TRUST, AUTHORITY, AND INFORMATION FLOW IN SECURE DISTRIBUTED SYSTEMS

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

Michael David George

December 2020

© 2020 Michael David George
ALL RIGHTS RESERVED

TRUST, AUTHORITY, AND INFORMATION FLOW IN SECURE DISTRIBUTED SYSTEMS

Michael David George, Ph.D.

Cornell University 2020

Computing systems that make use of other people's data are an important and pervasive aspect of modern life. However, users have little ability to specify how their information should be used, and system providers have insufficient tools to provide assurance that they correctly handle that information.

Decentralized information flow control (DIFC) provides a framework for specifying policies on the use of information and ensuring that systems abide by those policies. However, existing DIFC systems do not address the complexity of the modern software ecosystem, where multiple entities interact to provide the software, data storage, and computational resources for a given application.

This dissertation aims to bridge the divide between the strong security guarantees provided by DIFC systems and the complex requirements of today's distributed applications.

We present Fabric, a platform for secure distributed storage and computation. Fabric provides a high-level programming model that enables heterogeneously trusted parties to share code, information, and computational resources while enforcing strong information flow constraints. We have evaluated Fabric by building applications that model the data storage, communication, and software evolution patterns present in existing applications, while adding appropriate information flow

constraints. Results from these applications suggest that Fabric has a clean, concise programming model, offers good performance, and enforces strong security policies.

In a decentralized setting like Fabric, it should be the case that a principal's security policy can only be violated if someone they trust misbehaves. Formalizing these notions in a decentralized system is challenging because statements of trust and definitions of policies are also decentralized.

To address this challenge, we show how to identify the data that influences information flows in a broad class of DIFC systems. We use influenced flows to formalize the notion that you can only be harmed by those you trust, and show that definitions of harm based on influenced flows generalize existing semantic definitions of information flow security.

Since trust statements in federated systems are distributed, determining whether a computation is safe may require a separate distributed computation, which may in turn introduce potentially unsafe information flows. We present a programming language for distributed authorization queries that tracks information flow. We show that programs in our language do not leak information, and that the language is sufficiently expressive to model nontrivial authorization queries.

BIOGRAPHICAL SKETCH

Michael completed his Bachelor of Arts degree with Honors in Mathematics and his Bachelor of Science in Computer Science at the University of Rochester in 2003. He then completed his Master of Arts degree in Mathematics in 2004 before heading to Cornell to work on his Doctorate.

He has always been excited about teaching; as a youth he loved the opportunity to teach scoutcraft to the younger scouts even though he complained that "they won't sit still and they won't shut up!" Since then he has had the opportunity to work with more mature students as a lecturer in Cornell's computer science department. He has taught a variety of courses including discrete mathematics, object-oriented programming, functional programming, operating systems, and algorithms.

Michael's other interests include playing the piano and fiddle, making ceramics, and contra dancing. He was married to Laura George in 2019, and has been serving as a deacon in the Presbyterian Church since early 2020.

This dissertation is dedicated

to those who hold a part of me,

and those of whom I hold a part.

ACKNOWLEDGEMENTS

This work would not have been possible without the constant support of my wife, Laura George, my parents Natalie and Donald George, nor my sister Valerie Garrison. They have and will remain the most important part of who I am.

I also owe a great debt of gratitude to my advisor, Andrew Myers, for his patient guidance as I learned how to do research as well as his many technical contributions to the work presented herein. I would also like to thank the members of my special committee: Fred Schneider, Hakim Weatherspoon and Lawrence Gibbons, for their helpful guidance and feedback.

My research is inextricably linked to the projects that I have worked on with the Applied Programming Languages group at Cornell. I would especially like to thank Jed Liu, K. Vikram, Owen Arden, Aslan Askarov, and Lucas Waye for their expertise and friendship, both of which have had a big impact on this work.

TABLE OF CONTENTS

1 Introduction **1**
1.1 The State of the Art 2
1.2 Contributions 6
1.2.1 The Fabric System 6
1.2.2 Decentralized Information Flow Analysis 9
1.2.3 Flows Through Dynamic Authorization 13

2 The Fabric System **16**
2.1 A Running Example 16
2.1.1 Security Considerations 18
2.1.2 Software Construction and Evolution 19
2.2 Fabric Programming Model 20
2.2.1 Data in Fabric 21
2.2.2 Computation in Fabric 22
2.2.3 Evolving Secure Software 24
2.2.4 Principals 28
2.2.5 Labels 31
2.2.6 The Decentralized Label Model 34
2.2.7 Information Flow 37
2.2.8 Novel Information Flow Constraints 39
2.2.9 Transactions 43
2.2.10 Exceptions and Rollback 45
2.2.11 Interacting With the Outside World 47
2.2.12 Summary of the Fabric Programming Model 48
2.3 The Fabric System 48
2.3.1 Communications Layer 50
2.3.2 Distributed Objects 52
2.3.3 Dynamic Fetch Authorization 54
2.3.4 Dynamic Type Checking 56
2.3.5 Concurrency 57
2.3.6 Distributed Transaction Management 58
2.3.7 Nested Transactions 59
2.3.8 Hierarchical Commits 61
2.4 Evaluation 63
2.4.1 System Implementation 64
2.4.2 FriendMap Application 65

2.4.3 Course Management System 68
2.4.4 OO7 Benchmark 71
2.4.5 Other Applications 72
2.5 Related Work 74
2.6 Summary 80

3 The Decentralized Security Principle 82
3.1 System Model 84
3.1.1 System Model Requirements 85
3.1.2 Information Flow 85
3.1.3 Transmission and Relabeling Flows 89
3.1.4 Influenced Flows 90
3.1.5 Influenced Flows Generalize Flows 92
3.2 The Decentralized Security Principle, Formalized 92
3.2.1 Label Model Axioms 93
3.2.2 DSP with Strict Harm 97
3.2.3 Strict Harm and Noninterference 100
3.2.4 DSP with Downgrading 101
3.2.5 Influenced Flows and Nonmalleability 105
3.3 The Extended Decentralized Label Model 106
3.3.1 Constructing a DTH 109
3.4 Application to Fabric 111
3.5 Related Work 114

4 Information Leaks via Authorization Requests 116
4.1 System Overview 120
4.2 Language and System Model 124
4.2.1 Standard Features 124
4.2.2 Principals and Delegation 124
4.2.3 Distributed State 128
4.2.4 Dependent Types and Proofs 130
4.2.5 Labeled Values and Relabeling 134
4.3 Security Condition 136
4.3.1 Definitions 137
4.3.2 Proof of Security 140
4.4 Actsfor Revisited 143
4.5 Lessons for Fabric 145
4.6 Revocation 148

4.7 Related Work 152

5 Conclusions **154**
5.1 Contributions to Distributed Software Platforms 154
5.2 Contributions to Information Flow Analysis 155
5.3 Contributions to Dynamic Distributed Authorization 156
5.4 Future work 157
5.5 Summary 160

Bibliography **161**

LIST OF FIGURES

2.1 Overview of the FriendMap social mashup example. 17
2.2 The stores and objects in the FriendMap example application. 23
2.3 Evolution in the FriendMap example. 27
2.4 Principal hierarchy from the FriendMap example. 29
2.5 Syntax of labels in the Decentralized Label Model. 35
2.6 Implicit flow example 38
2.7 Overview of the components of Fabric workers and stores. 49
2.8 Cache hierarchy for Fabric objects. 52
2.9 Logs of nested distributed transactions 60
2.10 A portion of the FriendMap application implementation. 67

3.1 The syntax of EDLM labels 107
3.2 The EDLM flows-to relation 108
3.3 Construction of a DTH $[\![S]\!]$ from a set S of trust assertions. 110

4.1 Example of read channels in dynamic authorization. 118
4.2 Actsfor implementation. 122
4.3 An example Fabric program that uses dynamic acts-for checking . . . 123
4.4 Syntax and semantics for standard language features. 125
4.5 Syntax and semantics for language features for principals and delegation 126
4.6 Relation defining $\ell \preccurlyeq \ell'$. 127
4.7 Syntax and semantics for language features for distributed state . . . 129
4.8 Language features for proofs and dependent types 131
4.9 The static implication relation 133
4.10 Language features for information flow 135
4.11 Dynamic label and principal tests in Fabric. 145
4.12 Broker example using class **where** constraints. 147

Chapter 1
Introduction

We have entered an era saturated with digital data about people's personal lives. A large and growing portion of the population uses some form of social media to share details of their day-to-day activities. Businesses rely on digital systems to track the behavior of customers and employees alike, and this tracking is becoming easier as more of our transactions are taking place online. Health care and government institutions maintain records about the populace as well.

Digital data has great potential to improve our lives—limited only by our ability to come up with new and interesting ways to combine it. To address the demand for new ways to integrate and share data, a diverse ecosystem of software developers has sprung up, including large institutional players like Facebook and Google, and a multitude of smaller websites and application developers.

The key ingredient for this kind of ecosystem to thrive is the ability to seamlessly combine software and data from an open-ended set of sources. However, that ability comes with some risk: users consider the data that describes their lives to be private, and they require assurance that such data remains confidential. Moreover, because they rely on these systems more and more, the integrity and availability of data is also crucial. Allowing arbitrary combinations of software to operate on and share sensitive data can clearly violate these confidentiality, integrity, and availability requirements.

In an open software ecosystem, there are many parties that have security concerns. Each user cares about the confidentiality and integrity of their own information. Service providers also have security concerns: they may wish to protect the

confidentiality of their intellectual property or the integrity of the information that they give to users, for example. In fact, many service providers offer detailed privacy policies that legally require them to enforce per-user privacy settings.

The ability to combine data owned by different parties means that everyone's security concerns must be considered in tandem. If confidential data owned by multiple users is combined into a web page, for example, then each of the users must be satisfied that the web page is only viewed by principals who are allowed to learn that confidential information.

1.1 The State of the Art

Today's computing platforms typically address these security requirements by isolating applications from one another. For web applications, this approach is typified by the same-origin policy [Zal09], which prevents a web site from interacting with browser state that is associated with a different site. This rule was adopted to prevent cross-site scripting attacks, in which an untrusted web page examines the content of a trusted web application and then communicates that information back to a third party.

Unfortunately the same-origin policy is both too strong and too weak. It is too strong because service providers want to integrate third-party software (and advertisements) with our data. To achieve this functionality, service providers host content on behalf of advertisers and third party developers to make it appear as if the content all came from a single origin.

The same-origin policy is also too weak. Once a trusted party is willing to host content for an untrusted party (such as an advertiser), the same-origin policy can no longer protect sensitive information. Since the untrusted portion of the application comes from the same origin as the trusted application, it can observe the confidential information, and then smuggle that information to a third party in the form of a request, by loading an image for example.

Another example of the isolation approach to security can be found in the "app stores" that have become popular for mobile devices. When a user finds an application that they wish to install, the system provides them with a set of "permissions", such as access to the camera, storage device, the internet, or their contact database. Typically the user performs a mental calculation—they make a guess about what the application will do with the permissions that are granted to it, and then decide what the implications of those actions might be on the security of their information.

Once installed, the system will prevent the application from performing any sensitive operations for which it was not granted permission, but there is no way to provide assurance that the application acts as the user suspected. For example, an email application would naturally expect permission to access a user's contact list and also the internet. The user probably doesn't expect the program to periodically send an email to the author of the program containing all of the user's contacts, but nothing in the system prevents the program from doing so. Similarly, mobile games typically request permission to access the state of the telephone functionality, presumably so that they can pause themselves if a call comes in. But again, this is only an informal expectation—nothing prevents the game from posting your information

publicly, or even subversively encoding it into the high scores that it posts on the high-scores list.

The problem is that users must specify *access control* policies as a stand-in for the end-to-end *information flow* policies that they expect to be enforced. What the user cares about in these examples is who is allowed to learn about or affect specific pieces of data—regardless of whether the information is directly communicated to those parties or communicated through a third-party service, regardless of whether the channel goes through a single application or through multiple applications, and regardless of whether the information is communicated via disk storage, interprocess communication, networking, or some other mechanism.

As applications incorporate more and more functionality, they require more and more permissions. Isolation approaches typically lead to user fatigue, where users are unable or unwilling to reason about what an application might be doing, and just develop the habit of clicking "accept".

These problems stem from the fact that isolation-based security approaches are noncompositional: even if two components of a system are independently secure, the interactions between them may not be. When advertising and content interact in nontrivial ways in web applications, or when a mobile application combines looking up addresses and sending email, or when managing phone calls and posting scores are combined in a game, one has to look at the details of the combination to determine whether it is safe.

Information flow control [DD77, Mye99a, Sim03, ZBWM08, KYB+07] is an appealing approach to security which addresses these difficulties. Information flow

control systems track the flow of information through computations, ensuring that confidential data cannot affect less confidential output in any way. Information flow control enables developers and users to reason about the end-to-end impact of their policies without understanding the details of the systems that handle their data.

The theory of information flow control in individual programs running on trusted platforms is fairly well understood. However, applying information flow control to the modern distributed software ecosystem presents new challenges. Applications do not run on a single device, or even on devices that are administered by a single entity. Modern applications typically perform some parts of their computations on a client (typically administered by a user) and some parts on servers administered by cloud providers such as Google or Amazon. They also commonly access servers operated by the application providers and third parties, and may even span systems operated by multiple end users.

For these applications, there is no single entity that everyone trusts to enforce their security policies. Instead, each principal trusts some subset of the components of the system. This situation invalidates a key assumption underlying existing information flow control techniques: that there is a trustworthy entity that can statically check or dynamically monitor the running program to ensure it complies with information flow policies.

1.2 Contributions

Our goal is to adapt the techniques of information flow control to the modern landscape of distributed applications running on a partially trusted platform. This landscape of heterogeneous trust presents conceptual and practical challenges for the design and implementation of secure software. This dissertation addresses some of these challenges.

1.2.1 The Fabric System

Our first contribution to this effort, described in Chapter 2, is Fabric, a programming language and distributed system that serves as a platform for building secure federated systems. Fabric makes information flow analysis more tractable by integrating all components of a distributed application into a coherent program written in a single language using shared high-level abstractions.

This approach contrasts with the architecture of today's global computing infrastructure. Modern applications rely on a hodgepodge of interacting technologies, including browsers using JavaScript to operate over HTML and XML, native applications for mobile devices and desktop computers written Java, Objective C, and many other languages, web services written in a variety of static and dynamic languages, data storage ranging from flat files to structured relational databases to global cloud storage services.

Fabric raises the level of abstraction by integrating the tools needed to build distributed applications into the language and programming model. Higher level

primitives make it easier to reason about the behavior of the system as a whole, which is necessary for giving programmers and users strong end-to-end security assurance.

Providing a unified language makes both programming and program analysis more tractable, but we must also ensure that the language is expressive enough to model applications with the features that users have come to expect. The key features of today's software ecosystem that Fabric models include the following:

- **An open system** Fabric is an open platform like the web, rather than a collection of closed services like today's clouds. Anyone can bring code, data, or computational resources to the table, and those resources are all treated uniformly by the platform.
- **Distributed computing** Fabric supports computations that span multiple locations and trust domains. For example, parts of a typical web application might run on the user's browser, on the provider's web server, on a back-end database server, and on a third party's web service. Fabric supports this kind of design through function shipping and data shipping.
- **Persistent storage** Support for distributed persistent storage is tightly integrated into Fabric's computational model. All Fabric data is stored using the same interface, allowing us to reason about information flows through storage.
- **Mobile code** Fabric allows any developer to add new code to the system and allows code to be dynamically shipped to different hosts for execution. This mechanism is analogous to shipping of JavaScript code to browsers, shipping of SQL queries to databases, and even the distribution of application code through app stores.

Fabric extends these capabilities by supporting end-to-end information flow security, thereby enabling users to run modern integrated applications with assurance that their security policies will be respected.

To achieve these ends, trustworthy Fabric nodes perform information flow analysis on application code prior to execution. This analysis is aware of the principals that are responsible for operating different components of the system and the trust relationships between them, and does not make any assumptions about the behavior of untrusted components.

The analysis is also aware of the information flow policies associated with the data that the applications manipulate. It is therefore able to determine whether the applications abide by those policies, even if untrusted principals are acting maliciously. A key contribution of Fabric is the adaptation of prior information flow control analyses to function in the presence of partially trusted code on a partially trusted platform.

Integrating features for distributed computation and persistent storage into Fabric presents new challenges because previous systems that provide these features have not been concerned about potential information flows. To address these challenges, we have developed novel implementation techniques that avoid information leaks present in previous systems. These techniques are another major contribution of the Fabric system.

Fabric's computational model provides enough flexibility to faithfully model modern applications. We validate this claim by presenting a variety of example applications that we have implemented in Fabric. Most notably, we have built a model of a

social network that allows untrusted applications to manipulate users' private data while preserving the security requirements of the users. We show that Fabric's information flow control mechanisms ensure that the security requirements of each user are preserved by this application. Although existing social networking service allow third parties to develop applications today, we believe this is the first example of a service that integrates third-party applications while providing strong information flow guarantees.

Although Fabric's primary purpose is to provide a platform for investigating the security of integrated software platforms, it is important to show that the techniques we have developed can be made to perform efficiently. Chapter 2 contains a performance evaluation of a number of applications that we have implemented in Fabric; it shows that we need not sacrifice performance to achieve strong security.

1.2.2 Decentralized Information Flow Analysis

The second major contribution of this dissertation, described in Chapter 3, is a mathematical framework for evaluating partially trusted information flow control systems.

The gold standard for information flow control systems is a property called *noninterference* [SM03]. A system satisfies noninterference if an observer that sees only public output from the system can infer nothing about the secret input, and an attacker that can only influence low-integrity input cannot affect high-integrity output in any way.

In practice, noninterference is too restrictive to describe many systems that are considered secure. In practice, information flow control systems provide a weakened form of noninterference that allows private information to be declassified, or low integrity information to be endorsed. There are a variety of security properties that describe the safety of information flow in the presence of downgrading [SS05].

Proving that a system satisfies any of these information flow properties requires assumptions about the implementation of the system. For example, an enforcement mechanism that uses static program analysis assumes that the program that is being analyzed is the same as the program that is being executed, and that the hardware that executes the program does not allow private information to be read directly by attackers. These assumptions make sense when the system is running on a trusted, centralized platform, because users can rely on physical security and secure hardware to ensure the integrity of the platform.

In an open software ecosystem like Fabric, these assumptions are far too strong. Indeed, the definition of an open system is that any party can provide computational infrastructure. There is no reason to assume (and no way to verify) that an arbitrary participant correctly implements any security mechanisms.

Cryptographic techniques such as secure multi-party computation [CLOS02] and encrypted query processing [PRZB11] provide strong guarantees of security properties even for programs executing on untrustworthy platforms. However, these techniques tend to be specialized and expensive; they are not yet practical for application to general software [NLV11].

Instead, we rely on the insight that while no single component of the global computational infrastructure is trusted by everybody, each participant does trust some portion of the infrastructure. By making use of users' stated trust assumptions, we can state security properties that are relativized to each user's trusted computing base.

We refer to this idea as the Decentralized Security Principle (or DSP). Informally, the DSP states that a user can only be harmed if someone they trust is untrustworthy, and that the harm is limited to the extent of the trust that is granted. Making this property precise and proving that a system satisfies it requires clear definitions of "harm", "trust", and "trustworthiness".

In Chapter 3, we use the DSP to give a relativized definition of noninterference. We formalize the idea that a user is harmed if their confidential information affects data in the system in a manner that conflicts with their policies, or if their high-integrity data is modified in a way that is not specified by their policies. This formalization is consistent with noninterference-based definitions: we show that the total absence of harm is equivalent to noninterference.

We also give a relativized definition of a weakened form of noninterference called nonmalleable information flow [CMA17]. Here, the definition of harm is more subtle: information flows are allowed, but only if they are mediated by data with appropriate confidentiality and integrity constraints. We give a formal definition of an "influenced flow" and use it to define a security condition that is very similar to nonmalleable information flow.

In our model, a statement of trust has a well-defined and universal meaning. Roughly speaking, a user should express trust in another party (the *trustee*) if they believe that the trustee faithfully abides by the required system's semantics. If the trustee actually does correctly follow the system's semantics, we say that they are *trustworthy*. Our formal definition of trustworthiness for the information flow setting is given in Chapter 3.

Trustworthiness is *universal* in the sense that the formal definition is not application-specific; a user's statement of trust does not assert that the trustee can make correct *decisions* about the appropriate use of data, only that they correctly secure their systems.

Trustworthiness is *local* in the sense that the trustworthiness of a principal depends only on the actions that the principal takes. This means that an individual can use physical security, process management, and secure hardware [CD16, NMB+16] to ensure that their systems are trustworthy, even though they cannot ensure that other actors in the system are trustworthy. Because trustworthiness is local, service providers can reasonably claim to be trustworthy, although it is up to users to decide whether to accept those claims or not.

Trust should not be an all-or-nothing proposition, especially in an open software ecosystem designed to allow integration of different kinds of data. Each user trusts different principals to different extents. To capture these different levels of trust, each statement of trust in our model is *qualified* by the set of policies that may be violated if the trustee is not trustworthy.

1.2.3 Flows Through Dynamic Authorization

Trustworthy applications that handle sensitive data or perform sensitive operations must be able to determine the relevant security policies so that they can ensure that they are respecting them. In a distributed system, these policies may come from multiple sources and cannot be fixed at the time the software is written. For example, an application that integrates data from multiple social networks will draw the policies on some users' data from one social network while reading the policies on other users' data from another. In many authorization schemes (e.g., PeerAccess [WZB05], Cassandra [BS04]), even the authorization of a single action may require integration of data from multiple sources.

Implementing authorization in distributed systems is hard for many of the same reasons that implementing any secure distributed application is hard: Authorization decisions typically depend on the state of the system, but this state may be distributed across many locations and trust domains. The system must be able to answer authorization queries consistently, efficiently, with integrity, and without leaking confidential information to untrusted parties.

As a simple example, consider an authentication-as-a-service system such as OAuth [HL11]. The goal of these systems is to allow users to reuse existing accounts (such as Facebook or Google accounts) to identify themselves to third parties. This approach saves application providers from handling the tricky details of implementing authentication properly, while also reducing the proliferation of accounts that each user must manage.

Many users are justifiably reluctant to make use of these services, because they worry about the privacy implications [MH03a]. Some users do not want to allow the authentication providers to learn what websites they are logging into, when they log into them, or how often they do. Others are willing to allow these providers to learn that information, but are worried about that information leaking through the providers to their friends or the public at large.

