *∧ (bob*←))
{− Channel: Alice  −> Carol −}
fromAliceToCarol  = mkStdin  (((alice *∨ carol)*→) *∧ (alice*←))
{− Channel: Carol −> Alice −}
toAliceFromCarol = mkStdout  (((alice *∨ carol)*→) *∧ (carol*←))
(>>)   = (P.>>)
(>>=)  = (P.>>=)
```

BIBLIOGRAPHY

[1] Martín Abadi. Logic in access control. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, LICS '03, pages 228–233, Washington, DC, USA, 2003. IEEE Computer Society.

[2] Martín Abadi. Access control in a core calculus of dependency. In *11th ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 263–273, New York, NY, USA, 2006. ACM.

[3] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, January 1999.

[4] Martín Abadi, Michael Burrows, Butler W. Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Trans. on Programming Languages and Systems*, 15(4):706–734, 1993.

[5] Martín Abadi. Variations in access control logic. In Ron van der Meyden and Leendert van der Torre, editors, *Deontic Logic in Computer Science*, volume 5076 of *Lecture Notes in Computer Science*, pages 96–109. Springer Berlin Heidelberg, 2008.

[6] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *IEEE Symp. on Security and Privacy*, pages 191–205, May 2012.

[7] Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization: Technical report. Technical Report 1813–40138, Cornell University Computing and Information Science, May 2015.

[8] Christiaan Baaij. The ghc-typelits-natnormalise package. `http://hackage.haskell.org/package/ghc-typelits-natnormalise`.

[9] Sruthi Bandhakavi, William Winsborough, and Marianne Winslett. A trust management approach for flexible policy management in security-typed languages. In *Computer Security Foundations Symposium, 2008*, pages 33–47, 2008.

[10] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011.

[11] Moritz Y Becker. Information flow in trust management systems. *Journal of Computer Security*, 20(6):677–708, 2012.

[12] Moritz Y Becker, Cédric Fournet, and Andrew D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010.

[13] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[14] Gérard Boudol. Secure information flow as a safety property. In *Formal Aspects in Security and Trust (FAST)*, pages 20–34. Springer, 2008.

[15] Niklas Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Programming Languages and Systems*, pages 180–196. March 2006.

[16] Niklas Broberg and David Sands. Paralocks—Role-based information flow control and beyond. In *37th ACM Symp. on Principles of Programming Languages (POPL)*, January 2010.

[17] Niklas Broberg, Bart van Delft, and David Sands. The anatomy and facets of dynamic policies. In *IEEE Symp. on Computer Security Foundations (CSF)*. IEEE, 2015.

[18] Jeremy W Bryans, Maciej Koutny, Laurent Mazaré, and Peter YA Ryan. Opacity generalised to transition systems. *International Journal of Information Security*, 7(6):421–435, 2008.

[19] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *20th ACM SIGPLAN Int'l Conf. on Functional Programming*, ICFP 2015, pages 289–301. ACM, 2015.

[20] Mike Cardwell. Abusing HTTP status codes to expose private information, 2011. `https://grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information`.

[21] Hubie Chen and Stephen Chong. Owned policies for information security. In *17th IEEE Computer Security Foundations Workshop (CSFW)*, June 2004.

[22] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions

for usable information flow control in Aeolus. In *2012 USENIX Annual Technical Conference*, June 2012.

[23] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 242–253, July 2006.

[24] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *16th USENIX Security Symp.*, August 2007.

[25] Karl Crary, Aleksey Kliger, and Frank Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15(2):249–291, March 2005.

[26] Dominique Devriese and Frank Piessens. Information flow enforcement in monadic libraries. In *Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '11, pages 59–72, New York, NY, USA, 2011. ACM.

[27] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-malleable cryptography. In *SIAM Journal on Computing*, pages 542–552, 2000.

[28] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *20th ACM Symp. on Operating System Principles (SOSP)*, October 2005.

[29] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *2012 Haskell Symposium*, pages 117–130. ACM, September 2012.

[30] C. Ellison. SPKI requirements. Internet RFC-2692, September 1999.

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

[32] Robert Escriva. libmacaroons, 2016. `https://github.com/rescrv/libmacaroons`.

[33] David Ferraiolo and Richard Kuhn. Role-based access controls. In *15th National Computer Security Conference*, 1992.

[34] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, 2006.

[35] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[36] Adam Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 11–22. ACM, 2015.

[37] Yuri Gurevich and Itay Neeman. DKAL: Distributed-knowledge authorization language. In *IEEE Symp. on Computer Security Foundations (CSF)*, pages 149–162. IEEE, 2008.

[38] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, January 1998.

[39] Michael Hicks, Stephen Tse, Boniface Hicks, and Steve Zdancewic. Dynamic updating of information-flow policies. In *Foundations of Computer Security Workshop*, 2005.

[40] Jon Howell and David Kotz. A formal semantics for SPKI. In *ESORICS 2000*, volume 1895 of *Lecture Notes in Computer Science*, pages 140–158. Springer Berlin Heidelberg, 2000.

[41] John Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1):67–111, 2000.

[42] Jean-Baptiste Jeannin, Guido de Caso, Juan Chen, Yuri Gurevich, Prasad Naldurg, and Nikhil Swamy. DKAL$\star$: Constructing executable specifications of authorization protocols. In *Engineering Secure Software and Systems*, pages 139–154. Springer, 2013.

[43] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: A programming language for authorization and audit. In *13th ACM SIGPLAN Int'l Conf. on Functional Programming*, September 2008.

[44] Oleg Kiselyov and Chung-chieh Shan. Functional Pearl: Implicit configurations–or, type classes reflect the values of types. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 33–44. ACM, 2004.

[45] Edward Kmett. Parameterized monads in haskell. http://comonad.com/reader/2007/%20parameterized-%20monads-%20in-%20haskell/.