These are fundamentally concerns about information flow. Fabric provides high-level abstractions that address these concerns, and it makes sense to apply these general tools to the reason about the specific problem of information flow through authorization. However, this presents a challenge, because the existing work on information flow assumes that there is a fixed, pre-existing set of authorized flows [SM03]. This assumption is inconsistent with our desire to analyze realistic authorization protocols.

Chapter 4 examines the issue of information flows through dynamic authorization requests in detail. We apply the formal concept of flow developed in Chapter 3 to a model programming language that contains a simplified version of Fabric's security and communication mechanisms. We present an information flow type system and formally prove that any well-typed program exhibits no harmful flows. We then implement a model of the authorization primitives that are built into Fabric as a well-typed program in this language, thus showing that Fabric's dynamic authorizations can be implemented securely.

Our analysis goes beyond previous work because we explicitly model the distribution of the authorization state throughout the system, and we ensure that no accesses

to the distributed state can leak information inappropriately. Our language model also avoids the assumption the authorization state is fixed, which makes the analysis more suitable for a long-running open system.

The language we have developed in Chapter 4 is interesting in its own right. We make use of a restricted form of dependent types to allow programmers to dynamically construct security proofs in their programs. We discuss how the same technique could be applied to the Fabric language to simplify and clarify common design patterns. This approach was inspired by research that uses dynamically constructed proofs to improve the scalability of distributed systems [LMA+14].

In summary, this dissertation presents three primary contributions: the design and implementation of the Fabric system, the formal definition and application of the decentralized security principle, and the introduction and analysis of the problem of information leaks through dynamic authorization requests. These contributions are concrete steps towards the vision of a modern integrated software ecosystem that preserves the security of all participants, but many challenges remain. Chapter 5 concludes with a discussion of some potential next steps.

Chapter 2
The Fabric System

This chapter presents the design and implementation of the Fabric system.

We demonstrate Fabric's features using an example application called FriendMap. FriendMap consists of untrusted code that integrates information from a social network with data from a partially trusted mapping service. Section 2.1 describes the FriendMap application in detail.

Section 2.2 describes the details of the Fabric programming model and explains how we use Fabric's abstractions to construct the FriendMap example. Section 2.3 describes the architecture of the Fabric system and the design of the mechanisms required to securely realize that architecture.

To evaluate the design of Fabric, we have built several example applications, including the FriendMap example. Section 2.4 describes our experience with implementing these applications, and presents performance results. These results suggest that Fabric's abstractions can be implemented efficiently. We conclude in Section 2.6 by discussing related work and future directions.

2.1 A Running Example

To illustrate the security challenges faced by applications running on a partially trusted platform, we present a running example application that we call "FriendMap". This application allows a user of a social network to create a map displaying the locations of their friends. Let us call the user "Alice" and one of her friends "Bob."

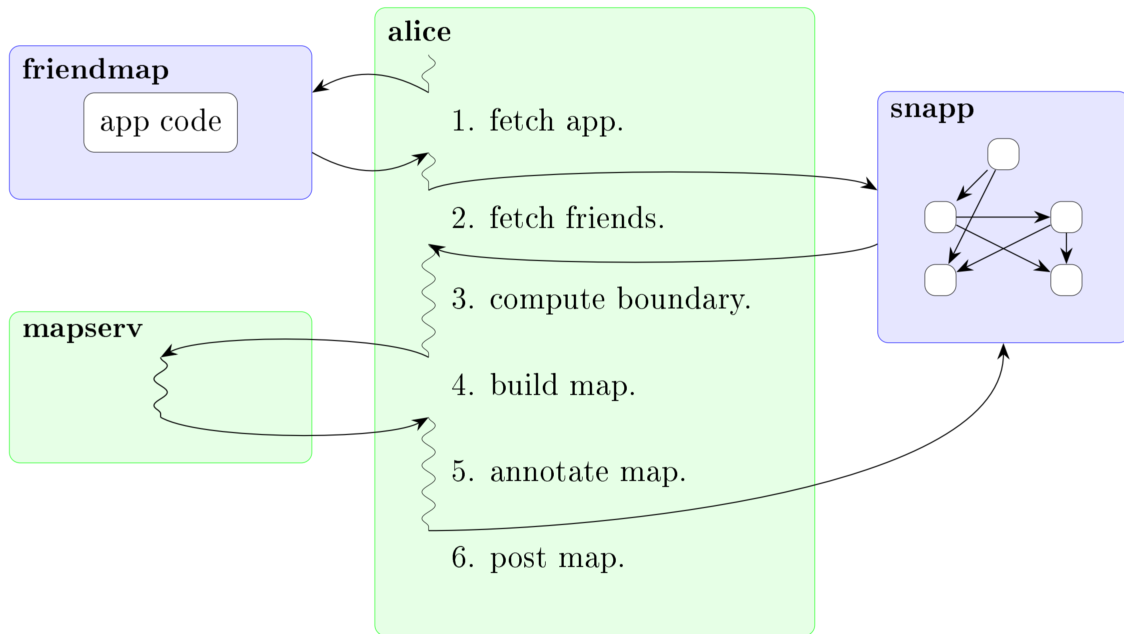

Figure 2.1: Overview of the FriendMap social mashup example.

Figure 2.1 shows the interactions Alice's client makes while executing FriendMap. First, Alice's client downloads the application code (1) to execute locally. The FriendMap application then fetches the locations of Alice's friends (2) from the social network ("Snapp"). Based on those locations, it determines what geographical area it will need to display (3). It then submits this bounding box to a third party map service ("MapServ"), which constructs a map of the area (4). The blank map is returned to Alice's client, where the friends' locations are added (5). Alice can then choose to post the map to the social network to share with her friends (6).

2.1.1 Security Considerations

Even this simple example has complex security requirements because the principals trust each other to differing degrees. For example, Alice trusts MapServ to learn some information about her friends, but Bob may not trust MapServ at all. In that case, FriendMap must avoid using his location to compute the map request.

Similarly, although Bob trusts Alice to see his location, he may not trust Alice's friends with the same information. If so, FriendMap must either avoid posting the resulting map where Alice's friends can see it or omit Bob's location from the map.

Further, none of the involved principals trust the provider of the FriendMap code. Therefore some mechanism is needed to ensure that the code enforces their policies; any principal who controls this mechanism or the node on which it operates must be trusted to enforce these policies. In this example, Bob trusts Alice to enforce the confidentiality of his location, so Alice's node is responsible for enforcing this confidentiality policy.

In real applications, policies are more nuanced than lists of entities allowed to learn information. In the FriendMap example, Bob may consider his exact location confidential, but be willing to release some information about where he is, such as the city he is in. Alternatively, he may not mind letting the public know where he was yesterday, but may wish to keep his current location secret.

These more complex policies can be thought of as a form of controlled declassification: Bob is willing to "declassify" secret information if it is processed in a certain way or if it is embargoed for a certain period of time. Although the platform must support declassification, it is critical that any downgrading is authorized. In this

example, the platform must ensure that the code performing the declassification is either provided by or endorsed by Bob.

2.1.2 Software Construction and Evolution

Enforcing the integrity of application code and the input to those programs is meaningless if the developers of the applications are unable to reason about the correctness of those programs. This task is challenging in the modern software ecosystem because applications are composed of many components that are developed by different organizations and are upgraded on different schedules. In an environment like Fabric, where different principals trust different software developers and providers, the challenges are even greater. Every user must have confidence that the integrity of the data they care about is maintained, even if the software that manipulates that data is provided by a partially trusted source.

In the FriendMap example, the FriendMap application is built on top of APIs provided by both Snapp and MapServ. As part of the development process, the FriendMap developers will make assumptions about the specifications of these APIs. These specifications are typically stated informally in documentation, if they are explicitly stated at all; but even with unstated assumptions, the developers expect software dependencies to operate consistently over time.

These requirements stand in conflict with the ability for applications to be upgraded over time. It is important for Snapp to be able to update their service to add new features, deprecate old features, and fix bugs without coordinating with the open-ended set of app developers that integrate with their platform. Similarly,

FriendMap should be able to upgrade their software to make use of new features as they become available without having to coordinate their upgrades with all of the APIs that they rely on.

To meet these goals, the components that make up today's software are typically gathered together at least twice: once by the developer for compilation and testing, and once on behalf of the end user for execution. Programs in more dynamic systems—like JavaScript libraries on the web—may be refetched and reassembled every time a page is displayed. In fact, the situation is even more complex for JavaScript, because code may be cached: One cannot even be sure that all of the components of the application were fetched at the same time.

Such dynamic relinking allows for software to be upgraded piecemeal over time, but it can also lead to violations of the assumptions that developers make about the software that they depend on. These violations can lead to corruption of high integrity data.

Fabric provides support for software interoperability and evolution while ensuring that parties that any *user* does not trust cannot violate the correctness or secrecy of their data. It does this by tracking the provenance of code in the system and using program analysis to maintain the integrity of important data.

2.2 Fabric Programming Model

Before describing how the Fabric system meets the goals described above, we present the Fabric programming model. This discussion introduces the key abstractions of

Fabric and sets the stage for our presentation of the Fabric system implementation in Section 2.3.

Application developers interface with the Fabric system using the Fabric programming language. The Fabric language is an extension of Jif [MZZ+06], which in turn extends Java [AGH05]. Fabric extends the object-oriented paradigm of these languages by providing support for secure distributed computation over persistent objects.

We assume that the reader is familiar with object-oriented programming and with the Java programming language. We do not assume the reader is familiar with Jif; relevant concepts will be introduced as they are needed.

2.2.1 Data in Fabric

As in Java, all data in Fabric are represented as *objects*. Objects have an identity that other objects can refer to, and they maintain state in the form of fields.

Unlike Java objects, Fabric objects are persistent and distributed. Each object is stored on a named host, the *store* of the object. Object references are composed of the name of the store that holds the object and a 64-bit *object identifier* (OID) that is unique within the store. Because Fabric references are global and persistent, it makes sense to serialize them into a form suitable for export from Fabric. We often encode Fabric references in URLs of the form "`fab://`*`[store]`*`/`*`[oid]`*;" we refer to these as Fabric URLs.

Because Fabric objects are distributed and persistent, they can be used to model database rows, entries in persistent key-value stores, files, and other mechanisms for

persistent storage. Similarly, Fabric references can be used to model foreign keys in a database, directory entries, or URLs on the web.

Figure 2.2 depicts some of the important objects in the FriendMap example. There are three stores, operated by Snapp, FriendMap, and MapServ respectively. Objects representing the users Alice and Bob are stored by Snapp, while FriendMap stores an object containing the code for the FriendMap application. MapServ stores objects containing the data used to construct their maps.

2.2.2 Computation in Fabric

As in Java, all Fabric computation takes place by executing methods on objects. Unlike Java, Fabric executes computations on a specified host, the *worker* of the computation. Fabric programs can explicitly transfer execution to a different worker by making a remote call. The syntax `o.f@w(...)` indicates that the method `f` of object `o` should be invoked on the worker `w`. We refer to the ability to transfer control to remote workers as *function shipping*.

Fabric has no requirement relating the worker where a computation is executing to the stores of the objects that the computation accesses. Whenever a computation accesses an object, a copy of the object is sent to the worker, and any updates to the object are sent back to the store that holds the object[1]. We refer to the process of moving data from stores to workers and back as *data shipping*.

Together, function and data shipping can model a large number of interaction patterns used in distributed software. Data shipping models situations like http GET

[1]This is a conceptual description of what happens. In reality Fabric has sophisticated caching and transactional mechanisms that are explained in Section 2.3.

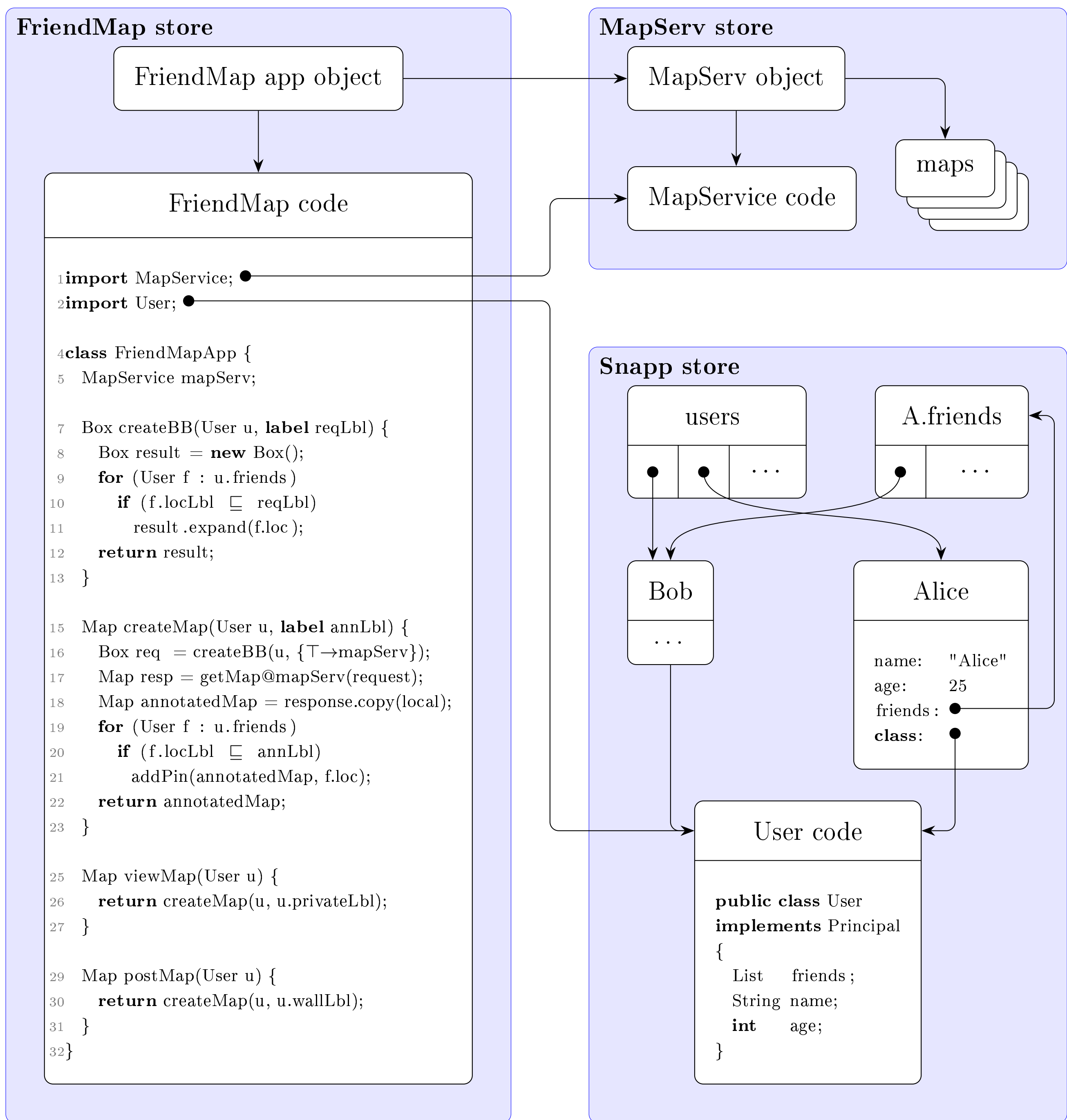

Figure 2.2: The stores and objects in the FriendMap example application. Notice that objects can refer to objects on other stores. For example the FriendMap application object points to the MapServ object. Note also that like all other data in Fabric, code is stored in Fabric objects: See Section 2.2.3 for more details.

requests and features for remote connectivity, where data moves to the host running a computation; Function shipping models situations like SQL queries, remote methods, and web APIs, where the computation moves to the data.

The FriendMap example uses data shipping to move objects from FriendMap and Snapp to Alice's worker. It also uses function shipping to construct the map on MapServ's worker to improve performance.

2.2.3 Evolving Secure Software

In Fabric, as in Java, every object has a reference to its *class* (or *class object*), which is a separate object containing code defining the object's dynamic behavior. When methods are invoked on an object, the object's class object is fetched and the code is run.

In contrast with systems like Java that allow the libraries that programs link against to be resolved differently at run time and at compile time, Fabric binds names in code to *specific* immutable versions of the libraries that the code links to. Similarly, once an object is created, the class of that object is fixed to a specific immutable class object. This design means that the code defining the methods of an object will not change over time.

This design choice may seem to hinder software evolution, because software is typically upgraded today by simply replacing the old versions of libraries with the new versions. However, we believe that simply relinking software leads to software instability, because application programmers do not have a chance to evaluate the compatibility of new and old libraries, and the library vendors have only informal

mechanisms for specifying which portions of upgrades are backward compatible. This situation ultimately results in the costly and error prone manual quality assurance processes that accompany software deployment today [BFI14].

Instead of using dynamic linking to evolve software, Fabric leverages the object-oriented techniques of inheritance and subtyping. In Fabric, new versions of classes are stored in separate class objects, so that they can co-exist and interact in the same system. If the new versions of classes are backward-compatible, then they should be subtypes of the old versions; this allows existing software that links against the old version to interact with objects using the new version. If the new version is incompatible with the old version, then it should not be a subtype: this forces software that relies on the old functionality to be updated as well. In short, there are no different versions of “the same” class: if two versions of a class are different, then as far as the Fabric system is concerned, they are different classes.

In Java (and many other languages), the name of a type is synonymous with the identity of the type. This leads to two problems in the context of Fabric. The first is that while different versions of classes are technically different in terms of the way the system works, programmers find it convenient to talk about a class without referring to a specific vendor and version of that class. Second, a global system like Fabric would require a process for allocation of class names, and this would represent a central service that would need to be trusted.

Instead, we build on abstractions that we’ve already discussed: code is data in the system, and are therefore stored as objects. The identity of a class is simply the reference to the object containing its code. Section 2.2.8 shows that reusing the

object abstraction for classes also clarifies the novel information flow mechanisms we have designed.

Rather than require programmers to insert Fabric URLs into their source code, we allow them to use the familiar Java naming scheme. However, programmers are also required to provide a *codebase*: a mapping from fully qualified Java names to Fabric URLs. A codebase is similar to a Java classpath, except that it contains global persistent references and is stored along with the published class. Codebases ensure that names mentioned in the code are always resolved to the same class objects.

To make the process of software evolution clearer, let us look at the upgrade path in the FriendMap example, as illustrated in Figure 2.3. Suppose that Snapp decides to extend their service by adding a `mood` field to the `com.snapp.User` class. They would do so by creating a new class, also called `com.snapp.User`[2]. The new class would explicitly extend the old class, which means that the existing FriendMap code (which the Snapp developers have no control over) can continue to work, even with objects of the new User type.

At a later time, the developers of FriendMap may wish to release a new version that uses the `mood` of the user's friends to color the annotations that are placed on the map. FriendMap cannot assume that all users have the new User type (and thus the mood field). For example, the existing users Alice and Bob would not be upgraded to the new versions. Therefore, FriendMap must decide how it should handle old users.

[2]The Fabric language has a mechanism that allows programmers to use a different name, such as `oldVersion.com.snapp.User`, allowing them to distinguish between different classes with the same name.

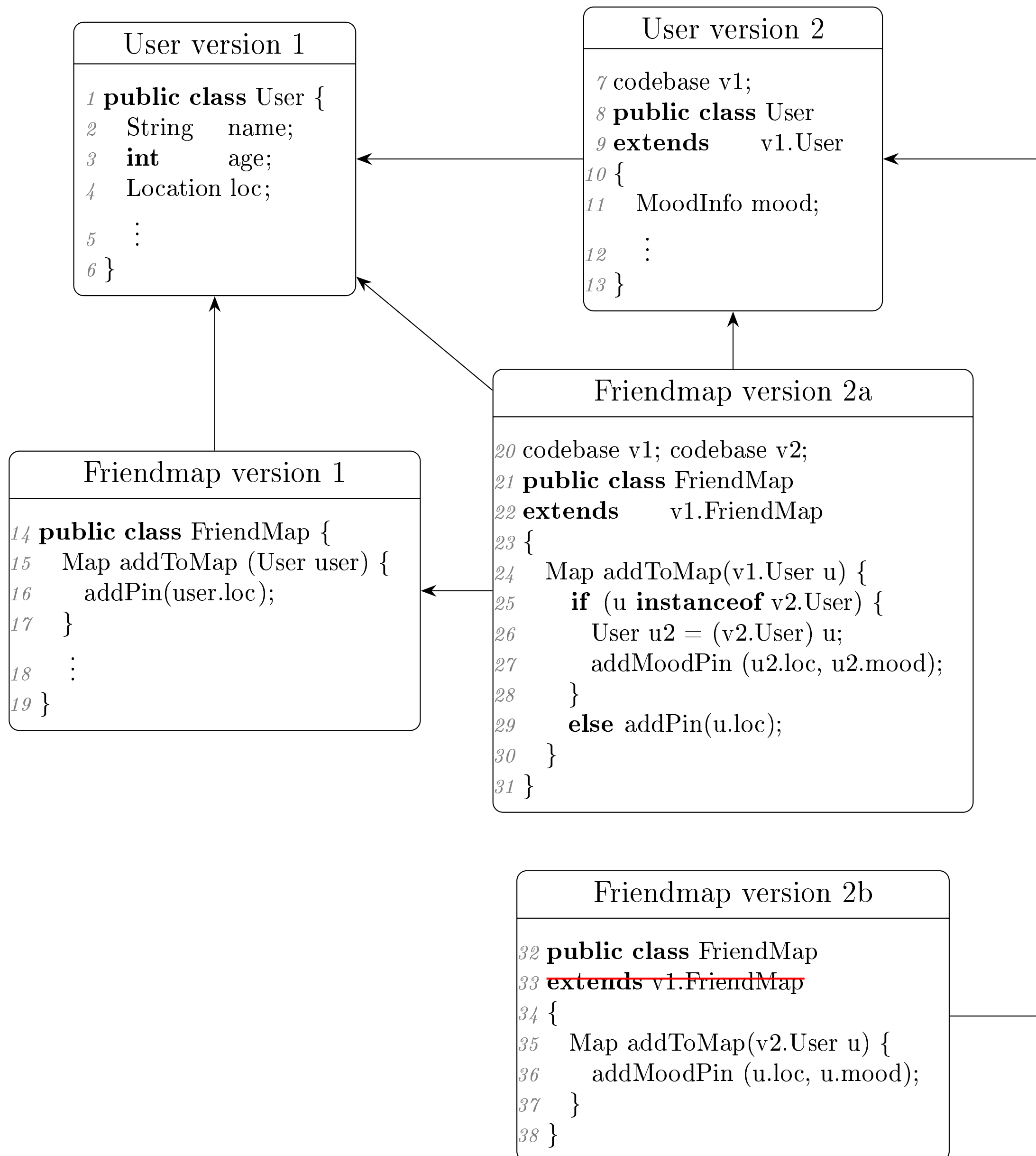

Figure 2.3: Software evolution in the FriendMap example. In the beginning, Snapp publishes the User API, version 1. Next, FriendMap decides to use this API to provide version 1 of its service. Later, Snapp upgrades the User API by adding a `mood` field; to remain backward compatible they make the new `User` class extend the old (line 9). FriendMap can later choose to upgrade their service to make use of the new `mood` field, in a way that is either backward compatible (version 2a) or not (version 2b).

One possibility would be to change the FriendMap interface so that it only accepts objects of the new User type. This choice would prevent the new FriendMap from being a subtype of the old FriendMap. Alternatively, the new FriendMap application could use dynamic downcasting to detect and use the new `mood` field, and therefore remain backward compatible.

The key point is that the FriendMap developers are forced to consider the ramifications of updating their application, and have the ability to document whether new versions are backward compatible by inheriting from the old versions or not.

2.2.4 Principals

The key feature of Fabric is its use of information flow control to protect the confidentiality and integrity of information flowing through it. Fabric uses the concepts of *labels* and *principals* to describe the security requirements on data within the system and to reason about whether programs are safe.

Every entity in Fabric that can trust or be trusted is represented by a *principal*. In the FriendMap example, Alice, Bob, Snapp, FriendMap, and MapServ are all principals.

We use the notation $p \preccurlyeq q$ to indicate that p trusts q completely. We also say that p delegates to q or that q acts for p. *Complete* trust entails the ability to extend further trust, so if $p \preccurlyeq q$ and $q \preccurlyeq r$ then $q \preccurlyeq r$; this coupled with the fact that principals trust themselves make the set of principals under the $\preccurlyeq$ relation into a preorder, which we refer to as the *principal hierarchy*.

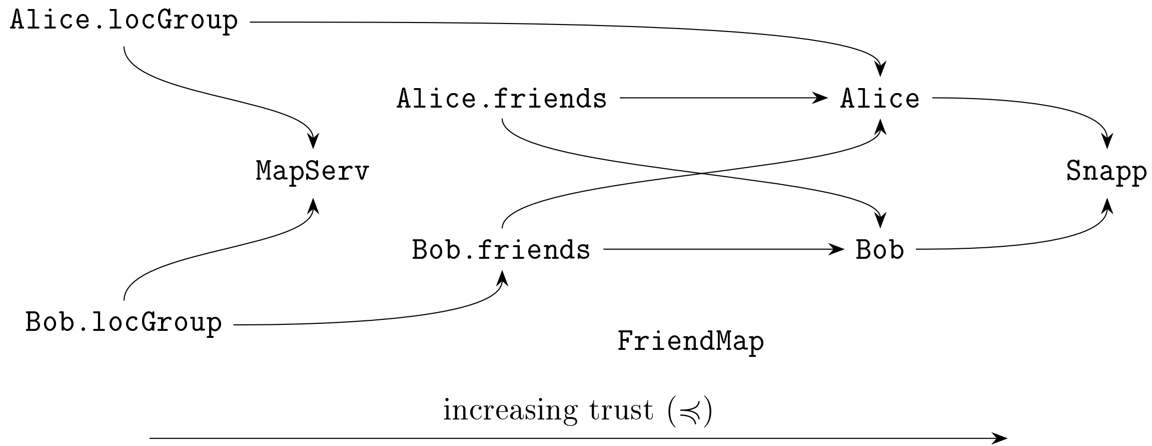

Figure 2.4: Principal hierarchy from the FriendMap example. Note that the application provider `FriendMap` neither trusts nor is trusted by any other principal. Also note that Bob has indicated that he trusts his friends to handle his location information appropriately because `Bob.locGroup` $\preccurlyeq$ `Bob.friends`, but Alice does not, because `Alice.locGrp` $\not\preccurlyeq$ `Alice.friends`.

Complete trust is a very strong notion, but because anyone can create new principals, it can be used to encode partial delegation.[3]

In the FriendMap example, each user principal also has a `friends` principal that can also be used in the specification of policies. As is shown in Figure 2.4, the principal representing Alice's group of friends delegates to both Alice and to each of her friends, but Alice does not delegate to her group of friends; this allows her to retain privileges that she does not extend to them. In addition, the users have `locGrp` principals that delegate to the set of principals that are trusted to enforce the policies on their locations. For example, Alice doesn't trust her friends not to reveal her location, so `Alice.locGrp` $\not\preccurlyeq$ `Alice.friends`.

[3] The Delimited Trust Hierarchy structure described in Chapter 3 encodes partial trust directly, and therefore needs fewer auxiliary principals.

Principals have both an identity and state (their delegation information), and are therefore represented as objects in Fabric. A Principal object's class must implement the `fabric.lang.Principal` interface, which contains the method

```
1 boolean delegatesTo(Principal other);
```

This method defines the Fabric principal hierarchy, which in turn gives meaning to the information flow labels described below.

Using an arbitrary method to define the principal hierarchy gives programmers a natural and flexible way to specify delegation. Principals can access objects from different stores and even make remote calls while deciding whether they should delegate to each other.

The integrity of the delegation relation is critical for ensuring security: untrusted parties should not be able to affect decisions about whether they are trusted. Principals also have different kinds of confidentiality concerns: communication performed while executing `delegatesTo` should not leak information. Moreover, delegation relationships often depend on confidential data whose secrecy must be protected.

Fabric's general-purpose information flow control analysis provides assurance that the implementation of the `delegatesTo` method satisfies these confidentiality and integrity requirements. However, there is an interesting circularity here, because the static and dynamic checks performed by the Fabric system depend in turn on the principal hierarchy.

We investigate this interplay in Chapters 3 and 4. Chapter 3 presents a framework for reasoning about what state can affect delegation and relabeling decisions. Chapter 4 analyzes the communication that occurs while performing delegation queries.

2.2.5 Labels

Every object `o` in Fabric has a *label* that identifies the expected policy for the use of that data. The expression `o.`**label** refers to an object encapsulating the label of `o`.

Labels specify both confidentiality and integrity policies. When thinking about information flow it can be useful to separately consider the constraints imposed by confidentiality concerns and those imposed by integrity concerns. We use the notation $C(\ell)$ to refer to the "confidentiality part" of ℓ, and we use $I(\ell)$ to refer to the "integrity part" of ℓ.

Confidentiality. The confidentiality portion of a label describes how "public" a piece of information is. Secret information should not affect changes to public information, but public information can safely influence private information. Therefore, we say that information with a public label $C(\ell_1)$ can *flow to* a variable with a secret label $C(\ell_2)$, which is written $C(\ell_1) \sqsubseteq C(\ell_2)$.

For example, MapServ wants to disseminate its maps to the public, so the label on those maps (the "`maps`" boxes in Figure 2.2) should be public. On the other hand, the annotated map with Alice's friends' locations ("`annotatedMap`") contains confidential information about her friends' locations. Therefore, it would be acceptable to copy information from `map` to `annotatedMap`, but not the other direction. These constraints are reflected by the following relationships:

$$C(\texttt{map.}\mathbf{label}) \sqsubseteq C(\texttt{annotatedMap.}\mathbf{label}), \text{ but}$$

$$C(\texttt{annotatedMap.}\mathbf{label}) \not\sqsubseteq C(\texttt{map.}\mathbf{label})$$

Integrity. The integrity portion of a label describes how trusted a piece of information has. Low integrity information should not affect changes to high integrity information, but flows in the other direction are permissible. Therefore, we say that information with a high integrity label $I(\ell_1)$ can flow to a variable with a low integrity label $I(\ell_2)$, which is written $I(\ell_1) \sqsubseteq I(\ell_2)$.

In the FriendMap example, the FriendMap application code ("`FriendMap.`**class**") is considered to be low integrity, so it should only be allowed to affect low-integrity information such as `annotatedMap`. It would be unsound for the untrusted application to affect Alice's list of friends ("`Alice.friends`"). These constraints are reflected by the following relationships:

$$I(\texttt{FriendMap.}\mathbf{class}.\mathbf{label}) \sqsubseteq I(\texttt{annotatedMap.}\mathbf{label}), \text{ and}$$
$$I(\texttt{FriendMap.}\mathbf{class}.\mathbf{label}) \not\sqsubseteq I(\texttt{Alice.friends.}\mathbf{label})$$

Enforcement. The flows-to relation $\sqsubseteq$ describes the expected flows of information if the entire system behaves properly. The second important question one can ask about a label ℓ is whether the policy indicated by a label can be violated if a given principal p behaves improperly. This concept is encapsulated by the "is trusted to enforce" relation $p \succcurlyeq \ell$. We have overridden the symbol $\succcurlyeq$ because if $p \succcurlyeq q$ and $q \succcurlyeq \ell$, then $p \succcurlyeq \ell$[4].

In the FriendMap example, if Snapp were untrustworthy, it could release the locations of the users to third parties. This is a risk that the users are willing to

[4]Note that as defined the "is-trusted-to-enforce" relation ($\succcurlyeq$) is not a partial order—it is not even a binary relation over a single set, since it relates the set of principals to the set of labels. In Chapter 3 we extend it to a preorder on the union of labels and principals by suitably defining the relation $\ell_1 \succcurlyeq \ell_2$ (definition 3.13).

take, and they designate this by stating that $\texttt{Snapp} \succcurlyeq \texttt{user.location.}\textbf{label}$. On the other hand, the users are not willing to believe that the FriendMap application provider is trustworthy; thus, for example, $\texttt{FriendMap} \not\succcurlyeq \texttt{Alice.location.}\textbf{label}$.

Downgrading. The third important question one can ask about a label is whether the policy associated with the label can change, and under what conditions. Policies can be relaxed for many reasons: for example, they can change in response to a user action, a change in the state of a computation, or even just the passage of time.

Fabric provides a very flexible mechanism for specifying policy relaxation: the rules for when and how a policy may be relaxed are specified using a Fabric program that explicitly downgrades a policy (by declassifying confidential data or endorsing untrusted data). Of course, the ability to write such a program (or to affect the data it uses to make decisions) must be restricted [CMA17, ZM01].

Authority. These restrictions are naturally expressed by requiring the code and data that affect downgrading to have sufficiently high integrity. If downgrading data from label ℓ to label m could harm principal p, then the code and data used to authorize the downgrading should have an integrity label indicating that it has not been influenced by anyone p doesn't trust.

A downgrade from ℓ to m may harm p if ℓ contains p's confidentiality restriction and m does not, or if m contains p's integrity restrictions and ℓ does not. Therefore, the integrity required to downgrade from ℓ to m depends on the confidentiality and integrity requirements of both ℓ and m, as well as the set of principals who may be harmed if those requirements are dropped.

We summarize the required integrity using a function called *authority*; information labeled k may influence the downgrading of data from ℓ to m if $k \sqsubseteq \mathit{authority}(\ell, m)$. The label $\mathit{authority}(\ell, m)$ has high enough integrity to ensure it is controlled by the principals who may be harmed by the downgrading[5].

In the FriendMap example, Bob may trust Alice to learn his exact location, but may only trust her friends to learn what city he is in. To enable this kind of policy, he could provide a class `CityFinder` that uses his location to determine what city he is in. In order to declassify Bob's location and place it on the map, `CityFinder`'s label must satisfy the following:

$$\texttt{CityFinder.}\mathbf{label} \sqsubseteq \mathit{authority}(\texttt{Bob.locationLabel},\ \texttt{mapLabel})$$

This requirement prevents a class like `CityFinder` from being provided by a principal that Bob doesn't trust.

2.2.6 The Decentralized Label Model

Fabric labels are drawn from the *Decentralized Label Model (DLM)* [Mye99b, ML00]. In the DLM, the only primitive relation is the acts-for relation; labels are composed of principals, and the flows-to, is-trusted-to-enforce, and authority relations are derived from acts-for.

[5] Cecchetti et al. [CMA17] have shown that the confidentiality component of $\mathit{authority}(\ell, m)$ is also important. If the confidentiality of the *authority* label is too high, the downgrading may be susceptible to poaching attacks or confused deputy attacks. Fabric does not currently implement confidentiality requirements on authority, but Chapter 3 discusses the confidentiality and integrity requirements for the *authority* function in more detail.

$$
\begin{array}{rcllll}
o, r, w & \in & Prin & & & \\
\ell & ::= & \{C; I\} & & & \\
C & ::= & o \rightarrow r & | & C_1 \sqcup C_2 & | \quad C_1 \sqcap C_2 \\
I & ::= & o \leftarrow w & | & I_1 \sqcup I_2 & | \quad I_1 \sqcap I_2
\end{array}
$$

Figure 2.5: Syntax of labels in the Decentralized Label Model.

The syntax of DLM labels is shown in Figure 2.5. A DLM label is composed of a *confidentiality component* and an *integrity component*. Components are constructed by taking formal conjunctions ($\sqcup$) and disjunctions ($\sqcap$) of confidentiality or integrity *policies*.

Confidentiality policies consist of two principals: an *owner* and a *reader*, and are written using the syntax $o \rightarrow r$ (where o is the owner and r is the reader). Similarly, integrity policies are denoted $o \rightarrow w$ where o is the owner and w is the writer.

In the FriendMap example, Bob's location may have the confidentiality policy {`Bob` → `Bob.locGrp`}, indicating that Bob controls the policy on the data, but he allows the data to flow to the `locGrp` principal (and implicitly to himself). Similarly, Alice's wall may have the integrity policy {`Alice` ← `Alice.friends`}, indicating that she owns her wall, but allows her friends to post as well.

As mentioned above, the principal hierarchy H in Fabric is generated by the `delegatesTo` method of the `Principal` class. The flows-to, is-trusted-to-enforce, and authority relations are defined in terms of H.

The $\sqsubseteq$ relation is derived from the interpretation of a confidentiality label as an owned set of readers or writers: information may flow from ℓ_1 to ℓ_2 if according to every owner, ℓ_1 has more writers and fewer readers than ℓ_2.

When interpreting DLM labels in the context of Fabric, we had two reasonable choices for defining the is-trusted-to-enforce relation. The first possibility would be to assume that a principal p is trusted to enforce a policy ℓ if and only if p acts for all owners of ℓ; the second choice is that $p \succcurlyeq \ell$ if and only if every owner of ℓ considers p to be both a reader and a writer of ℓ.

To see the difference between the possible definitions, consider a principal hierarchy in which $r \not\succcurlyeq o$. Should $r \succcurlyeq \{o \to r\}$? On one hand, r is specified as a "reader" of the data, so it would make sense to assume that if r is untrustworthy, the data may be leaked. On the other hand, defining $\succcurlyeq$ such that $r \succcurlyeq \{o \to r\}$ gives r the ability to violate o's policy, even though o has not expressed any trust in r.

The interpretation of the DLM that is currently implemented in Fabric defines the relation $\succcurlyeq$ so that $r \succcurlyeq \{o \to r\}$, but this discrepancy reveals an interesting interplay between trust and policy in the decentralized setting. We investigate this interplay in more detail in Chapter 3.

The *authority* function for DLM labels is inspired by the requirements for enforcing robust declassification [ZM01]. We require that the data and code used to influence the declassification of data labeled $\{o \to r\}$ must be writable only by o; this is captured by the requirement that if $\{o \to r\} \sqsubseteq \ell$ and $\{o \to r\} \not\sqsubseteq m$ then $\{\top \leftarrow o\} \sqsubseteq \mathit{authority}(\ell, m)$.

Similarly, the data and code used to influence the endorsement of data to the label $\{o \leftarrow w\}$ requires o's authority. Thus we require that if $\ell \not\sqsubseteq \{o \leftarrow w\}$ and $m \sqsubseteq \{o \leftarrow w\}$ then $\{\top \leftarrow o\} \sqsubseteq \mathit{authority}(\ell, m)$.

2.2.7 Information Flow

The key to Fabric's security is that all application code executed by trustworthy workers is first statically analyzed, and programs that might exhibit unsafe information flows are rejected. Fabric uses an information flow type system based on the Jif language [MZZ+06] to perform this analysis. This section explains standard information flow concepts that are present in Jif; the next section (Section 2.2.8) describes our novel extensions to handle the new features that Fabric provides.

In the Fabric language each variable `x` has a static type τ that includes a label $L(\tau)$ bounding the information that has affected that data. We also refer to this label as $L(\texttt{x})$.

Note that if `x` has a reference type, then $L(\texttt{x})$ describes the information contained in the *reference*, not the information contained in the object itself. The data of an object is contained in its fields, so the information content of the object itself is bounded by the labels of its fields. Since objects are the unit of communication in Fabric, we require that all of the fields of an object must have the same label;[6] this label can be accessed using the special field `x.`**label**.

When type checking a statement, the type system ensures that any information flows caused by a statement are safe according to the $\sqsubseteq$ relation. For example, when checking the statement "`x = y;`", the initial value of `y` influences the final value of `x`. Therefore the type checker must ensure that $L(\texttt{y}) \sqsubseteq L(\texttt{x})$.

[6] We believe that a fairly straightforward transformation can automatically convert an object with heterogeneous fields to a collection of objects with homogeneous fields, but we have not implemented this transformation.

```
1 if (y == true)
2   x = true;
3 else
4   x = false;
```

Figure 2.6: Implicit flow example. This program is equivalent to the program "`x = y;`", so it should only be accepted if $L(\mathtt{y}) \sqsubseteq L(\mathtt{x})$.

Executing a statement causes information to flow from the inputs of the statement to its outputs, but the outputs may also be affected by the context in which the statement is executed. For example, in the statement shown in Figure 2.6, the resulting value of `x` reflects the value of `y` even though the statements `x =` **true** and `x =` **false** do not mention `y` directly. This situation is referred to as an *implicit flow*; *explicit flows* are flows that are clear without considering the context.

Information flow type systems reason about implicit flow by noting that all effects of a statement are all influenced by the fact that the statement is executing. The type system keeps track of a bound on all information that has affected the fact that the statement is executing; this is referred to as the *program counter label* or $\underline{\mathtt{pc}}$. In the example in Figure 2.6, the branch on line 1 influences the $\underline{\mathtt{pc}}$ on lines 2 and 4. These in turn influence the assignments to `x`. Therefore, there must exist some label $\underline{\mathtt{pc}}$ such that $L(\mathtt{y}) \sqsubseteq \underline{\mathtt{pc}}$ and $\underline{\mathtt{pc}} \sqsubseteq L(\mathtt{x})$. Since $\sqsubseteq$ is a partial order, this is true if and only if $L(\mathtt{y}) \sqsubseteq L(\mathtt{x})$. This condition is exactly the same as would be required for the equivalent statement "`x = y;`".

In general, the flows-to relation $\sqsubseteq$ only *approximates* the true information security requirements of an application; sometimes it prevents flows that applications need. Like other systems with information flow control, Fabric allows these flows using

downgrading operations. Declassification is a downgrading operation that reduces confidentiality; endorsement is one that boosts integrity.

Downgrading can be dangerous to security, so the syntax of Fabric makes all declassification and endorsement explicit. Further, downgrading may only happen in contexts that are unaffected by low-integrity information. Specifically, to downgrade from label ℓ to label m, we require $\underline{\texttt{pc}} \sqsubseteq \mathit{authority}(\ell, m)$. This restriction enforces *robust downgrading* [AM11], which prevents the adversary from causing these operations to be misused.

2.2.8 Novel Information Flow Constraints

Fabric expands on standard information flow constraints by adding novel constraints. These constraints ensure that the features it provides for distributed computation are used safely.

Function shipping. The expression `o.f@w(...)` causes control to be transferred to the worker `w`. Once control is transferred, `w` will be responsible for ensuring that `f` is executed faithfully, and that `f` respects all information flow policies. Worker `w` must be trusted to enforce the confidentiality of the arguments that are sent to it, the confidentiality of the $\underline{\texttt{pc}}$, and the integrity of the return value.

The receiver of a remote call must also ensure that the sender is trusted to enforce the integrity of the $\underline{\texttt{pc}}$ and the arguments, and is trusted to enforce the confidentiality policy on the return values. These checks cannot be performed statically, since a remote call may come from any worker at any time. Instead, the compiler generates

dynamic checks that are performed by the receiving worker before the remote call is executed.

Data shipping. When a Fabric reference is dereferenced, the object it refers to may need to be shipped from the store on which it resides. This means that the store may learn that the reference is being dereferenced, which reflects the $\underline{\texttt{pc}}$ at the point of dereference. We refer to this kind of covert channel as a *read channel.*

To ensure that any read channels are safe according to the $\sqsubseteq$ relation, every dereference is statically checked to ensure that the store holding the object is trusted to enforce the confidentiality of the $\underline{\texttt{pc}}$. We extend reference types to include an *access label* $A(\tau)$ which restricts the stores that may hold objects of type τ. An object of type τ can only be stored on a store s if $s \succcurlyeq A(\tau)$, while a reference to an object of type τ can only be dereferenced in a context where $\underline{\texttt{pc}} \sqsubseteq A(\tau)$. Together, these constraints ensure that s can only observe a fetch when $s \succcurlyeq C(\underline{\texttt{pc}})$.

Mobile code. The ability to seamlessly integrate untrusted code with secure data is critical to constructing a secure software ecosystem. The key insight behind our approach is that code is data, and that we can reuse the same mechanisms used to control information flow *in* programs to restrict unsafe information flow *from* programs to the data that they manipulate.

To see how this works, let us consider the FriendMap application. To execute the application, Alice must invoke a method on an object `fm`. This object may come from FriendMap, Snapp, or anyone else, but the *code* for the application resides in a separate object, `fm.`**class**, and has its own label. The label on a class object is also

called the *provider label* of the class; the Fabric language introduces the keyword **provider** as shorthand for **class.label**.

The same rules that prevent an untrusted principal from modifying sensitive data are used to ensure the provenance of code: in this case a worker would not fetch `fm.`**class** from store s unless s is trusted to enforce $I(\texttt{fm.}\mathbf{class.label})$. Moreover, if s is trustworthy, they would prevent any untrusted principal from influencing the class.