[46] Edward Kmett. The reflection-extras package. http://hackage.haskell.org/package/reflection-extras.

[47] Edward Kmett. The reflection package. http://hackage.haskell.org/package/reflection.

[48] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[49] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *13th ACM Symp. on Operating System Principles (SOSP)*, pages 165–182, October 1991. *Operating System Review*, 253(5).

[50] Ninghui Li, Benjamin N Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, 2003.

[51] Ninghui Li, John C Mitchell, and William H Winsborough. Design of a role-based trust-management framework. In *IEEE Symp. on Security and Privacy*, pages 114–130, 2002.

[52] Peng Li and Steve Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

[53] Sam Lindley and Conor Mcbride. Hasochism: The pleasure and pain of dependently typed haskell programming. In *In Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Haskell '13*, pages 81–92, 2013.

[54] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, October 2009.

[55] Jed Liu and Andrew C. Myers. Defining and enforcing referential security. In *3rd Conf. on Principles of Security and Trust (POST)*, pages 199–219, April 2014.

[56] Bob Martin, Mason Brown, Alan Paller, Dennis Kirby, and Steve Christey. 2011 cwe/sans top 25 most dangerous software errors. *Common Weakness Enumeration*, 7515, 2011.

[57] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[58] Microsoft. Introduction to code signing. `https://msdn.microsoft.com/en-us/library/ms537361(v=vs.85).aspx`.

[59] Kazuhiro Minami and David Kotz. Secure context-sensitive authorization. *Journal of Pervasive and Mobile Computing*, 1(1):123–156, March 2005.

[60] Kazuhiro Minami and David Kotz. Scalability in a secure distributed proof system. In *4th International Conference on Pervasive Computing*, volume 3968 of *Lecture Notes in Computer Science*, pages 220–237, Dublin, Ireland, May 2006. Springer-Verlag.

[61] Benoît Montagu, Benjamin C. Pierce, and Randy Pollack. A theory of information-flow labels. In *26th IEEE Symp. on Computer Security Foundations (CSF)*, pages 3–17, June 2013.

[62] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[63] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[64] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[65] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[66] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symp. on Security and Privacy*, pages 165–179, 2011.

[67] Moni Naor. Bit commitment using pseudorandomness. *Journal of cryptology*, 4(2):151–158, 1991.

[68] Dominic Orchard and Tom Schrijvers. Haskell type constraints unleashed. In *International Symposium on Functional and Logic Programming*, pages 56–71. Springer, 2010.

[69] James Lee Parker. *LMonad: Information flow control for Haskell web applications*. PhD thesis, University of Maryland, College Park, 2014.

[70] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. on Programming Languages and Systems*, 25(1), January 2003.

[71] Aza Raskin. How to detect the social sites your visitors use, 2011. `http://www.azarask.in/blog/post/socialhistoryjs`.

[72] RedMonk. The redmonk programming language rankings: January 2016, January 2016. `http://redmonk.com/sogrady/2016/02/19/language-rankings-1-16`.

[73] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2009.

[74] Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 13–24. ACM, 2008.

[75] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[76] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.

[77] Ravi S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In *4th European Symp. on Research in Computer Security (ESORICS)*, September 1996.

[78] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus Authorization

Logic (NAL): Design rationale and applications. *ACM Trans. Inf. Syst. Secur.*, 14(1):8:1–8:28, June 2011.

[79] Austin Seipp. Reflecting values to types and back. `https://www.schoolofhaskell.com/user/thoughtpolice/using-reflection`.

[80] Deian Stefan, Alejandro Russo, David Mazières, and John C Mitchell. Disjunction category labels. In *Proceedings of the 16th Nordic conference on Information Security Technology for Applications*, pages 223–239, 2011.

[81] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[82] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *16th ACM SIGPLAN Int'l Conf. on Functional Programming*, ICFP '11, pages 266–278, New York, NY, USA, 2011. ACM.

[83] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 202–216, July 2006.

[84] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 137–148. ACM, 2012.

[85] The Glasgow Haskell Compiler, 2016. `https://www.haskell.org/ghc/`.

[86] Mahesh V Tripunitara and Ninghui Li. A theory for comparing the expressive power of access control models. *Journal of Computer Security*, 15(2):231–272, 2007.

[87] Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. In *9th ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 115–125, 2004.

[88] Philip Wadler. Propositions as types. *Communications of the ACM*, 2015.

[89] Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, and Alejandro Russo. It's my privilege: Controlling downgrading in DC-labels. In *Proceedings of the 11th International Workshop on Security and Trust Management*, September 2015.

[90] William H Winsborough and Ninghui Li. Towards practical automated trust negotiation. In *3rd Policies for Distributed Systems and Networks*, pages 92–103. IEEE, 2002.

[91] William H Winsborough and Ninghui Li. Safety in automated trust negotiation. In *IEEE Symp. on Security and Privacy*, pages 147–160, May 2004.

[92] William H Winsborough, Kent E Seamons, and Vicki E Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 1, pages 88–102, January 2000.

[93] Marianne Winslett, Charles C Zhang, and Piero A Bonatti. Peeraccess: A logic for distributed authorization. In *19th ACM Conf. on Computer and Communications Security (CCS)*, pages 168–179. ACM, 2005.

[94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[95] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66. ACM, 2012.

[96] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 15–23, June 2001.

[97] Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2–3):209–234, September 2002.

[98] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. on Computer Systems*, 20(3):283–328, August 2002.

[99] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[100] Charles C Zhang and Marianne Winslett. Distributed authorization by multiparty trust negotiation. In *ESORICS 2008*, pages 282–299. Springer, 2008.

[101] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 272–286, June 2005.

[102] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3), March 2007.