Provider-bounded label checking. When type checking code, the compiler uses the provider label to ensure that untrusted code does not affect any trusted data. Our insight is that we can think of code as data that only affects the decision to execute the statements within it. From this perspective, it is clear that requiring $\mathbf{provider} \sqsubseteq \underline{\texttt{pc}}$ exactly captures the right constraints for checking mobile code.

This constraint prevents `friendmap` from providing code that violates policies that `friendmap` is not trusted to enforce. The code in `fm.`**class** cannot modify high integrity information because an updating an object labeled ℓ requires $\underline{\texttt{pc}} \sqsubseteq \ell$. Since $\mathbf{provider} \sqsubseteq \underline{\texttt{pc}}$, we see that $\mathbf{provider} \sqsubseteq \ell$, and therefore if $p \succcurlyeq I(\mathbf{provider})$, we must have $p \succcurlyeq I(\ell)$.

Confidential code. Using the provider label to bound the $\underline{\texttt{pc}}$ provides another feature for no extra effort: confidential code. Businesses may wish to provide programs that contain trade secrets. By publishing such code with a high-confidentiality label, a business can prevent competitors from extracting secrets from the code.

Using information flow control to protect the confidentiality of code is very restrictive. In particular, since the code will affect any outputs it produces, it forces the outputs to be confidential. If the tax preparer wants to send this confidential data back to its users (or to the IRS), the would have to explicitly declassify the output. We believe this is a good thing: it forces the authors to consider the ramifications of releasing that information on the confidentiality of the code, and makes their assumptions explicit.

Provider labels and downgrading. Thinking of the code as data that affects the $\underline{\texttt{pc}}$ also sheds light on the type-checking rules for downgrading. In order to allow a downgrading statement (**endorse** or **declassify**), previous information flow control systems have required code containing downgrading statements to be granted *authority* by the principal whose policy is being relaxed. The assumption in these systems is that authorized code is manually inspected by the principal whose authority is claimed.

We can think of an authority declaration as an assertion of a fact about the class object: namely that it only downgrades data in compliance with (unstated) policies. Because the policies are unstated, they cannot be independently verified by the worker. Instead, the worker relies on the integrity of the object to ensure that it has the expected property.

The integrity constraint that the worker must verify is exactly the same as the additional robustness constraints imposed by provider-bounded label checking. That is, to downgrade data from label ℓ to label m, robustness requires that $\underline{\texttt{pc}} \sqsubseteq$ $\mathit{authority}(\ell, m)$. If the downgrading statement appears in the code of a class C,

then provider-bounded label checking adds the constraint $\texttt{C.}\textbf{provider} \sqsubseteq \underline{\texttt{pc}}$. Together, these requirements imply that $\texttt{C.}\textbf{provider} \sqsubseteq \mathit{authority}(\ell, m)$, which allows the worker to conclude that the code has only been provided by a party that is trusted to enforce $\mathit{authority}(\ell, m)$.

In this way, we can view authority checking as a special case of robust downgrading. In fact, we could remove Jif-style authority declarations entirely and simply rely on provider-bounded label checking and robustness to ensure that code is sufficiently authorized. However, we have decided to retain authority clauses because we feel that they also serve a documentation purpose: they give programmers a reminder that calling a method may cause information to be downgraded.

2.2.9 Transactions

Fabric is designed to model the interaction between software and data storage; this means that presenting a clear consistency model is important. Although much of the modern web uses weak consistency models, we believe that strong consistency is important for constructing correct software that operates over high-integrity data, and for clearly stating the security properties of the system.

Therefore Fabric provides a strong consistency model. Blocks of code can be marked `atomic`, and Fabric will execute those blocks within a transaction. Fabric transactions satisfy the ACID properties [HR83] with a few caveats:

- **Atomicity:** either all of the side effects of a transaction are performed, or none of them are.

- **Consistency:** if the state of the system satisfies the preconditions of the code within an atomic block, then the postconditions of that code will be satisfied. Fabric adds the caveat that only postconditions that relate objects read or written by the transaction are maintained.
- **Isolation:** until a transaction completes successfully, none of the side effects of that transaction are visible to code outside of that transaction.
- **Durability:** once control proceeds to the statement after an atomic block, the effects of the transaction will remain present.

These properties allow programmers to operate under the illusion that their program is the only thing in the world that is executing; this illusion makes it easier for them to reason about the correctness of their programs. It also makes static information flow checking tractable: our analysis relies on the fact that when a program performs a dynamic label comparison, the results of that comparison remain meaningful for the duration of the transaction.

Of course this abstraction is just an illusion, so we must ensure that our implementation provides the same security guarantees that the high-level abstraction promises. These issues are discussed in Section 2.3 and Chapter 4 (particularly Section 4.5).

Because Fabric's correctness is predicated on the Decentralized Security Principle, the ACID properties can only be assumed to hold for high-integrity data. By misbehaving, a principal can violate the ACID properties for some objects, but only if the principal is trusted to enforce the integrity of those objects. For example, a misbehaving principal could violate the durability of the transaction by executing

the transaction protocol and then "forgetting" that an object was committed. Nevertheless, this misbehavior should not violate the ACID properties of any objects that the principal is not trusted to handle.

The abstraction boundary between this high-level view of transactions and the low-level implementation is currently violated in one user-visible way. Fabric makes use of optimistic concurrency control, which means that transactions may be rolled back and replayed. However, it is impossible to rollback people, so output that is user visible may be repeated, or input re-requested. This is a well-known problem with optimistic concurrency control [RG05, HMPJH05]; addressing it is left to future work.

2.2.10 Exceptions and Rollback

Because the Fabric implementation has support for rolling back transactions to handle concurrency failures, we have explored exposing this capability to programmers for handling other kinds of errors.

Fabric inherits *exceptions* from Java, which are a mechanism for handling unexpected circumstances that arise during program execution. In Java, programmers are expected to provide some form of *exception safety*: some guarantee about the state of the program when an exception is thrown. While providing strong exception safety guarantees helps callers of methods to reason about the correctness of their code, implementing strong exception safety is difficult and error-prone. Moreover, because exceptions are usually only used to handle exceptional circumstances, exception correctness is often neglected by programmers and overlooked by testing.

Fabric provides a simple mechanism for implementing strong exception safety. If an unexpected exception is thrown across the boundary of a transaction, then the exception causes the transaction to be rolled back. This means that the offending code caused no side effects on the persistent state. Rollback makes it significantly easier to reason about the correctness of the code that handles the exception.

In Java, exceptions are objects, and this is true in Fabric as well. Exception objects can contain helpful diagnostic information, such as a string describing the exceptional situation. Rolling back the entire state that was modified by a failed transaction would limit the usefulness of exception objects: since the diagnostic information is created as part of the transaction, it would be obliterated as part of the rollback.

Instead, only objects that existed at the start of the transaction are rolled back. Diagnostic information about the failure can be "smuggled" out of the transaction by including it in a newly created exception object. Newly created objects are only observable through the caught exception object, since all other observable references existed before the atomic section and are thus rolled back.

This design still allows exceptions to "change" as they cross the boundary of an atomic section, as illustrated in this example:

```
1 Exception exc = new Exception("message before transaction");
2 atomic {
3   exc.setMessage("inside the transaction");
4   throw exc;
5 } catch (Exception e) {
6   // e.getMessage will return "message before transaction".
7 }
```

Although this result could cause confusion, we think this situation is unlikely to arise in practice, and we find this to be the most logically consistent way to integrate exceptions and transactions.

2.2.11 Interacting With the Outside World

Fabric applications can be written using a mixture of Java, Fabric, and FabIL (the Fabric intermediate language). FabIL is an extension to Java that supports transactions, remote calls, and access control. A key difference between Fabric and FabIL is that FabIL does not enforce information flow security.

More concretely, FabIL supports the following subset of Fabric's features: it provides `atomic` blocks, supports the syntax **new** `C@s(...)` for constructing persistent objects on stores, and gives the ability to make remote calls with the syntax `o.m@w(...)`. Transaction management is performed on Fabric and FabIL objects but not on Java objects, so the effects of failed transactions on Java objects are not rolled back. Additionally, objects created in FabIL are equipped with a programmer-specified access control policy for protecting the object at run time.

FabIL and Java code is considered trusted, and workers only execute trusted code that is stored on their local file system. This design is compatible with the decentralized security principle because the effects of trusted FabIL and Java code are confined to principals that already trust the nodes running the code.

FabIL can be convenient for code whose security properties are not accurately captured by static information-flow analysis, making the labels of the full Fabric language inconvenient. One example is code implementing cryptography, where the

annotation burden of labels is not worth the cost; a second example is the code implementing internals of Fabric, such as its built-in class objects.

2.2.12 Summary of the Fabric Programming Model

The goal of the Fabric programming model is to provide an abstraction that can model most of today's global software ecosystem while providing strong security guarantees. To the extent possible we have applied existing object-oriented software design techniques inherited from Java and existing information flow control techniques inherited from Jif.

The novel features of the Fabric language are required to apply these techniques to a globally distributed system. Mobile code, function and data shipping, and transactions are required to manage the location of data and computation. The is-trusted-to-enforce relation, provider bounded label checking, access label checking, and the information flow constraints imposed on remote calls are necessary techniques to apply existing information flow control techniques to these distributed features.

2.3 The Fabric System

In this section we turn to the design of the Fabric system. The Fabric system is comprised of multiple nodes, each operated by a principal. There is no Fabric "instance", just as there is no "instance" of the web—anyone can participate in the Fabric by starting a new node.

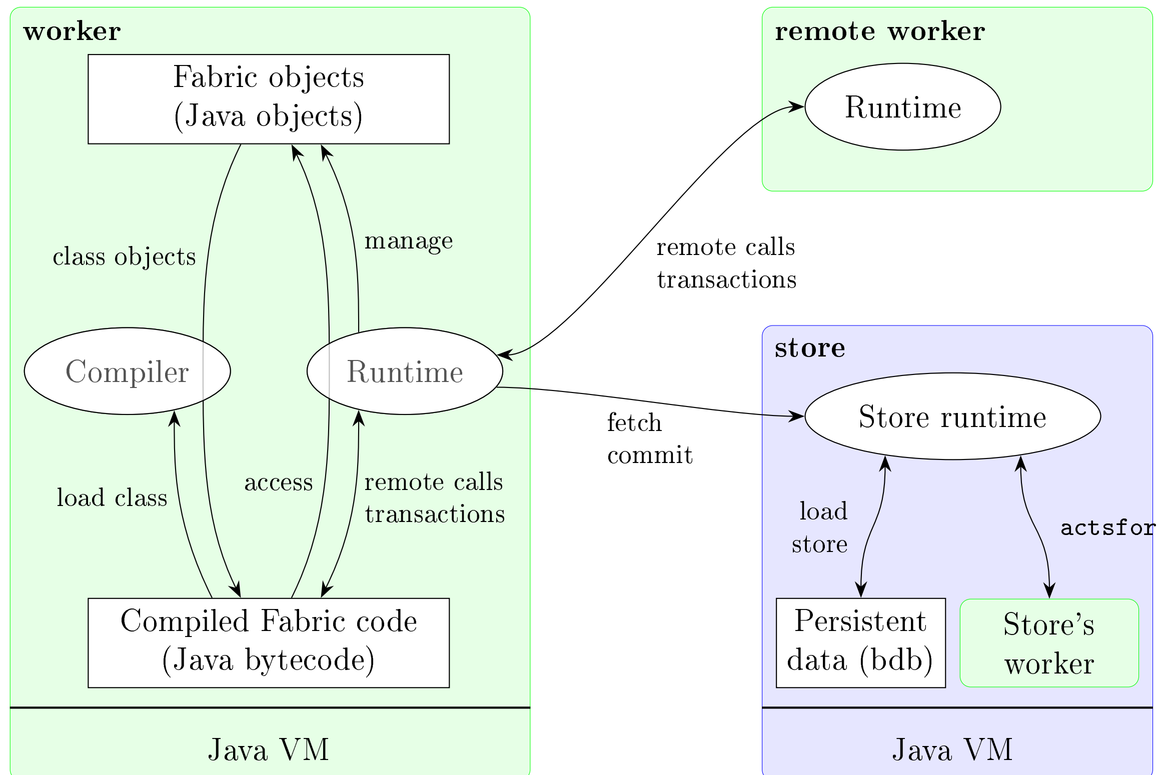

Figure 2.7: Overview of the components of Fabric workers and stores.

Each node acts in one or more roles: it can act as a store, a worker, or a dissemination node. The components of stores and workers are shown in Figure 2.7. Stores are responsible for holding the definitive versions of objects and for performing access control and concurrency control on those objects; Workers are responsible for executing Fabric programs.

Dissemination nodes are untrusted nodes that act as a shared cache for distributing signed and encrypted representations of popular Fabric objects. They model content distribution networks in the internet. They are not central to the design of Fabric—Fabric can be used without any dissemination nodes (in fact, many of our experiments in Section 2.4 are run without the dissemination layer).

2.3.1 Communications Layer

Every Fabric node has a distinguished host name, just as internet hosts have DNS names. The Fabric architecture assumes that given a host name, any Fabric worker can establish an authenticated secure channel to the corresponding Fabric node. In our implementation, this assumption is realized using DNS for name resolution and SSL for communication. The allocation of host names and the distribution and maintenance of the public-key infrastructure are the only centralized roots of trust that Fabric relies on.

Adapting SSL to the communication patterns between Fabric nodes is a software design challenge. SSL provides a simple interface: a connection to a remote host is represented as a socket, and programmers can write and read data from the socket. The SSL abstraction ensures that the data that is written on one host is reliably

presented to the remote host in the same order as it was written. The SSL abstraction also uses a combination of public-key and shared-key cryptography to ensure that the data stream cannot be read or modified by third parties while in transit.

At first glance, the socket abstraction seems like a good primitive for realizing the Fabric architecture. For example, communication between hosts on the web is built entirely on top of the socket interface; and much of the Fabric architecture is based on the design of the web.

The most natural way to use the socket abstraction to implement Fabric is to open a socket for each message, send the message, receive the response, and then close the connection. Unfortunately, SSL sockets are quite expensive to establish, and Fabric nodes typically exchange a large number of messages with the same peers.

To amortize the cost of establishing secure connections, we built a general purpose communication library that provides connection pooling for SSL connections. The library is similar to structured streams [For07]. It provides a socket-like interface; clients of the library can create socket objects (called "subsockets" to distinguish them from SSL sockets) and use them to send and receive data. However, multiple subsockets connected to the same endpoint multiplex a single underlying SSL socket.

The Fabric communication layer (FCL) is designed as a stand-alone library that is not dependent on the rest of Fabric. It provides support for different name resolution protocols as well as different protocols for negotiating secure connections. In addition to allowing the FCL to be a generally useful library, this support has enabled us to experiment with Fabric more easily, and will allow us to easily transition to different name services and authentication mechanisms.

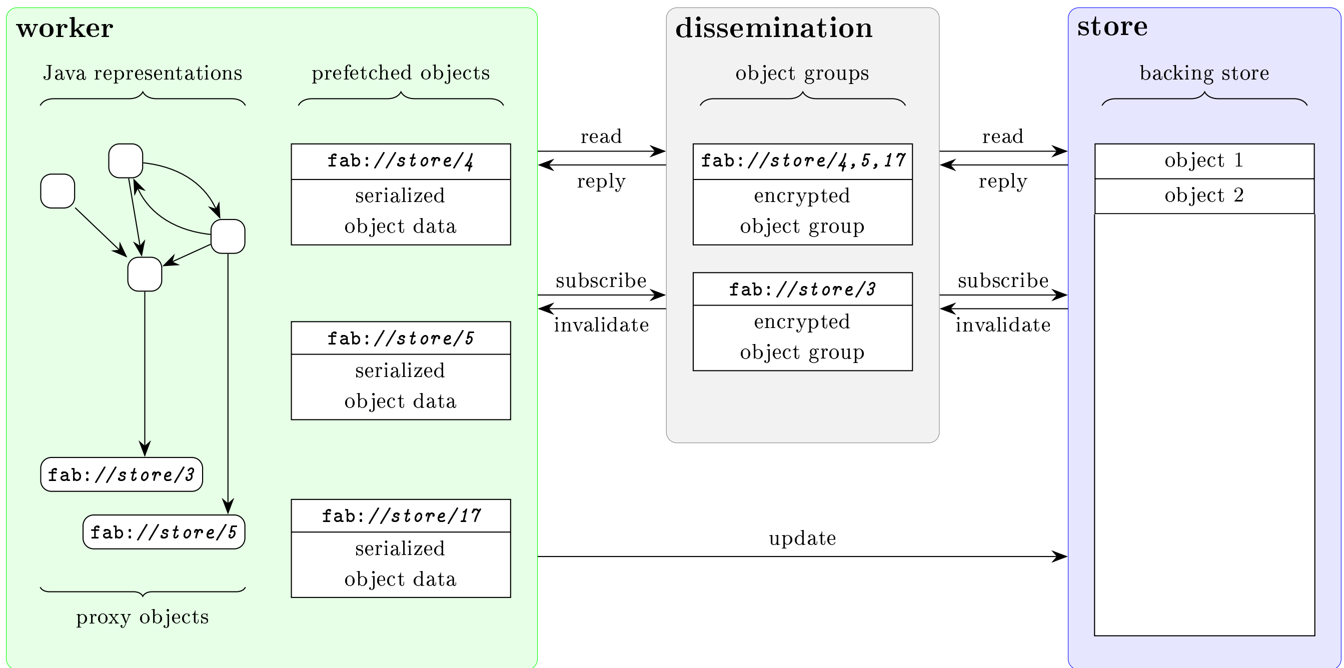

Figure 2.8: Cache hierarchy for Fabric objects.

On top of the communication layer we have built an RPC-style messaging layer for Fabric messages and responses. Workers and stores provide simple remote APIs for transaction management, cryptographic key distribution, and control transfer. These APIs are described in the remainder of this section.

2.3.2 Distributed Objects

The Fabric programming model provides seamless access to persistent, distributed objects. At run time, Fabric workers and stores collaborate to allow programs to read and write objects as they execute.

Whenever a Fabric program dereferences a reference, the worker ensures that there is an in-memory copy of the object for the program to use. It does this through a hierarchy of caches, as shown in Figure 2.8.

Workers represent Fabric objects as Java objects in their local address spaces. A reference to a Fabric object that has not been locally instantiated is represented by a proxy object containing the Fabric object's OID. After the Fabric object is locally instantiated, references to the proxy object are replaced with direct references to the object's Java representation (a process known as swizzling [Wil91]).

Workers instantiate Java representations of Fabric objects by deserializing encoded representations of those objects. These serialized representations are stored in a local cache in the worker.

A worker's local cache is populated by requesting objects from the dissemination layer. The dissemination layer is a collection of untrusted nodes that distribute encrypted and signed copies of objects. The purpose of the dissemination layer is to reduce the load on stores that hold popular objects. Our dissemination layer implementation is built using the FreePastry distributed hash table [RD01b].

Fabric objects are typically much smaller than network messages. To amortize communication overhead, Fabric nodes send groups of objects instead of individual objects.

Object groups on the dissemination layer are encrypted using a shared key corresponding to the label of the objects in the group (the objects in the group share a label). Workers fetch these shared keys directly from the stores, but the overhead of fetching them is amortized because many object groups share the same key.

Object groups are constructed dynamically by stores. Stores group objects based on the object graph and observed access patterns in an effort to improve locality.

If a read request for an object misses in the dissemination layer, then either the dissemination layer or the worker will request the object directly from the store. If the worker contacts the store directly, then the communication layer already encrypts the communication channel, so the objects do not need to be re-encrypted before being sent.

Behind all these layers of caches lies the store, which holds the definitive persistent state of the objects. Fabric does not require any particular implementation of the backing store, but our store implementation uses the Berkeley Database (BDB) [OBS99] as a backing store, and maintains in-memory caches of both unencrypted and encrypted object groups.

The objects in these caches become stale when an object is updated. The dissemination layer can subscribe to the store to be notified when an object group becomes stale; similarly, workers can subscribe to the dissemination layer. Subscriptions are only a best-effort mechanism to improve performance; consistency is maintained by the transactional mechanisms described in Section 2.2.9.

2.3.3 Dynamic Fetch Authorization

If the request for an object `o` gets from a worker w all the way to the store, the store is responsible for ensuring that it is safe to respond to the request. The store must ensure that $w \succcurlyeq C(\texttt{o}.\textbf{label})$.

In the Fabric label model, this check requires testing whether some of the principals mentioned on `o`.**label** delegate to w. As described in Section 2.2.4, Fabric's principals are very flexible: they can execute Fabric computations to decide whether they delegate to other principals. For this purpose, each store has a co-located worker that it can use to run the `delegatesTo` method.

There are no special constraints on what the `delegatesTo` method can do: it is general-purpose Fabric code just like any other. This means that while deciding whether to grant a read request, the store may end up recursively making fetch requests to other stores, or performing remote calls and distributed transactions.

While this may seem like unnecessary complexity, structuring Fabric authorization in this way provides a number of benefits. The first is that the information flow and transactional abstractions that Fabric provides give us assurance that we are implementing the tricky details of authorization correctly: even with a simpler authorization mechanism we would still have to ensure that we are making consistent authorization decisions and that we are doing so in a manner that does not create unsafe information flows.

The other main benefit is that implementing sophisticated authorization has given us insight into real security challenges that existing authorization and authentication systems exhibit. In particular, trying to type-check our `delegatesTo` implementations brought the read channels inherent in distributed authorization to our attention; this motivated the work presented in Chapter 4.

2.3.4 Dynamic Type Checking

Let us return now to the process that takes place when a program running on a worker performs a dereference. We have seen that the worker consults a variety of caches, the store performs access control checks, and eventually a serialized representation of the object makes its way back to the worker. At this point the worker is not quite ready to return the object to the computation that needs it.

Fabric's security relies on the fact that all programs executed by trustworthy workers are well-typed. Type checking relies on an assumption that the objects fetched from other nodes agrees with their static types. Therefore each object fetched must be checked to make sure that it conforms to the expected type.

The most important requirement is that the object's class is well-typed and defines a subtype of the type of the reference. To ensure this property, the worker fetches and compiles the object's class before deserializing the object. The compilation step is when all of the static information flow constraints described in Section 2.2 are checked.

Once the worker type checks the object's class, it can then use the type information from the object's class to deserialize the object's fields. Two additional constraints must be checked:

- Each object has a label object that is used for dynamic information flow checking. Likewise, each class has a label that is used for static information flow control. The object is valid only if these two labels are the same.
- The stores of any references in the object must be trusted to enforce the confidentiality of the access labels on the types of the references.

2.3.5 Concurrency

Objects in Fabric are mutable, so the Fabric system needs some form of concurrency control for programs that access the same objects. As described in Section 2.2.9, Fabric provides a transactional abstraction to programmers. Fabric's runtime design uses a combination of optimistic and pessimistic concurrency control to implement that abstraction.

Fabric uses optimistic concurrency control to coordinate access to objects from different workers. Whenever an object is updated, the store generates a new version number for that object. When objects are shipped between nodes, they are tagged with a version number.

Workers log reads and writes to objects during computation. The first write to an object during a transaction also logs the prior state of the object so that it can be restored in case the transaction aborts.

To reduce logging overhead, the copy of each object at a worker is stamped with a reference to the last transaction that accessed the object. No logging needs to be done for an access if the current transaction matches the stamp.

At the end of the transaction, the workers use a two-phase commit protocol [ML85]. During the prepare phase, each worker contacts the stores of the objects that have been read or written. If any of these objects have been concurrently modified by a different transaction, the pending transaction's effects are rolled back and the transaction is reexecuted with up-to-date objects. Otherwise, the changes are committed to the stores.

Within a single worker, multiple threads may be executing concurrently. Pessimistic concurrency control (locking) prevents these threads from interfering with each other. When a thread reads or writes an object, the runtime system acquires a (local) read or write lock for the object. The thread blocks if the lock would conflict with another lock held by a different thread.

2.3.6 Distributed Transaction Management

Remote calls can take place within Fabric transactions, causing transactions to be distributed across multiple workers. The semantics of distributed transactions should be the same as those of local computations: the whole transaction should be isolated from other Fabric transactions, and its side effects should be committed atomically.

Since Fabric objects are global, a single object may be used by multiple workers in the same transaction. If an object is written within a transaction and subsequently read within the same transaction, the updated value should be observed. Therefore, updates must be propogated as control is transferred within a transaction from one worker to another.

Supporting distributed transactions is challenging: For consistency, workers need to compute on the latest versions of shared objects as they are updated; For performance, workers should be able to locally cache objects that are shared but not updated; For security, updates need to be propogated without leaking confidential information to untrusted workers.

To address these challenges, each Fabric transaction maintains an append-only *writer map* indicating the worker that modified each object most recently. If an

object is updated during a distributed transaction, the node performing the update becomes the object's *writer* and stores the definitive copy of the object for the transaction.

If an updated object already has a writer, the previous writer is notified and relinquishes the role. This notification is not an unsafe covert channel because the $\underline{\texttt{pc}}$ at the write must be lower than the object's label, which the current writer is already trusted to read. The change of object writer is also recorded in the writer map, which is passed through the distributed computation along with control flow.

An update to object o at worker w adds a writer mapping with key $hash(oid, tid, key)$ and value $\{w\}_{\texttt{key}}$, where *oid* is the OID of object o, *tid* is the transaction identifier, and *key* is the object's label's shared encryption key. This mapping permits a worker that has the right to read or write o, and therefore has the encryption key for o, to learn whether there is a corresponding entry in the writer map, and to determine which node is currently the object's writer. Nodes lacking the key cannot exploit the writer mapping because without the key, they cannot compute the hash. Because the transaction id is included in the hash, they also cannot watch for the appearance of the same writer mapping across multiple transactions.

The size of the writer map is a covert channel. To reduce the capacity of this channel, it is padded with dummy entries to make its size a power of 2.

2.3.7 Nested Transactions

Because transactions can be nested, transaction logs are hierarchical. When a local subtransaction commits, its log is merged with the parent transaction log.

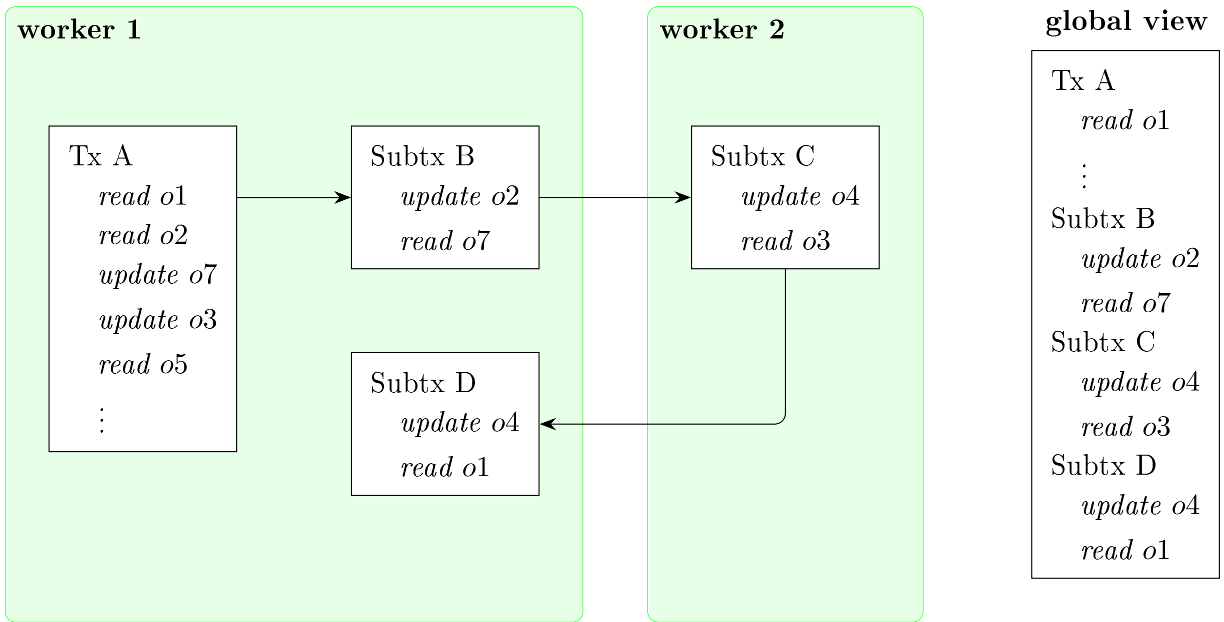

Figure 2.9: Logs of nested distributed transactions

To maintain consistency, transaction management must span multiple workers in the general case. Each worker maintains transaction logs for each top-level transaction it is involved in. These transaction logs must be stored on the workers where the logged actions occurred, because the logs may contain confidential information that other workers may not see.

Figure 2.9 illustrates the log structures that could result in a distributed transaction involving two workers. In the figure, a transaction (A) starts on worker 1, then starts a nested subtransaction (B), then calls code on worker 2, which starts another subtransaction (C) there. That code then calls back to worker 1, starting a third subtransaction (D). Conceptually, all the transaction logs together form a single log that is distributed among the participating workers, as shown on the right-hand side.

When D commits, its log is conceptually merged with the log of C, though no data is actually sent. When C commits, its log, including the log of D, is conceptually merged with that of B. In actuality, this causes the log of D to be merged with that of B, but the log for C remains on worker 2. When the top-level transaction commits, workers 1 and 2 communicate with the stores that they have interacted with, using their respective parts of the logs.

2.3.8 Hierarchical Commits

A transaction may span worker nodes that do not trust each other, creating both integrity and confidentiality concerns. An untrusted node cannot be relied to commit its part of a transaction correctly. More subtly, an insecure commit protocol might cause an untrusted node to learn information it should not. For instance, simply learning the identities of other nodes that participated in a transaction may reveal sensitive information. Fabric's *hierarchical two-phase commit* protocol addresses these problems.

The commit protocol is a hierarchical version of the usual two-phase commit protocol. The first phase begins with the worker that started the top-level transaction. It initiates the commit by contacting all the stores for whose objects it is the current writer in the writer map, and all the other workers to which it has issued remote calls. These other workers then recursively do the same, so the first phase of the protocol constructs a *commit tree*, a spanning tree of the transaction's *remote-call graph*. This process allows all the stores involved in a transaction to be informed

about the transaction commit, without relying on untrusted workers to choose which nodes to contact and without revealing to workers which other nodes are involved in the transaction lower down in the commit tree.

The two-phase commit protocol then proceeds as usual, except that messages are passed up and down the commit tree rather than directly between a single coordinator and the stores. The first phase of the protocol not only constructs the commit tree but also causes each participating store to validate the transaction by checking permissions and comparing version numbers. Each store that successfully validates the transaction prepares to commit it. The second phase of the protocol informs all participants whether the prepared transaction should be committed or aborted.

Of course, a worker in this tree could be compromised and fail to correctly carry out the protocol, causing some stores to be updated in a way that is inconsistent with other stores. However, this worker could already have introduced this inconsistency by simply failing to update some objects or by failing to issue some remote method calls. The untrusted worker's power over the transaction is merely to prevent it from happening at all, which is not a security violation.

Once a transaction is prepared in the first phase of the two-phase commit, it is important for the availability of the objects involved that the transaction is committed quickly. The transaction coordinator should remain available, and if it fails after the first phase of the transaction, it must recover rapidly. An unavailable transaction coordinator could become an availability problem for Fabric, and the availability of the coordinator is therefore a trust assumption. To prevent denial-of-service attacks, prepared transactions are timed out and aborted if the coordinator is unresponsive.

This failure is considered a violation of trust, but in keeping with the security principles of Fabric, the failing coordinator can only affect the consistency of objects whose integrity it is trusted to enforce. This design weakens Fabric's consistency guarantees in a circumscribed way, in exchange for stronger availability guarantees.

2.4 Evaluation

Our goal in designing Fabric was to provide a programming model that is expressive enough to model today's complex software ecosystem, while providing security guarantees in a principled way. To evaluate Fabric, we have built an implementation of the Fabric system as well as a number of example applications. We have also ported several existing applications to the Fabric language.

We use these applications to evaluate Fabric along several dimensions:

- We evaluate the expressiveness of Fabric's programming model by modeling systems with complex trust relationships and communication patterns. We find that the programming model is expressive.
- We evaluate the effort required to build Fabric programs by comparing Fabric and non-Fabric versions of the same programs. We find that Fabric's built-in support for data and function shipping simplifies many applications, but that the annotation burden for information flow labels is high. Part of this burden is due to the complexity of the information flow policies themselves, but we also discuss approaches for reducing some of the redundancy required by our current type system.

- We show that our Fabric system implementation is efficient by measuring the performance of Fabric implementations of industry-standard benchmarks. We show that Fabric performs comparably to other distributed systems.

2.4.1 System Implementation

The Fabric compiler is implemented as a source-to-source translation from the Fabric language to Java. We first translate to an intermediate language FabIL, which contains support for handling distributed Fabric objects and transactions but does not perform information flow analysis.

The Fabric and FabIL compilers are implemented using the Polyglot extensible compiler framework [NCM03]. The FabIL compiler extends the Polyglot Java-to-Java compiler (J1), while the Fabric compiler extends the Jif compiler. The Fabric compiler contains 14k lines of Java code; the FabIL compiler contains 11k lines of Java code, and there are an additional 3.5k lines of code that are shared between the two.

The Fabric runtime system is divided into the store, worker, and dissemination node implementations. The store comprises 3.5k lines of Java code; the worker contains 6k lines of Java code; and the dissemination layer comprises 2.2k lines of code. There are also 13k lines of shared code between them, such as the communications layer and common utilities.

We have ported a number of useful libraries to FabIL and Fabric. We have partially ported the GNU Classpath implementation of the `java.util` package [Fou], this comprises almost 10k lines of FabIL code.

We have also ported the Servlets with Information Flow (SIF) library [CVM07] from Jif to Fabric. This library provides an API for building web applications in Jif; We make use of this library to provide a front-end to some of our example applications. The ported SIF library contains 6.5k lines of Java, FabIL and Fabric code.

Altogether, the Fabric compiler, runtime system, and libraries contain nearly 100k lines of code.

2.4.2 FriendMap Application

We have implemented the FriendMap application that we've used as a running example in this chapter. Our implementation faithfully models the separate roles of the users Alice and Bob, the Snapp service, the MapServ service, and the FriendMap application.

Our implementation of the FriendMap application includes most of the application features we have discussed in this chapter. There are separate codebases provided by Snapp (250 LOC), MapServ (630 LOC), and FriendMap (770 LOC). In addition, we have created a "version 2" of the Snapp `User` class that includes user moods, and subsequently extended the FriendMap code to make use of the extended users. The Snapp extension required an additional 70 lines of Fabric code, while the extended FriendMap application required 80 lines of code. The entire application was developed by 2 developers over a period of 2 months.

As part of the implementation, we had to develop 500 lines of general-purpose utility classes that needed to be trusted by everybody. These include basic classes

such as a Principal that delegates to a set of other principals and a principal that delegates to a single unique principal. These could be folded back into the Fabric standard library.

Although we were able to replicate many facets of today's software ecosystem in the FriendMap application, we found the type annotations required for information flow analysis to be rather burdensome. Access label checking and provider label checking increase the number of constraints that must be satisfied by the program at nearly every program point; this requires annotations on every method to declare that the program must satisfy those constraints in order to call these methods.

Figure 2.10 shows a particularly complex example. The safety of the `createMap` requires a complicated set of relationships between labels and principals, and in order to statically check the method, these requirements must be documented in the type.

While implementing the FriendMap application, we realized that many of these annotations are duplicated. The FriendMap application requires several relationships between the different principals and labels that it uses. Fabric's type system requires every method to explicitly declare pre- and post-conditions. Since many of these invariants are required for several methods, there is a great deal of duplication.

In the 770 lines of code of the FriendMap codebase, roughly 320 of them are type annotations (or documentation of the type annotations). By removing duplication, we could reduce this to roughly 80 lines of annotation (thus reducing the codebase to about 530 LOC).

This observation has inspired the design of first-class proof objects, which are explored further in Chapter 4. A constraint class would bundle together the static

```
1  public
2  MapImage[l,{*→s}] {*l}
3  createMap
4  (
5    final label l, final Store s,
6
7    final User[owner,network] user,
8    final principal owner, final principal network,
9
10   final label friend_access_bound
11 )
12 throws ( NullPointerException{*provider; user} )
13 where
14   // user fetch depends on:
15   {caller_pc; *provider; *MapServer.provider} <= {*→network}, // code and caller pc
16   {user}                                      <= {*→network}, // user reference
17   {friend_access_bound; network; l}           <= {*→network}, // policies
18
19   // second fetch dependent on copied map, which depends on
20   {s}          <= {*→network}, // the map service
21   {*←service} <= {*→network}, // the store at which the result is created
22   {*l}         <= {*→network}, // the intermediate state of the map
23
24   // information that flows to the map service:
25   {caller_pc; *provider; *MapServer.provider} <= {*→service.store$}, // code and caller pc
26   {user; {_→; *←network,owner}}               <= {*→service.store$}, // user and user's contents
27   {friend_access_bound; network; l}           <= {*→service.store$}, // fetch policy
28
29   // information that flows to the resulting map:
30   {caller_pc; *provider; *MapServer.provider} <= {*l}, // code and caller pc
31   {user; {_→; *←network,owner}}               <= {*l}, // user and user's contents
32   {s; l}                                      <= {*l}, // store and label of result
33   {friend_access_bound; network; l}           <= {*l}, // friends policies
34   {*←service}                                 <= {*l}, // fetched map
35
36   // fetch of user's friends:
37   {caller_pc; *provider; *MapServer.provider} <= {*friend_access_bound}, // code and caller pc
38   {user; {_→; *←network,owner}}               <= {*friend_access_bound}, // user and user's contents
39   {friend_access_bound; network; l}           <= {*friend_access_bound}, // policies
40   {s}                                         <= {*friend_access_bound}, // the store at which the result is created
41   {*l}                                        <= {*friend_access_bound}, // the intermediate state of the map
42
43   // intermediate objects created at local store
44   l <= {*→worker$}, {*←worker$} <= l,  // worker ≽ l
45
46   // l objects created at s
47   {*←s} <= l, l <= {*→s}, {*→s} <= {*→s}, // result created at s
48
49   // invariants
50   user.p equiv (network,owner)
51 {
52   ...
53 }
```

Figure 2.10: A portion of the FriendMap application implementation. These annotations reflect the real flow of information through the `createMap` method. They reflect the complexity of the necessary trust relationship: the method is only safe if these constraints hold.

requirements shared by many methods; non-null objects of this class would serve as proofs of these constraints. Implementation of first-class proofs in Fabric is left for future work.

2.4.3 Course Management System

To evaluate the performance impacts of the Fabric system model on a more complete example, we ported a portion of Cornell's Course Management System (CMS) to FabIL. CMS is a 54k line J2EE web application written using EJB 2.0 [BMH06], backed by an Oracle database. It has been used for course management at Cornell University since 2005; at present, it is used by more than 40 courses and more than 2000 students.

Our experience showed that Fabric's high-level primitives for security, transactions, persistent storage, and mobile code yielded a simpler and faster implementation.

Implementation. The production version of CMS uses the model/view/controller design pattern; the model is implemented with Enterprise JavaBeans using Bean-Managed Persistence. For performance, hand-written SQL queries are used to implement lookup and update methods, while generated code manages object caches and database connections. The model contains 35 Bean classes encapsulating students, assignments, courses, and other abstractions. The view is implemented using Java Server Pages, and the controller is implemented as a Java Servlet object.

We ported CMS to FabIL in two phases. First, we replaced the Enterprise JavaBean infrastructure with a simple, non-persistent Java implementation based on the Collections API. We ported the entire data schema and partially implemented the query functionality of the model, focusing on the key application features. Of the 35 Bean classes, 5 have been fully ported. Replacing complex queries with object-oriented code significantly simplified the model: the five fully ported classes were reduced from 3100 lines of code to 740 lines, while keeping the view and controller mostly unchanged. This intermediate version, which we refer to as the Java implementation, took one developer a month to complete and contains 23k lines of code.

Although significantly simpler, the Java implementation does not have support for concurrency control, distributed operation, or persistence. It serves as a baseline for evaluating the performance and complexity of the FabIL implementation.

Porting the Java implementation to FabIL required only superficial changes, such as replacing references to the Java Collections Framework with references to the corresponding Fabric classes, and adding label and store annotations. The FabIL version adds fewer than 50 lines of code to the Java implementation, and differs in fewer than 400 lines. The port was done in less than two weeks by an undergraduate initially unfamiliar with Fabric. These results suggest that porting web applications to Fabric is not difficult and results in shorter, simpler code.

A complete port of CMS to Fabric with fine-grained labels would have the benefit of federated, secure sharing of CMS data across different administrative domains, such as different universities. It would also permit secure access to CMS data from other applications. We leave this to future work.

	Page Latency (ms)		
Implementation	Course	Students	Update
EJB	305	485	473
Hilda	432	309	431
FabIL	35	91	191
FabIL/memory	35	57	87
Java	19	21	21

Table 2.1: CMS page load times (ms) under continuous load.

Performance. The performance of Fabric was evaluated by comparing five different implementations of CMS: the production CMS system based on EJB 2.0, the in-memory Java implementation (a best case), the FabIL implementation, the FabIL implementation running with an in-memory store (FabIL/memory), and a fifth implementation developed earlier using the Hilda language [YSRG06]. Comparing against the Hilda implementation is useful because it is the best-performing prior version of CMS. The performance of each of these systems was measured on some representative user actions on a course containing 55 students: viewing the course overview page, viewing information about all students enrolled in the course, and updating the final grades for all students in the course. All three of these actions are both compute- and data-intensive.

All Fabric and Java results were acquired with the app server on a 2.6GHz single-core Intel Pentium 4 machine with 2GB RAM. The Hilda and EJB results were acquired on slightly better hardware: the Hilda machine had the same CPU and 4GB of memory; EJB results were acquired on the production configuration, a 3GHz dual-core Intel Xeon with 8GB RAM.

Table 2.1 shows the median time to perform three user actions under continuous load, for each of the measured systems. The first three measurements in Table 2.1 show that the Fabric implementation of CMS runs faster than the previous implementations of CMS. The comparison between the Java and nonpersistent FabIL implementations illustrates that much of the run-time overhead of Fabric comes from transaction management and from communication with the remote store.

2.4.4 OO7 Benchmark

To evaluate the overhead of Fabric computation at the worker when compared to ordinary computation on nonpersistent objects, and to understand the effectiveness of object caching at both the store and the worker, we used the OO7 object-oriented database benchmark [CDN93]. We measured the performance of a read-only (T1) traversal on an OO7 small database, which contains 153k objects totaling 24Mb.

The results of these measurements are summarized in Table 2.2. Performance was measured in three configurations: (1) cold, (2) warm, with stores caching object groups, and (3) hot, with both the store and worker caches prepopulated.

The results show that caching is effective at both the worker and the store. However the plain in-memory Java implementation of OO7 runs in 66ms, which is about 10 times faster than the worker-side part of the hot traversal. Because Fabric is designed for computing on persistent data, this is an acceptable overhead for many, though not all, applications. For computations that require lower overhead, Fabric applications can always incorporate ordinary Java code, though that code must implement its own failure recovery.

Cache state	Traversal time					
	Total (ms)	App	Tx	Log	Fetch	Store
Cold	9153	10%	2%	12%	74%	2%
Warm	6043	27%	3%	6%	61%	3%
Hot	840	46%	14%	24%	0%	17%

Table 2.2: Breakdown of the running time for the T1 traversal on a small OO7 database. "App" gives the time spent executing application code; "Tx" gives the time spent on local transaction management. The "Log" column gives the profiling overhead. "Fetch" is time spent communicating with the store, and "Store" is time spent by the store to respond to those requests.

2.4.5 Other Applications

We have also built other applications to evaluate the performance and flexibility of the Fabric programming model. Although less complex than the FriendMap example presented above, they show that Fabric can be used for a variety of applications with complex information flow constraints.

Multiuser Calendar

We ported the multiuser calendar application originally written for SIF [CVM07] to Fabric. This application allows users to create shared events and to control the visibility of their events using information flow policies.

The application is structured as a standard web application server running on a Fabric worker node. Persistent data is kept on one or more storage nodes, but the worker and the stores do not necessarily trust each other. The design allows users to maintain their calendar events on a store they trust, and application servers can

run on any worker the user trusts. This design is in contrast to current distributed calendars where all events are maintained on a single globally trusted domain.

The SIF Framework is written in Jif, so the existing code was already annotated with information flow labels. Porting SIF and the Calendar application to Fabric required replacing transaction and persistence mechanisms with Fabric's built-in primitives. Static checks performed by the Fabric compiler force the insertion of additional dynamic label and principal tests, to ensure that persistent object creation and remote calls are secure.

As with our FabIL port of CMS, by replacing special-purpose code for converting Java objects to a persistent SQL-backed representation, we were able to simplify the code. The SIF implementation of the Calendar application comprised roughly 1800 lines of code; the Fabric implementation removed roughly 400 of these. The Fabric port of the 4k line SIF library itself also had about 4k lines of code.

Bidding Agent

In this example application, a user supplies an agent to choose between two ticket offers made by different airlines. The choice may depend on factors confidential to the user, such as preferred price or expected service level. Airlines, in turn, supply agents that compete for the best offer to provide to the user, while maximizing profit. This example is about 570 lines of code.

Four parties participate: a trusted broker, two airlines, and the user. They are represented by Fabric principals `Broker`, `AirlineA`, `AirlineB`, and `User`. Principal `Broker` is trusted by all of the others: `Broker`$\succcurlyeq$`AirlineA`, `Broker`$\succcurlyeq$`AirlineB`, and

`Broker`$\succcurlyeq$`User`; no other trust relationships are assumed. Every principal is associated with a Fabric store.

This example shows that the users and airlines are both able to provide untrusted code to the trusted broker. The guarantees provided by Fabric's type checking allow the broker to execute this code while respecting the information flow policies specified by all parties.

2.5 Related Work

Fabric provides a higher-level abstraction for programming distributed systems. Because it aims to help with many different issues, including persistence, consistency, security, and distributed computation, it overlaps with many systems that address a subset of these issues. However, none of these prior systems addresses all the issues tackled by Fabric.

Fabric fits into a substantial history of efforts to integrate information flow control into practical language-level programming abstractions; prior systems include SPARK/Ada [Bar03], Jif and Jif/split [MZZ+06, ZZNM02, ZCMZ03], FlowCaml [Sim03], Aura [JVM+08], Swift [CLM+07], LIO [SRMM11], Jeeves [YYSL12], Paragon [BvDS13], and IFC [EJM+14]. These previous systems are either not distributed, or provide only limited control over distributed computation. Many of the contributions of Fabric arise from fully extending information flow methods into the realm of distributed computation over persistent data, where we have encountered new side channels and uncovered new connections between integrity and authority.

Jif/split [ZZNM02], SIF [CVM07], and Swift [CLM+07] are prior distributed systems with mutually distrusting nodes, but with more limited goals than Fabric. While these prior systems use language-based security to enforce strong confidentiality and integrity, they do not allow new nodes to join the system, and they do not support consistent, distributed computations over shared persistent data.

In contrast to language-level information flow tracking, systems such as Asbestos [EKV+05], HiStar [ZBWKM06], Flume [KYB+07], Laminar [RPB+09], and TaintDroid [EGC+10] track information flow dynamically at the operating system or virtual machine level. These systems do not require programming language support, but are only able to track information flows at a coarse granularity.

DStar [ZBWM08] applies OS-based DIFC in a distributed setting. Like Fabric, DStar is a decentralized system that allows new nodes to join, but unlike Fabric, it does not require certificate authorities. As with other OS-based DIFC systems, DStar does not require language support, but controls information flow more coarsely. DStar does not support consistent distributed computations, data shipping, or mobile code. It also has no notion of code integrity or secrecy.

Like Fabric, Aeolus [CPS+12] is a platform for building distributed applications that support information flow control. Aeolus's information flow tracking is similar to that provided by OS-based DIFC systems, but Aeolus also provides high-level programming language abstractions that give programmers fine-grained information flow control. Unlike Fabric, Aeolus is a closed system with a centralized authorization service and without support for mobile code.

OceanStore [REG+03] shares the goal with Fabric of a federated, distributed object store, but focuses more on storage than on computation. It provides consistency only at the granularity of single objects, and does not help with consistent distributed computation. OceanStore focuses on achieving durability via replication. Fabric stores could be replicated but currently are not. Unlike OceanStore, Fabric provides a principled model for declaring and enforcing strong security properties in the presence of distrusted workers and stores.

Some previous distributed storage systems have used transactions to implement strong consistency guarantees, including Mneme [Mos90], Thor [LAC+96] and Sinfonia [AMS+07]. Cache management in Fabric is inspired by that in Thor [CAL97]. Fabric is also related to other systems that provide transparent access to persistent objects, such as ObjectStore [LLOW91] and GemStone [BOS91]. These prior systems do not focus on security enforcement in the presence of distrusted nodes, and do not support consistent computations spanning multiple compute nodes.

Distributed systems with support for consistency, such as Argus [Lis85] and Avalon [HW87], usually have not offered single-system view of persistent data, and none enforce information security. Emerald [BHJL86] gives a single-system view of a universe of objects while exposing location and mobility, but does not support transactions, data shipping or secure federation. InterWeave [CDP+00] is a persistent distributed shared memory system that synthesizes data- and function-shipping similarly to Fabric, and allows multiple remote calls to be bound within a transaction while remaining atomic and isolated with respect to other transactions. However, it does not appear to be feasible to build a system like Fabric on top of InterWeave,

because InterWeave has no support for information security and its mechanisms for persistence and concurrency control operate at the granularity of pages. The work of Shrira et al. [STT08] on exo-leases supports nested optimistic transactions in a client–server system with disconnected, multi-client transactions, but does not consider information security. MapJAX [MCCL07] provides an abstraction for sharing data structures between the client and server in web applications, but does not consider security. J-Orchestra [TS09] creates distributed Java programs by partitioning programs among assigned network locations. Standard Java synchronization operations are emulated across multiple hosts, but neither security nor persistence is considered. Other recent language-based abstractions for distributed computing such as X10 [CDE+07] and Live Objects [OBDA08] also raise the abstraction level of distributed computing but do not support persistence or information-flow security.

Several distributed storage systems including PAST [RD01a], Shark [AFM05], CFS [DKK+01], and Boxwood [MMN+04] use distributed data structures to provide scalable file systems, but offer weak consistency and security guarantees for distributed computation.

IFDB [SL13] provides a SQL-based interface to a single persistent database while tracking information flow fully dynamically. It is not a federated system like Fabric, nor does it provide type-level integration in the language.

Many previous languages [JL78, MWC10, Mil06, MSL+08] have explored integrating abstractions for authorization and access control into the programming model. However, these languages do not integrate reasoning about information flow and rely on the programmer to use these abstractions appropriately to enforce security.

UrFlow [Chl10] enforces information flow control in web applications with policies expressed by SQL queries. UrFlow prevents implicit flows in application code, but not those introduced by the queries themselves.

Hails [GLS+12] dynamically enforces information flow control for Haskell web applications. Like Fabric applications, Hails web apps compose mutually untrustworthy components that may access persistent data. However, Hails components implement a model–view–controller design pattern and may not invoke each other directly, though multiple view–controllers may share the same model. Hails does not prevent read channels, but does prevent termination and timing channels [SRB+12].

Cross-origin resource sharing (CORS) [CJD+18] extends the same-origin policy to allow web sites to specify domains that may load resources from other origins. A browser implementing the CORS API performs a "preflight request" to determine what restrictions apply to a resource before fetching the resource. The CORS API does not protect against read channels: preflight requests may leak information from the requesting page.

Fabric's support for secure mobile code can be compared to proof-carrying code (PCC) [Nec97], a general mechanism for transmitting proofs about code to code consumers. Fabric does not contain a general proof checker; clients check code they receive using the Fabric type system. The Fabric approach is analogous to the bytecode verifier used by Java [LY99], which similarly type-checks JVM bytecode.

Various attempts have been made to strengthen isolation guarantees for JavaScript. Chugh et al. [CMJL09] dynamically check loaded code against statically identified residual information-flow requirements. Conscript [ML10] applies aspects to

JavaScript primitives, isolating loaded scripts in useful ways. Caja [MSL+08] provides isolation in web mashups by using capabilities to protect access to resources at a fine granularity. Secure information flow can be enforced by checking capabilities at statically predetermined locations [BS11], assuming a static analysis of information flow. Hedin and Sabelfeld [HBBS14] dynamically enforce secure information flow within a JavaScript DOM tree. Securing mobile code in Fabric has similar challenges to securing JavaScript, but Fabric's mobile code may express more general computations, including creating and accessing persistent data, and may communicate with arbitrary nodes.

System extensibility and evolution has been explored in many contexts. To our knowledge, Fabric's mobile code support is the first to address the information security of the assembly and evolution of components in a general distributed setting.

SPIN [GB01] is an extensible operating system that allows core kernel functionality to be dynamically specialized by modules written in Modula-3. Like Fabric, SPIN leverages language-level features—such as interfaces and type safety—to provide isolation for untrusted system modules. Unlike Fabric, SPIN uses namespace isolation to control access to system resources: capabilities are implemented as references to system resources, with a type capturing access privileges. In contrast, name resolution in Fabric is orthogonal to security, and the security implications of linking with low integrity code are captured by the type system.

Prior work on expressive module systems explored several approaches to component reuse and evolution. Unit [FF98] and Knit [RFS+00] are component definition and linking languages that enable programmatic assembly of components.

Composite units are assembled out of smaller ones, and some architectural properties are checked, such as type consistency (in [FF98]) or user-defined constraints (in [RFS+00]). These systems provide more flexible control of namespaces, but they do not address the security of the produced code.

Codebases have similarities to the classpath entries in JAR files [Ora99]. These references are neither versioned nor immutable, so the meaning of Java classes can change over time. JAR files allow packages to be *sealed*, to control who can insert classes into them. Sealing is orthogonal to our consistency requirements: it does not ensure that classes are named consistently nor that the meaning of code is fixed.

2.6 Summary

We have explored the design and implementation of Fabric, a new, general platform for secure sharing of information and computation resources. Fabric provides a high-level abstraction for secure, consistent, distributed general-purpose computations on persistent, distributed information. Persistent information is conveniently presented as language-level objects connected by pointers. Mobile code can be dynamically downloaded and used securely by applications, subject to policies for confidentiality and integrity. Fabric exposes security assumptions and policies explicitly and declaratively. It flexibly supports a range of computation styles moving code to data or data to code. Results from implementing complex, realistic systems in Fabric, such as FriendMap, CMS, and SIF, suggest it has the expressive power and performance to be useful in practice.

Fabric's security model is based on information flow control, which makes it inherently compositional, even in a decentralized system. Fabric's provider-bounded label checking preserves this compositional security assurance even in the presence of mobile code. As a result, code and data from different, partially trusted sources can be combined while providing relatively strong security assurance.

Fabric embodies several important technical contributions. Fabric extends the Jif programming language with new features for distributed programming, while showing how to integrate those features with secure information flow. This integration requires new implementation mechanisms such as writer maps, distributed transaction logging, and hierarchical two-phase commit. The mobile-code architecture is an interesting and useful component in its own right; provider-bounded verification should be a useful technique for securing other mobile-code systems.

Fabric succeeds in offering both a simple, general abstraction for building secure systems and an implementation that can be used to build real applications with stronger security assurance than in any previous platform for distributed computing.

Chapter 3
The Decentralized Security Principle

In the last chapter we gave an informal description of what security means to Fabric users: the policies they express on their data can only be violated if principals that they explicitly trust to enforce those policies are untrustworthy. In order to use this metric to evaluate a system, we must first make the notions of harm and trust more concrete.

To see why this is important, let us consider the informal semantics that we proposed for the DLM labels in Chapter 2. Recall that a DLM confidentiality policy takes the form $\{o \to r\}$, where o and r are principals. The intended meaning of this policy is that o is the "owner" of the data, while r is a "reader" of the data.

The principal o might expect that she has control over the policy of the data, since she is designated as the owner. This expectation is reinforced by the fact that any code that downgrades the policy must have an explicit authority clause naming o. Only o can provide code marked with o's authority; such code must have provider label $\{o \leftarrow o\}$.

Based on these expectations and the informal statement of the DSP, o would be justified in concluding that data annotated with the policy $\{o \to r\}$ can only be downgraded by herself or someone she trusts. And yet, the semantics we gave to DLM labels specifies that $r \succcurlyeq \{o \to r\}$, so that even if $r \not\succcurlyeq o$, r can misbehave and cause the policy to be downgraded.

This example shows that the definitions for the $\succcurlyeq$, $\sqsubseteq$, and *authority* relations that we gave in Chapter 2 do not make it possible for the system to satisfy the DSP,

at least in this informal sense. This informal argument suggests that for the system to satisfy the DSP, the label model should have the following property:

Property 1. *If* $r \not\succcurlyeq C(\ell)$ *and* $\ell \not\sqsubseteq \ell'$ *then* $r \not\succcurlyeq I(\mathit{authority}(\ell, \ell'))$.

In this chapter, we construct a mathematical framework for reasoning about trust and authority in systems running on partially trusted platforms. We define a general notion of information flow and influenced information flow in a distributed system. We use these constructions to define harm in a way that describes the intended behavior of DIFC systems.

We give two different sets of definitions; the first describe a system that satisfies noninterference, while the second takes downgrading into account. We state and prove a form of the decentralized security principle for each set of definitions.

Our framework puts few requirements on the system model, and is therefore applicable to a broad range of settings. Flows and influenced flows are defined for an arbitrary transition relation on states; the only unusual requirement is that states can be partitioned into labeled components.

The label model is also treated abstractly in our framework. We only assume that a label model defines sets of principals, labels, along with is-trusted-to-enforce, flows-to, and authority relations that satisfy a few simple axioms.

We instantiate the label model with an extended version of the DLM (called EDLM) that is designed with the DSP in mind. The EDLM refines the DLM's trust hierarchy by allowing principals to express delimited trust in one another. Delimited trust allows principals to qualify the amount of trust they place in other principals, and consequently bound the amount of harm that other principals can cause.

In a distributed system like Fabric, no single node has a complete view of the state of the system. It should be the case that nodes can make conservative decisions about the safety of information flows even without a complete view of the state of the system.

We show that the ELDM has this property by giving a decision procedure for the safe relabeling relation that ensures that flows that are deemed safe in the presence of partial information remain safe if further information is taken into account. Furthermore, we show that if any flow is deemed unsafe that it is possible that the flow is actually unsafe. In this sense, our decision procedure is the best possible conservative approximation of the true relabeling relation.

3.1 System Model

The intended meaning of a label is usually described informally in terms of information flows and the information that influences those flows. This section presents a very general system model and defines information flows and influenced information flows. We also state and prove some basic properties of these relations.

The definitions in this section do not depend on a particular label model; all that is required is a set *Prin* of principals and a set *Lbl* of labels. Section 3.2 introduces additional requirements on a label model and uses them to state and prove the decentralized security principle.

3.1.1 System Model Requirements

The decentralized security principal is intended to apply to a broad class of systems. Here we describe the requirements and assumptions we make about these systems.

We consider systems with intended semantics given by a transition relation ($\rightarrow$) between states. For simplicity, we assume that the transition relation is total: for every state s there is at least one state s' with $s \rightarrow s'$.

Principals are responsible for enforcing security. Specifically, we assume that each component in the system is operated by a principal. For simplicity, we treat all components owned by a principal as a single component, and assume that each principal operates a component. We assume that the system state is partitioned into data held by each principal.

Since the systems of interest are those that protect information flow security, we assume that all data has an (explicit or implicit) label. The entire state of the system can therefore be encoded as a function $s : \mathit{Prin} \times \mathit{Lbl} \rightarrow \Sigma$ for an uninterpreted set Σ; the value $s(p, \ell)$ gives the portion of the local state of node p that is labeled ℓ.

Because the state is indexed by principal–label pairs, these pairs are called *locations*; the set of all locations is called $\mathit{Loc} := \mathit{Prin} \times \mathit{Lbl}$. By convention, λ, μ, and ν refer to locations.

3.1.2 Information Flow

Given a transition relation $\rightarrow$, it is useful to describe whether it causes information to flow from one location to another, and also what information influences those

flows. This section formalizes the definition of flow; Section 3.1.4 gives the definition of an influenced flow.

The definitions given in this section are inspired by definitions of noninterference based on weak bisimulation. The differ because they are not predicated on an information flow ordering on labels. Instead, they simply indicate what flows occur; we can then ask whether these flows are safe according to the semantics of the label model.

Definition 3.1 (State equivalence)**.** *If $\Lambda \subseteq Loc$ is a set of locations, s_1 and s_2 are* equivalent at Λ *(written $s_1 \approx_\Lambda s_2$) if for all $\lambda \in \Lambda$, $s_1(\lambda) = s_2(\lambda)$. If $s_1 \not\approx_\Lambda s_2$ we say s_1 and s_2 differ at Λ.*

$\overline{\Lambda}$ is defined as $Loc \setminus \Lambda$, so $s_1 \approx_{\overline{\Lambda}} s_2$ means s_1 and s_2 differ only at Λ (if at all).

Although these definitions and those given below are parameterized on a set of locations, we will primarily focus our attention on a single location at a time. By abuse of notation, λ is used to indicate $\{\lambda\}$, so that $s_1 \approx_\lambda s_2$ and $s_1 \approx_{\overline{\lambda}} s_2$ are defined.

Intuitively, information flows from λ to μ if altering the data at λ and then allowing the system to make some progress affects the data at μ. While it may be tempting to define progress as a single transition step, doing so treats transition steps as observable events, rather than artifacts of modeling decisions. Such a definition would lead to semantic conditions that are not stable under refinement.

Instead, we define flow in terms of an observable transition relation ($\overset{\Lambda}{\twoheadrightarrow}$). Intuitively, a Λ-observer notices a transition if the state labeled Λ changes. This intuition suggests the following tentative definition:

Definition 3.2 (Tentative Λ-observable transition relation)**.** *If Λ is a set of locations, we write $s \overset{\Lambda}{\twoheadrightarrow}_0 s'$ if*

$$s \to s_1 \to s_2 \to \cdots \to s_n \to s' \quad \text{and} \quad s \approx_\Lambda s_1 \approx_\Lambda s_2 \approx_\Lambda \cdots \approx_\Lambda s_n \not\approx_\Lambda s'$$

The $(\overset{\Lambda}{\twoheadrightarrow}_0)$ relation may not be total because there may be some states from which the Λ state never changes, a situation that looks like nontermination to a Λ observer. Our actual definition of the observable transition relation makes this situation observable[1] by adding a transition from s to itself in this case:

Definition 3.3 (Λ-observable transition relation)**.** *If Λ is a set of locations, a state s* Λ-observably transitions *to s' (written $s \overset{\Lambda}{\twoheadrightarrow} s'$) if either*

1. $s \to s_1 \to s_2 \to \cdots \to s_n \to s'$ *and* $s \approx_\Lambda s_1 \approx_\Lambda s_2 \approx_\Lambda \cdots \approx_\Lambda s_n \not\approx_\Lambda s'$, *or*
2. $s = s'$ *and there is no* $s'' \not\approx_\Lambda s$ *with* $s \to^* s''$.

Lemma 3.4 (The $(\overset{\Lambda}{\twoheadrightarrow})$ relation is total)**.** *For all Λ and s, there exists an s' with $s \overset{\Lambda}{\twoheadrightarrow} s'$.*

Proof. Follows from the definition and the fact that $\to$ is total. □

As above, if λ is a single location, we will use λ to denote $\{\lambda\}$, so that $(\overset{\lambda}{\twoheadrightarrow})$ is defined. Now we can define the flow relation:

Definition 3.5 (Flow)**.** *Information* flows *from λ to μ (written $\lambda \rightsquigarrow \mu$) if there exists observable transitions $s_1 \overset{\mu}{\twoheadrightarrow} s_1'$ and $s_2 \overset{\mu}{\twoheadrightarrow} s_2'$ such that $s_1 \approx_{\overline{\lambda}} s_2$ and $s_1' \not\approx_\mu s_2'$.*

[1] This is a deliberate simplification. In reality, nontermination is not an observable event. This assumption could probably be removed by giving more complex definitions of flows and influenced flows that explicitly account for nontermination; we leave a more detailed investigation of nontermination to future work.

We refer to (s_1, s_2) *as a witness of the flow* $\lambda \rightsquigarrow \mu$*. This situation is depicted in the following diagram:*

$$\begin{array}{ccc} s_1 & \xrightarrow{\mu}\!\!\!\!\twoheadrightarrow & s_1' \\ \approx_{\overline{\lambda}} & & \not\approx_{\mu} \\ s_2 & \xrightarrow{\mu}\!\!\!\!\twoheadrightarrow & s_2' \end{array}$$

The $\rightarrow$ relation describes the intended semantics of the system, but the definition of flow considers misbehavior. An attacker with the ability to alter data at location λ can take the system state from s_1 to s_2; the presence of the flow indicates that this influence will be observable in location μ. Dually, an attacker able to observe data at μ can make inferences about whether the original state was s_1 or s_2, even if s_1' and s_2' are supposed to subsequently transition to the same state.

The definition of flow is quite simple; witnesses can only differ in a single location. However, the flow relation still captures the behavior of attacks that affect multiple locations:

Lemma 3.6 (Flow from states differing in multiple locations)**.** *Let* Λ *be a finite set of locations. Given transitions* $s_1 \overset{\overline{\Lambda}}{\twoheadrightarrow} s_1'$ *and* $s_2 \overset{\overline{\Lambda}}{\twoheadrightarrow} s_2'$ *with* $s_1 \approx_{\overline{\Lambda}} s_2$ *and* $s_1' \not\approx_\mu s_2'$*, there exists some* $\lambda \in \Lambda$ *with* $\lambda \rightsquigarrow \mu$*.*

Proof. By induction on the size of Λ: form a chain of intermediate states, each differing from the next in a single element of Λ. Since $\overset{\mu}{\twoheadrightarrow}$ is total (Lemma 3.4), at least one pair of adjacent states will form a witness of a flow to μ. □

The flow relation is defined in terms of single observable steps of the transition relation $\rightarrow$. However, it also describes the many-step behavior of the system:

Lemma 3.7 (Big-step flow)**.** *Suppose* $s_1 \overset{\mu}{\twoheadrightarrow}^* s_1'$ *and* $s_2 \overset{\mu}{\twoheadrightarrow}^* s_2'$, *with* $s_1 \approx_{\overline{\lambda}} s_2$ *and* $s_1' \not\approx_\mu s_2'$. *Then* $\lambda \rightsquigarrow^* \mu$.

Proof. Induction on the lengths of $s_1 \overset{\mu}{\twoheadrightarrow}^* s_1'$ and $s_2 \overset{\mu}{\twoheadrightarrow}^* s_2'$. □

3.1.3 Transmission and Relabeling Flows

Flows can be characterized by the way they move information. Recall that a location consists of a principal that represents a node, as well as a label. Therefore a flow $(p, \ell) \rightsquigarrow (q, m)$ represents both communication (between p and q) and relabeling (from ℓ to m).

It can be helpful to focus on flows that either communicate data with a fixed label, or only relabel locally:

Definition 3.8 (Transmission flow, relabeling flow)**.** *A* transmission flow *from p to q with label ℓ is a flow of the form $(p, \ell) \rightsquigarrow (q, \ell)$, while a* relabeling flow *from ℓ to m at p is a flow of the form $(p, \ell) \rightsquigarrow (p, m)$.*

Operating system and programming language based information flow control systems are intended to prevent unsafe relabeling flows. Cryptographic techniques can be used to avoid transmission flows.

It should be possible to refine a system's transition relation to add relabeling flows either before or after transmission, so that all flows become either transmission flows

or relabeling flows. However, we leave proving this result for a general transition relation to future work.

3.1.4 Influenced Flows

The definition of flow captures the kind of information flows ruled out by noninterference. However, it is well known [SS05] that noninterference is too restrictive to describe many systems that are nevertheless considered secure.

There are many proposals for specifying the circumstances under which policies can be downgraded. Some (e.g., [Mye99a, PC00]) allow policies to be downgraded by any code that possesses sufficient authority. Others (e.g., [LZ05, KAMS19]) encode constraints on downgrading in the policies themselves, and only allow downgrading that is consistent with those policies. A third approach is to interpret policies in the context of a principal hierarchy or other authorization state (e.g., [ML00, BS06]). Many systems, including Fabric, use a combination of these approaches.

Reasoning about the safety of downgrading operations requires thinking about the integrity of the parts of the state that can influence relabeling decisions.

For example, if downgrading operations are only allowed in code that has been authorized by Alice, it had better be the case that principals Alice doesn't trust can't change the value of that code. Similarly, if a label model says that an information flow is safe as long as Bob trusts Chuck, the state used to determine whether Bob trusts Chuck must have high integrity.

These and similar restrictions are discussed in more detail in Section 3.2. Here we present a more nuanced relation that identifies the state that influences a flow.

Intuitively, data at ν influences a flow from λ to μ if changing the state at ν changes whether information flows or not. This intuition leads to the following definition, which is inspired by the definition of nonmalleable information flow:

Definition 3.9 (Influenced flow)**.** *Information labeled ν* influences *a flow from μ to ν (written $\lambda \overset{\nu}{\rightsquigarrow} \mu$) if there exists four states s_1, s_2, t_1, and t_2 such that $s_1 \approx_{\overline{\nu}} t_1$ and $s_2 \approx_{\overline{\nu}} t_2$, and such that (s_1, s_2) is a witness to a flow from λ to μ but (t_1, t_2) is not. This situation is depicted in the following diagram:*

$$\begin{array}{ccccccc}
t_1' & \overset{\mu}{\twoheadleftarrow} & t_1 & \approx_{\overline{\nu}} & s_1 & \overset{\mu}{\twoheadrightarrow} & s_1' \\
\approx_\mu & & \approx_{\overline{\lambda}} & & \approx_{\overline{\lambda}} & & \not\approx_\mu \\
t_2' & \overset{\mu}{\twoheadleftarrow} & t_2 & \approx_{\overline{\nu}} & s_2 & \overset{\mu}{\twoheadrightarrow} & s_2'
\end{array}$$

In the example of restricting downgrading to authorized code, the code would have label ν, while the data to be downgraded would have its label changed from λ to μ. Modifying the code that performs the downgrading (or the data it examines to decide whether to downgrade) without changing the rest of the system state would change the state from t_1 to s_1, or from t_2 to s_2. Without adding the downgrade, there was no flow ($t_1' \approx_\mu t_2'$), while after the change there is a flow (since $s_1' \not\approx_\mu s_2'$). In this way, modifying the code labeled ν influences the flow from λ to μ.

Similarly, influenced flow describes a system with a stateful flows-to relation. A label model may allow a flow from λ to μ if Bob trusts Chuck, but that determination may require examination of the system state. If the portion of the state that is examined is labeled ν, then there would be an influenced flow $\lambda \overset{\nu}{\rightsquigarrow} \mu$. Changing the state at ν would cause the system to allow or disallow the flow from λ to μ.

3.1.5 Influenced Flows Generalize Flows

The influenced flow relation gives strictly more information about the system than the flow relation. Here are some basic properties showing the relationships between the two relations:

Lemma 3.10 (Influenced flows imply flows)**.** *If* $\lambda \overset{\nu}{\rightsquigarrow} \mu$ *then* $\lambda \rightsquigarrow \mu$ *and* $\nu \rightsquigarrow \mu$*.*

Proof. Follows immediately from the definition. □

Lemma 3.11 (Flows imply influenced flows)**.** *If* $\lambda \rightsquigarrow \mu$ *then* $\lambda \overset{\lambda}{\rightsquigarrow} \mu$*.*

Proof. Choose $t_1 = t_2 = s_1$ in the definition of influenced flow. □

Lemma 3.12 (Asymmetry)**.** *It is not necessarily the case that* $\lambda \overset{\nu}{\rightsquigarrow} \mu$ *implies* $\nu \overset{\lambda}{\rightsquigarrow} \mu$*.*

Proof. Using the notation in Definition 3.9, we can construct a transition system where t_1', s_1', and s_2' all differ from each other on μ, while $t_1' \approx_\mu t_2'$. In this case both (s_1, t_1) and (s_2, t_2) are witnesses of flows from ν to μ, so these four states are not a witness to a λ-influenced flow. By choosing a sufficiently simple state space, we can ensure that these are the only witnesses of flows from ν to μ, thus ensuring that $\nu \overset{\lambda}{\not\rightsquigarrow} \mu$. □

3.2 The Decentralized Security Principle, Formalized

The decentralized security principle states that one's security should be immune to the actions of principals that one does not trust. Stated another way, if your security is harmed, it must be the result of an action by a principal you trust.

With definitions of flows and influenced flows in hand, we can describe the set of flows that a system exhibits. The question of whether these flows are harmful, and whether principals are trusted, depends on the semantic meaning of the labels.

There are many label models that describe the set of allowed flows and the conditions under which they are safe [MPP13]. This section treats the label model as an abstraction that provides a handful of relations between principals and labels that satisfy certain properties. Section 3.3 instantiates this abstraction with a label model designed with the DSP in mind.

Using these assumptions, we give two definitions of harm that are appropriate for reasoning about information flow. The first definition is based on the flow relation and is more restrictive than the second; we show that lack of harm using this definition is equivalent to noninterference. The second definition of harm is based on the influenced flow relation; lack of harm using this definition is similar to nonmalleable information flow.

A definition of harm describes the guarantees that a principal can expect if their trustees are trustworthy. It should also describe the actions that trustworthy principles should avoid. In this sense, the DSP is a kind of compositionality result: composing components operated by trustworthy principals should produce a trustworthy system. We show this property for each of the two definitions of harm.

3.2.1 Label Model Axioms

Section 3.1 only assumed that there is a set of labels and a set of principals. In order to identify which flows are harmful, we need additional structure relating principals

to labels and labels to each other. This additional structure is specified by giving a label model.

We are interested in confidentiality and integrity. Our first assumption is that each label has a confidentiality and an integrity component. Formally, we require that there is a set CLbl of confidentiality labels and a set ILbl of integrity labels such that $\mathit{Lbl} = \mathit{CLbl} \times \mathit{ILbl}$. We refer to the projections as C and I.

Flows-to relation. Some labels are more restrictive than others by definition. Our second assumption is that the label model provides relations $\sqsubseteq_C$ and $\sqsubseteq_I$ on CLbl and ILbl indicating when one label is more restrictive than another. $\ell \sqsubseteq_C m$ and $\ell \sqsubseteq_I m$ both mean that information labeled ℓ can safely affect data labeled m. We define $(\sqsubseteq)$ on labels as the product of $\sqsubseteq_C$ and $\sqsubseteq_I$, and we omit the subscripts when they are clear from context.

Unlike previous information flow analyses, we do not assume that the set of labels forms a lattice under $\sqsubseteq$. We do require that $\sqsubseteq_C$ and $\sqsubseteq_I$ are preorders (reflexive and transitive).[2]

Trusted-to-enforce relation. The label model must also provide a set Prin of principals. We do not assume any structure on the set of principals by themselves, but we do assume that each principal is only trusted to enforce a subset of the labels. The statement $p \succcurlyeq \ell$ should be read "p is trusted to enforce ℓ". We assume the label model defines the $(\succcurlyeq_C)$ relation between principals and confidentiality labels, and

[2] One can always formally turn a preorder into a lattice by taking equivalence classes and completing under $\sqcup$ and $\sqcap$. However, it may not be obvious how to define the $\succcurlyeq$ and *authority* relations on the resulting equivalence classes. Eschewing the lattice requirement removes this complexity.

the ($\succcurlyeq_I$) relation between principals and integrity labels. We define $\succcurlyeq$ on labels as the product of these two relations.

For the interpretation of C as confidentiality and I as integrity to make sense, we require certain relationships between $\succcurlyeq$ and $\sqsubseteq$:

1. If $\ell \sqsubseteq_I m$, then ℓ has higher integrity than m. There should therefore be fewer principals trusted to enforce ℓ. In this case, we require that if $p \succcurlyeq \ell$ then $p \succcurlyeq m$.

2. If $\ell \sqsubseteq_C m$ then m is more secret than ℓ. There should therefore be fewer principals trusted to enforce m. In this case, we require that if $p \succcurlyeq m$ then $p \succcurlyeq \ell$.

We can simplify these axioms by introducing a *trust ordering* on labels:

Definition 3.13 (Trust ordering). *We say ℓ is* harder to enforce *than m (written $\ell \succcurlyeq m$) if for all $p \succcurlyeq \ell$, we have $p \succcurlyeq m$. If $\ell \succcurlyeq m$ and $m \succcurlyeq \ell$ we write $\ell \simeq m$.*

This notation makes it easy to remember the relationship between $p \succcurlyeq \ell$ and $\ell \succcurlyeq m$: these two facts together imply that $p \succcurlyeq m$.

Using this notation, the above requirements become (1) If $C(\ell) \sqsubseteq C(m)$ then $C(\ell) \preccurlyeq C(m)$, and (2) If $I(\ell) \sqsubseteq I(m)$ then $I(\ell) \succcurlyeq I(m)$. Stated another way, $\sqsubseteq$ and $\preccurlyeq$ are the same on confidentiality labels, and are opposite on integrity labels.

Authority function. In the presence of downgrading, the $\sqsubseteq$ relation does not define the entirety of the allowable information flows. Instead, the allowable flows are dictated by the code that is authorized to declassify.

When a principal adds a label to data, they have some intended allowable use for that data. Compliance with that intent can be thought of as an unstated invariant of the code and data that influence either the decision to downgrade the data or the choice of data to be downgraded.

The ability to maintain invariants is synonymous with high integrity. Therefore it makes sense to use an integrity label to categorize the information that may influence the decision to downgrade a label ℓ.

The confidentiality of the code and data used to authorize downgrades is also important. Cecchetti et al. have shown that allowing secret data to influence downgrading decisions can cause confused deputy vulnerabilities [CMA17].

We assume that the label model defines a function $\mathit{authority} : \mathit{Lbl} \times \mathit{Lbl} \to \mathit{Lbl}$. Given labels ℓ and m, the label $\mathit{authority}(\ell, m)$ identifies the state that is allowed to influence a relabeling from ℓ to m. The integrity component of the authority indicates how trusted the code or data must be, while the confidentiality component indicates how transparent the decision to downgrade must be.

The requirements on $\mathit{authority}(\ell, m)$ depend on whether a flow $\ell \rightsquigarrow m$ would be a downgrade. If $C(\ell) \sqsubseteq C(m)$ then a flow $\ell \rightsquigarrow m$ does not declassify ℓ data, so it does not require a relationship between ℓ and $\mathit{authority}(\ell, m)$. Likewise, if $I(\ell) \sqsubseteq I(m)$ then we don't require a relationship between $\mathit{authority}(\ell, m)$ and m.

If $C(\ell) \not\sqsubseteq C(m)$, then $\mathit{authority}(\ell, m)$ and $C(\ell)$ are closely related: The statement $p \succcurlyeq C(\ell)$ means that if p is untrustworthy, ℓ can be inappropriately leaked. Certainly, if $p \succcurlyeq \mathit{authority}(\ell, m)$, then p could misbehave by introducing code that declassifies ℓ data inappropriately. Therefore, if $p \succcurlyeq \mathit{authority}(\ell, m)$ then we should have $p \succcurlyeq C(\ell)$.

On the other hand, $p \succcurlyeq C(\ell)$ means that p can view data labeled ℓ. That means that by misbehaving, p has the ability to change who can view data labeled ℓ. Thus p has the de-facto ability to change the meaning of $C(\ell)$. Therefore, if $p \succcurlyeq C(\ell)$ then we require $p \succcurlyeq \mathit{authority}(\ell, m)$.

These requirements can be summarized by saying that if $C(\ell) \not\sqsubseteq C(m)$ then $C(\ell) \simeq \mathit{authority}(\ell, m)$. A similar argument justifies the requirement that if $I(\ell) \not\sqsubseteq I(m)$ then $I(m) \simeq \mathit{authority}(\ell, m)$.

Label model definition. The above requirements are summarized in the following definition:

Definition 3.14 (Label model)**.** *A label model $\mathcal{M}$ consists of a set Prin, preorders $(\mathit{CLbl}, \sqsubseteq_C)$ and $(\mathit{ILbl}, \sqsubseteq_I)$, relations $(\succcurlyeq_C) \subseteq \mathit{Prin} \times \mathit{CLbl}$ and $(\succcurlyeq_I) \subseteq \mathit{Prin} \times \mathit{ILbl}$, and a function $\mathit{authority} : \mathit{Lbl} \times \mathit{Lbl} \to \mathit{Lbl}$ satisfying the following axioms:*

1. *If $C(\ell) \sqsubseteq C(m)$ then $C(\ell) \preccurlyeq C(m)$,*
2. *If $I(\ell) \sqsubseteq I(m)$ then $I(\ell) \succcurlyeq I(m)$,*
3. *If $C(\ell) \sqsubseteq C(m)$ then $C(\ell) \preccurlyeq C(m)$; otherwise $\mathit{authority}(\ell, m)) \simeq C(\ell)$*
4. *If $I(\ell) \sqsubseteq I(m)$ then $I(\ell) \succcurlyeq I(m)$; otherwise $\mathit{authority}(\ell, m)) \simeq I(m)$*

3.2.2 DSP with Strict Harm

We are now ready to define strict noninterference-based harm and prove the Decentralized Security Principle for this definition.

When considering strict noninterference-based information flow policies, there are two ways that a flow $(p, \ell) \rightsquigarrow^* (q, m)$ can harm the confidentiality of ℓ: either $q \not\succcurlyeq C(\ell)$ (in which case confidential data is placed on a component that is not trusted to hold it) or $C(\ell) \not\sqsubseteq C(m)$ (in which case the restrictions on the use of data are relaxed).

To capture this intuition, we extend the definition of $(\sqsubseteq_C)$ from labels to locations as follows:

Definition 3.15 (Confidentiality can-flow for locations $(\lambda \sqsubseteq_C \mu)$)**.** *We say $(p, \ell) \sqsubseteq_C (q, m)$ if $C(\ell) \sqsubseteq_C C(m)$ and $q \succcurlyeq C(\ell)$.*

Definition 3.16 (Confidentiality harm)**.** *We say $(p, \ell) \rightsquigarrow^* (q, m)$* harms *$C(\ell)$ if $p \succcurlyeq C(\ell)$, and $(p, \ell) \not\sqsubseteq_C (q, m)$.*

The requirement that $p \succcurlyeq C(\ell)$ prevents an untrusted party from manufacturing fake harm by inventing data, claiming it has label ℓ, and then leaking it. Following Zagieboylo et al. [ZSM19], we refer to such locations as C-compromised:

Definition 3.17. *If $p \not\succcurlyeq C(\ell)$ then the location (p, ℓ) is C-*compromised*. Similarly, Similarly, if $p \not\succcurlyeq I(\ell)$ then (p, ℓ) is I-*compromised*, and if $p \not\succcurlyeq \ell$ then (p, ℓ) is* compromised*.*

Dually, the flow harms q's integrity if either p is insufficiently trusted or ℓ is insufficiently restrictive:

Definition 3.18 (Integrity can-flow for locations $(\lambda \sqsubseteq_I \mu)$)**.** *We say $(p, \ell) \sqsubseteq_I (q, m)$ if $I(\ell) \sqsubseteq I(m)$ and $p \succcurlyeq I(m)$.*

Definition 3.19 (Integrity harm). *We say* $(p, \ell) \rightsquigarrow^* (q, m)$ harms $I(m)$ *if* $q \not\succcurlyeq I(m)$, *and* $(p, \ell) \not\sqsubseteq_I (q, m)$.

The various can-flow orderings on locations are *almost* preorders:

Lemma 3.20 (Can-flow is transitive and mostly reflexive). *The relations* $\sqsubseteq_C$, $\sqsubseteq_I$ *and* $\sqsubseteq$ *on locations are all transitive. They are reflexive (and thus preorders) when restricted to uncompromised labels.*

Proof. Follows immediately from the label model axioms. □

In fact, the can-flow orderings cannot possibly be reflexive because of the following interesting fact:

Lemma 3.21 (Information can only flow to/from uncompromised labels). *If* $\lambda \sqsubseteq_C \mu$ *then* μ *is not* C*-compromised. If* $\lambda \sqsubseteq_I \mu$ *then* λ *is not* I*-compromised.*

Proof. Follows immediately from the label model axioms. □

With a definition of harm, we can formalize the notion that a principal can only be harmed by someone they trust:

Theorem 3.22 (DSP with strict noninterference-based harm). *If* $(p, \ell) \rightsquigarrow^* (q, m)$ *harms* $C(\ell)$*, then there is a harmful single-step flow* $(p', \ell') \rightsquigarrow (q', m')$ *with* $p' \succcurlyeq C(\ell)$*. Dually, if the flow harms* $I(m)$ *then there is a harmful flow* $(p', \ell') \rightsquigarrow (q', m')$ *with* $q' \succcurlyeq I(m)$*.*

Proof. If $(p, \ell) = (p_0, \ell_0) \rightsquigarrow (p_1, \ell_1) \rightsquigarrow \cdots \rightsquigarrow (q, m)$, we can find the first i with $(p_i, \ell_i) \not\sqsubseteq_C (p_{i+1}, \ell_{i+1})$ (such an i must exists since $(p, \ell) \not\sqsubseteq_C (q, m)$). Since $\sqsubseteq_C$ is transitive, we have $(p, \ell) \sqsubseteq_C (p_i, \ell_i)$. Therefore $p_i \succcurlyeq C(\ell)$.

The proof for integrity flows is the same, except that we find the *last* i with $(p_i, \ell_i) \not\sqsubseteq_I (p_{i+1}, \ell_{i+1})$, instead of the first such i. $\square$

3.2.3 Strict Harm and Noninterference

In this section we give a typical definition of noninterference adapted to our distributed setting, and show that noninterference and lack of strict harm are equivalent. This result shows that our definitions of flow and strict harm correctly describe information flow.

The literature contains a large variety of definitions of noninterference. Ours is based on observational determinism [ZM03].

Noninterference is usually defined in terms of a low-equivalence relation requiring states to be equal at all labels below ℓ in the information flow ordering. The following definition of the low closure of a location simplifies the definitions of low equivalence and noninterference:

Definition 3.23 (Low-closure)**.** *If $\lambda \in Loc$, we define the* low-closure *of λ (written $\downarrow \lambda$) to be the set of locations that may not flow to λ. Formally, $\downarrow \lambda := \{\mu \mid \lambda \not\sqsubseteq \mu\}$.*[3]

We can now define low equivalence and noninterference:

Definition 3.24 (Low-equivalence)**.** *Two states s_1 and s_2 are* low-equivalent *with respect to location λ if $s_1 \approx_{\downarrow\lambda} s_2$.*

[3]Technically, this should be called the complement of the high-closure of $\{\lambda\}$ and written $\overline{\uparrow \lambda}$, but we adopt the notation given here for simplicity. Our definition of $\downarrow \lambda$ is downward closed, but does not contain λ.

Definition 3.25 (Noninterference)**.** *A system given by transition relation* $\rightarrow$ *is* non-interfering *with respect to location* λ *if for all* $s_1 \approx_{\downarrow\lambda} s_2$*, if* $s_1 \overset{\downarrow\lambda}{\twoheadrightarrow} s_1'$ *and* $s_2 \overset{\downarrow\lambda}{\twoheadrightarrow} s_2'$ *then* $s_1' \approx_{\downarrow\lambda} s_2$*. The relation is noninterfering if it is noninterfering with respect to all locations.*

We are now ready to show that our notion of harm captures noninterference:

Theorem 3.26 (Noninterference is no-harm)**.** *For a given label* ℓ*,* $(\rightarrow)$ *is noninterfering if and only if it exhibits no harm.*

Proof. The "if" direction follows from Lemma 3.6, while the "only if" follows directly from the definitions. □

3.2.4 DSP with Downgrading

Stating the DSP is more difficult in the presence of downgrading. Downgrading makes it possible for labeled ℓ to safely affect data labeled m even if $\ell \not\sqsubseteq m$. The conditions under which such a flow is safe application dependent, may differ for every label, and may change over time.

We assume that the intended policy for relabeling data from label ℓ to label m is stored with label $authority(\ell, m)$. This means there are several ways to cause harm:

Definition 3.27 (Confidentiality harm)**.** *An influenced flow* $(p, \ell) \overset{(r,n)}{\rightsquigarrow} (q, m)$ *is considered harmful to* $C(\ell)$ *if any of the following conditions apply:*

1. *If* $q \not\succcurlyeq C(\ell)$ *then* p *has harmed* $C(\ell)$ *by leaking data.*

2. *If* $C(\ell) \not\sqsubseteq C(m)$ *and* $I(n) \not\sqsubseteq I(authority(\ell, m))$ *or* (r, n) *is I-compromised, then* p *has harmed* $C(\ell)$ *by inappropriately declassifying data.*
3. *If* $C(\ell) \not\sqsubseteq C(m)$ *and* $C(authority(\ell, m)) \not\sqsubseteq C(n)$ *then* p *has harmed* $C(\ell)$ *by opaquely declassifying data.*
4. *If* $C(\ell) \not\sqsubseteq C(m)$ *and* $I(n)$ *is harmed, then* $C(\ell)$ *is harmed (not necessarily by* p*), because the meaning of* $C(\ell)$ *has been changed inappropriately.*
5. *If* $C(\ell) \sqsubseteq C(m)$ *and* $C(m)$ *is harmed then* $C(\ell)$ *has also been harmed (not necessarily by* p*).*

A few notes about this definition are in order:

- Condition 1 says that any flow from ℓ to an untrusted component q is harmful, regardless of the influence. One might object that the policy associated with ℓ might say that ℓ can be stored on q even though q is not trusted to enforce $C(\ell)$. However, such a policy would violate the DSP, because q could then leak data labeled ℓ despite not being trusted to enforce $C(\ell)$. Instead, the policy should state that data labeled ℓ can be declassified to a label that q is trusted to enforce; then p could remain trustworthy by first relabeling the data.
- Conditions 2 and 3 are important for ensuring robust declassification and transparent endorsement. Nonmalleable information flow is defined as the intersection of robustness and transparency [CMA17].

  It is tempting to combine these conditions into one by using the trust ordering on labels: Under mild additional assumptions, the condition

  $$I(n) \not\sqsubseteq I(authority(\ell, m)) \text{ or } C(authority(\ell, m)) \not\sqsubseteq C(n)$$

is equivalent to the condition $n \not\succcurlyeq authority(\ell, m)$.

We have avoided this unification because it obscures the meaning of condition 3. The real requirement for transparency is that the influence is sufficiently public, not that the label on the influence is harder to enforce. Transparency is fundamentally a statement about information flow, rather than trust or enforcement.

- Condition 5 does not consider a downgrading from ℓ to a *less* restrictive label m followed by m-harm to be harmful to label ℓ. Declassifying data is an explicit statement that leaking that data is less harmful.

The conditions for integrity harm are mostly dual to those for confidentiality harm:

Definition 3.28 (Integrity harm)**.** *An influenced flow* $(p, \ell) \overset{(r,n)}{\rightsquigarrow} (q, m)$ *is considered harmful to* $I(m)$ *if any of the following conditions apply:*

1. *If* $p \not\succcurlyeq I(m)$ *then* q *has harmed* $I(m)$
2. *If* $I(\ell) \not\sqsubseteq I(m)$ *and* $I(n) \not\sqsubseteq I(authority(\ell, m))$ *or* (r, n) *is* I*-compromised, then* q *has harmed* $I(m)$ *by inappropriately endorsing data.*
3. *If* $I(\ell) \not\sqsubseteq I(m)$ *and* $C(authority(\ell, m)) \not\sqsubseteq C(n)$ *then* q *has harmed* $I(m)$ *by opaquely endorsing data.*
4. *If* $I(\ell) \not\sqsubseteq I(m)$ *and* $I(n)$ *is harmed, then* $I(m)$ *is harmed (not necessarily by* q*), because the meaning of* $I(m)$ *has been changed inappropriately.*
5. *If* $I(\ell) \sqsubseteq I(m)$ *and* $I(\ell)$ *is harmed then* $I(m)$ *has also been harmed (not necessarily by* q*).*

We are now ready to state and prove the DSP with influenced-flow based definitions of harm.

Theorem 3.29 (DSP with downgrading)**.** *If a flow originating from a C-uncompromised location (p, ℓ) harms ℓ, then there is a principal p' such that p' causes harm and $p' \succcurlyeq \ell$. Dually, if a flow ends in an I-uncompromised location (q, m) harms m, then there is a principal q' such that q' causes harm and $q' \succcurlyeq \ell$.*

Proof. We present the proof for confidentiality harm; the proof for integrity harm is completely dual. We consider the different forms of confidentiality harm enumerated in Definition 3.27 in turn. For the first three forms of confidentiality harm p is causing the harm, so we can choose $p' := p$. Since (p, ℓ) is uncompromised, we know $p \succcurlyeq \ell$ as required.

For the fourth form of confidentiality harm, we know that (r, n) is I-uncompromised (otherwise condition 2 would be violated). We can therefore inductively find some r' that causes harm, with $r' \succcurlyeq I(n)$. We have $I(n) \succcurlyeq I(\mathit{authority}(\ell, m))$ (otherwise condition 2 or 3 would be violated). By the label model axioms, since $C(\ell) \not\sqsubseteq C(m)$, we have $I(\mathit{authority}(\ell, m)) \succcurlyeq C(\ell)$, so $r' \succcurlyeq C(\ell)$.

For the fifth form of confidentiality harm, we know that $C(\ell) \sqsubseteq C(m)$, so $C(\ell) \preccurlyeq C(m)$. We also know $m \succcurlyeq C(m)$ (otherwise p is directly causing harm, so we would be done). Therefore (q, m) is C-uncompromised. We can inductively conclude that there is some $q' \succcurlyeq C(m)$ that harms $C(m)$. Since $C(\ell) \preccurlyeq C(m)$ we have $q' \preccurlyeq C(\ell)$. □

3.2.5 Influenced Flows and Nonmalleability

Although the definition of influenced flow is inspired by the definition of nonmalleable information flow, there are subtle differences between our definitions and those given in [CMA17].

The formal definition of nonmalleability begins by using a set A of attackers to divide the space of labels into four quadrants; each label is classified as public or secret, and as trusted or untrusted. It then requires an untrusted attacker cannot change whether there is a flow from public to secret, and that secret data cannot change whether there is a flow from trusted to untrusted.

Influenced flows have finer resolution, because they detect flows between any two labels, even if those labels are both public or if they are both trusted with respect to a given attacker. We view these flows as important to the security of the system, because whether an attacker is trusted to enforce a given label may change over time.

Another distinction between the influenced-flow based definition of harm and nonmalleability is that our definition of harm places a restriction on the integrity of the data that influences endorsement and on the confidentiality of the data that influences declassification.

Although robust endorsement has been considered in the literature [AM10], the dual notion of transparent declassification has not been previously studied. We leave a detailed exploration of the ramifications of these differences to future work.

3.3 The Extended Decentralized Label Model

Section 3.2 defined a label model as as a collection of principals and labels satisfying certain properties. In this section we instantiate this definition with a language of information flow labels and relationships between them. This language both refines and extends the decentralized label model by allowing the specification of fine-grained trust relationships.

Principals and trust. The EDLM is parameterized on the set *Prin* of principals. Unlike many label models, EDLM does not assume a single trust ordering on principals. Instead, statements of trust are delimited: a principal expressing trust must also specify the extent of that trust. The extent of trust is specified by giving a *category* (usually denoted by $c, d, e \in C$).

The trust specification is described by a *delimited trust hierarchy* (DTH). A delimited trust hierarchy H is a C-indexed collection of preorders $\preccurlyeq_c^H$ on *Prin*. The notation $p \preccurlyeq_c^H q$ should be read "p trusts q to protect c in H". We omit H when it is clear from the context.

Labels. Figure 3.1 gives the syntax of EDLM labels. The labels are syntactically similar to those in the DLM, but policies are stated in terms of categories as well as principals, and the label semantics take the delimited trust hierarchy into account.

The simplest confidentiality label is a *confidentiality policy* $\{p \to c\}$. Informally, if data has the label $\{p \to c\}$, it means that principal p is concerned with the secrecy of the data, and if the secrecy of the data is compromised, p is harmed at level c.

$$
\begin{aligned}
p, q, r &\in \mathit{Prin} \\
c, d, e &\in C \\
\mathit{Pol} &::= \{p \leftarrow c\} \mid \{p \rightarrow c\} \\
\ell_c, m_c \in \mathit{CLbl} &::= \{p \rightarrow c\} \mid \ell_c \sqcup m_c \mid \ell_c \sqcap m_c \mid \{\} \\
\ell_i, m_i \in \mathit{ILbl} &::= \{p \leftarrow c\} \mid \ell_i \sqcup m_i \mid \ell_i \sqcap m_i \mid \{\}
\end{aligned}
$$

Figure 3.1: The syntax of EDLM labels

Dually, one can write an *integrity policy* $\{p \leftarrow c\}$, which means that principal p is concerned with the integrity of the data, and if the integrity of the data is compromised, p is harmed at level c.

As in the DLM, we form more complicated confidentiality or integrity labels by taking formal joins and meets of policies. Informally, data labeled with the join of ℓ and m (denoted $\ell \sqcup m$) should be treated at least as restrictively as both ℓ and m, while the meet (denoted $\ell \sqcap m$) means data should be no handled more restrictively than either.[4] Finally, we allow empty confidentiality policies, which describe completely public data, and empty integrity policies, which describe completely untrusted data.

The flows-to and is-trusted-to-enforce relations. The flows-to relation for EDLM labels is given in Figure 3.2. Data labeled with a confidentiality policy owned by p is allowed to flow to a label owned by p' if and only if p trusts p' to protect the data. Dually, data labeled with an integrity policy owned by p is only allowed to flow *from* a label owned by p' if p trusts p' to have protected the data.

[4]The original DLM [ML97] did not permit label meets, but a semantics for them is given in [CM06], and they are implemented as part of the Jif language [MZZ+06].

$$\frac{p \preccurlyeq_c p'}{\{p \to c\} \sqsubseteq_C \{p' \to c'\}} \text{ (conf. policy)} \qquad \frac{p \succcurlyeq_c p'}{\{p \leftarrow c\} \sqsubseteq_I \{p' \leftarrow c'\}} \text{ (int. policy)}$$

$$\frac{}{\{\} \sqsubseteq_C \ell} \qquad \frac{}{\ell \sqsubseteq_I \{\}}$$

$$\frac{\ell_1 \sqsubseteq m \quad \ell_2 \sqsubseteq m}{\ell_1 \sqcup \ell_2 \sqsubseteq m} \qquad \frac{\ell \sqsubseteq m_1}{\ell \sqsubseteq m_1 \sqcup m_2} \qquad \frac{\ell \sqsubseteq m_2}{\ell \sqsubseteq m_1 \sqcup m_2}$$

$$\frac{\ell_1 \sqsubseteq m}{\ell_1 \sqcap \ell_2 \sqsubseteq m} \qquad \frac{\ell_2 \sqsubseteq m}{\ell_1 \sqcap \ell_2 \sqsubseteq m} \qquad \frac{\ell \sqsubseteq m_1 \quad \ell \sqsubseteq m_2}{\ell \sqsubseteq m_1 \sqcap m_2}$$

Figure 3.2: The EDLM flows-to relation. The first line shows the relation between policies; the second line describes joins, and the third describes meets.

The remainder of the rules codify the idea that $\{\}$ places no restrictions on confidentiality or integrity, that $\ell \sqcup m$ is more restrictive than both ℓ and m, and that $\ell \sqcap m$ is less restrictive than either of them.

The is-trusted-to-enforce relation comes directly from the DTH: $q \succcurlyeq \{p \to c\}$ and $q \succcurlyeq \{p \leftarrow c\}$ if and only if $p \preccurlyeq_c q$.

The authority function. We define the *authority* function for EDLM in terms of a simpler function $\mathit{defn} : \mathit{Pol} \to \mathit{Lbl}$ that gives the defining label of a given policy. The EDLM gives a simple definition of $\mathit{defn} : \mathit{Pol} \to \mathit{Lbl}$:

$$\mathit{defn}(\{p \to c\}) := \{p \leftarrow c\} \quad \text{and} \quad \mathit{defn}(\{p \leftarrow c\}) := \{p \to c\}$$

This definition extends the *readers-to-writers* function of [ZM01] to support transparent endorsement. It is similar to the hourglass operator that Zagieboylo et al. use to enforce nonmalleable information flow [ZSM19].

We believe that this is not the only reasonable definition of *defn*. A more restrictive choice of the integrity of $defn(p)$ would allow a principal to delegate the ability to handle data without delegating the ability to make decisions about the policy on the data. A more public choice of the confidentiality of $defn(p)$ might be a way of requiring auditable endorsement. In the limit, if $defn(p)$ describes data that nobody is trusted to provide, it should become impossible to downgrade p, so our security definitions become equivalent to noninterference. We leave the exploration of this design parameter to future work.

The *authority* function is defined by extending the *defn* function to operate on joins and meets of labels.

Label model axioms. The operations defined in this section have been designed to satisfy the label model axioms of Section 3.2.1 in mind:

Lemma 3.30 (EDLM is a label model)**.** *The relations on EDLM labels defined above satisfy the label model axioms.*

Proof. By inspection. □

3.3.1 Constructing a DTH

In order to implement our model in a large distributed system, we need a way for principals to learn about the trust hierarchy. In a large distributed system, it is difficult to maintain globally consistent information. In particular, each principal may have only a partial view of the trust hierarchy. Thus the construction of the hierarchy should be robust with respect to partial information.

$$\frac{}{S, p \text{ says } q \succcurlyeq_c r \vdash p \text{ says } q \succcurlyeq_c r} \quad \text{(given)}$$

$$\frac{S \vdash p \text{ says } q \succcurlyeq_c p \quad S \vdash q \text{ says } r \succcurlyeq_c s}{S \vdash p \text{ says } r \succcurlyeq_c s} \quad \text{(trust)}$$

$$\frac{S \vdash p \text{ says } q \succcurlyeq_c r \quad S \vdash p \text{ says } r \succcurlyeq_c s}{S \vdash p \text{ says } q \succcurlyeq_c s} \quad \text{(transitivity)}$$

$$\frac{}{S \vdash p \text{ says } q \succcurlyeq_c q} \quad \text{(reflexivity)}$$

$$\frac{S \vdash p \text{ says } q \succcurlyeq_c p}{p \preccurlyeq_c^{[\![S]\!]} q}$$

Figure 3.3: Construction of a DTH $[\![S]\!]$ from a set S of trust assertions.

A second important property is the ability for principals to express trust relationships for other principals. For example, p's decision to extend trust to q may be dependent on an expensive computation, or p may be temporarily unavailable. In this case, a node that p trusts should be able to speak for p. Of course a principal should not be able to affect the security of another principal by issuing such statements.

To achieve these goals, we will now give a construction that takes as input a set S of statements of the form "p says $q \succcurlyeq_c r$," and outputs a valid DTH $[\![S]\!]$. The construction is given in Figure 3.3, and proceeds in two steps. First, we close the set of statements by allowing trusted parties to make new statements about trust (the trust rule). In addition, the set is closed under the DTH axioms. In the second step, we take only statements that a principal makes about itself as the resulting DTH. This requirement prevents statements from untrusted entities to affect the security of a principal.

It is straightforward to check that this construction satisfies the desired properties: the hard work has been done in defining the DTH axioms and analyzing the properties there. We need to check two things:

Lemma 3.31 (Construction yields a DTH)**.** *The relation* $\preccurlyeq^{[\![S]\!]}$ *satisfies the DTH axioms.*

Lemma 3.32 (Construction is stable under partial information)**.** *If* $S' \supseteq S$ *and* $p \preccurlyeq_c^{[\![S]\!]} q$ *then* $p \preccurlyeq_c^{[\![S']\!]} q$.

Proofs. Trivial. □

3.4 Application to Fabric

Principals are themselves reified as objects in Fabric. Each principal object specifies the set of principals that it trusts. As in the DLM, trust is complete—a principal either trusts another or it does not. The principal hierarchy is constructed from these specifications, allowing the system to make judgments based on the DLM.

As in our model, Fabric proposes an "is trusted to enforce" relation ($\preccurlyeq$) between principals and labels, and uses this relation to decide whether to an object to or receive an object from another node. However, the Fabric policy $\{o \to r\}$ treats r as a *reader*, and Fabric allows an object with that policy to be sent to a node operated by r (as an owner, o is also able to read an object labeled $\{o \to r\}$.

This difference exposes a shortcoming in the Fabric security model. Consider the following flow, where Alice (a), Bob (b), and Chuck (c) are principals with no trust relationship:

$$(a, \{a \to b\}) \rightsquigarrow (b, \{a \to b\}) \rightsquigarrow (c, \{a \to b\})$$

This trace is clearly harmful to Alice: her policy states that only Bob may read the data, and yet the data ends up on Chuck's node. However, no principal that Alice trusts performs a harmful step. According to the Fabric trusted-to-enforce relation, the first step is not harmful. The second step is harmful, but Bob is not trusted by Alice. Thus the Fabric system does not satisfy the decentralized security principle.

Now one might argue that in fact, Bob should implicitly be trusted to enforce policies that say he may read the data, and thus the second step in this flow is in fact a trusted harm step. In other words, the relation $p \preccurlyeq_q q$ should be implicit. There are three problems with this argument.

First, this assumption is inconsistent with the checks performed by the Fabric language. The Fabric language includes mechanisms for weakening information flow constraints through declassification and endorsement. If code is marked as having the authority of Alice, it is able to remove the label $\{a \to b\}$. The intent is that such code constitutes part of Alice's statement of policy (in addition to the labels on her data and the statements of trust made by her principal object), and as such should be checked by Alice herself. Code with the authority of Bob is unable to remove the policy $\{a \to b\}$. Based on these restrictions, a Fabric programmer could reasonably assume that $b \not\succcurlyeq_b a$.

Second, the implicit assumption that $b \succcurlyeq_b a$ prevents the system from enforcing useful real-world security policies. For example, certain psychological records are allowed to be seen by patients, but only as specified by the doctor. It would be reasonable to model these restrictions as the policy $\{\text{doctor} \to \text{patient}\}$, with the doctor's conditions for release specified as code blessed with the authority of the doctor. However, if patient $\succcurlyeq_{\text{patient}}$ doctor, then the doctor's consent becomes unnecessary, violating the spirit of the policy.

Third, under the implicit rule that $b \succcurlyeq_b a$, the labels $\{a \to b\}$ and $\{b \to a\}$ are treated identically[5]. This means that the complicated syntax for specifying policies is an unnecessary complexity.

One possible fix is to simply change the $\succcurlyeq$ relation for Fabric to match the one presented here, without adding delimited trust statements. However, this would lead to an overly restrictive system: data owned by Alice would be stuck on nodes controlled by Alice (or those who act for her). If Alice actually *does* trust Bob to handle data that he is able to read, she would have to construct a new principal p that trusts both she and Bob, and relabel her data from $\{a \to b\}$ to $\{p \to p\}$.

This argument demonstrates that the *delimited* trust statements that we are proposing provide a useful middle ground. We leave implementation on delimited trust in Fabric to future work.

[5] In fact, both $\{a \to b\}$ and $\{b \to a\}$ are both equivalent to $\{\top \to a \wedge b\}$, where $\top$ is the top principal in the trust ordering, and $a \wedge b$ is the meet of a and b

3.5 Related Work

Our notion of influenced flows is motivated by *robust declassification*, originally proposed by Zdancewic and Myers [ZM01]. Robustness is intended to rule out laundering attacks, where an active adversary is able to abuse allowed declassification statements in unexpected ways, thereby violating the intended information security policy. This restriction is formally captured by requiring that the set of information that a system releases is independent of the information that an attacker controls.

Several refinements of robust declassification have been defined. Chong and Myers [CM06] interpret robustness in the context of the DLM and define robustness with respect to all attackers. Askarov and Myers [AM10] use attacker influence and knowledge to give a semantic characterization of robustness. Myers et al. [MSZ06] show that robust declassification can be enforced in a language-based setting by restricting the information that can affect the context of a declassification.

Nonmalleability [CMA17] places additional information-flow restrictions on the context. Our authority labels generalize this restriction further by allowing a label model to specify an arbitrary information flow label.

Clarkson and Schneider [CS08] define a safety hyperproperty as a set of sets of traces in which "something bad doesn't happen," and show that various definitions of noninterference are safety hyperproperties. Cecchetti et al. [CMA17] extend this by showing that robust declassification, transparent endorsement, and nonmalleable information flow are all safety hyperproperties. Our definitions of flow and influenced flow focus on the "potentially bad things" that might happen and enable reasoning about the conditions under which they occur.

EDLM is based on the decentralized label model [ML00], and the design of the delimited trust hierarchy was partially inspired by Jif's dynamic principals [TZ07]. Other information flow label models also use trust as the basis for defining information flow labels. FLAM labels [ALM15] unify principals and labels. Rx [SHTZ06] has a more refined label model based on role-based access control.

Montagu et al. [MPP13] define a general label algebra structure and use it to compare several label models. Although label algebras include a notion of authority and downgrading, they cannot be used to define properties like robust declassification or nonmalleability because there is no information flow label associated with a given authority level.

Our *authority* label gives a coarse-grained restriction on downgrading, under the assumption that programs that are given that authority label correctly implement downgrading policies. Several richer languages for specifying downgrading policies exist. Chong and Myers [CM04] define eventual downgrading policies that specify the conditions under which data can be declassified and the policies that control them after they are downgraded. Reactive information flow labels [KS20, KAMS19] specify downgrading policies using finite automata. Paralocks [BS06] use logical formulas over a set of mutable lock variables to determine whether a relabeling is allowed.

Chapter 4
Information Leaks via Authorization Requests

The previous chapter presented a mathematical framework for reasoning about information flow control ina partially trusted system. In this chapter we narrow our focus and consider the impact that partial trust has on dynamic authorization. Dynamic authorization refers to any operation that a program or system performs at run time to determine whether to perform an action. In this chapter we focus on a specific authorization problem: The actions to be authorized are information flows; these actions should be permitted if the flows are compatible with the $\sqsubseteq$ relation.

Dynamic authorization is an important feature for building realistic software, because trust relationships and data policies are often unknown when programs are written. Dynamic authorization also increases the expressiveness of security-typed languages. For example, Jif programs often use dynamic authorization primitives to implement traversals over collections of heterogeneously labeled data.

Language features for dynamic authorization in information flow type systems have been studied in the literature [ZM07, TZ07, SHTZ06] and implemented in Jif. These analyses cannot be readily adapted to a partially trusted platform because they rely on assumptions that untrustworthy principals can violate.

In particular, most prior work has assumed that dynamic authorization queries have no visible side effects, and therefore have no impact on confidentiality.[1] This

[1] The only exception we are aware of is the FLAM authorization logic [ALM15]. FLAM's flow-limited judgments include an information-flow label constraining the set of facts needed to produce a derivation. FLAM assumes that the proof search process only leaks information to the hosts

assumption is reasonable in a centralized system: the users can specify their policies in advance, and the platform can simply read these specifications as needed. In a decentralized setting, however, the state that must be consulted may be distributed across different nodes, and is likely to span trust domains. As discussed in Section 2.2.8, queries to distributed state create read channels; Any security analysis that ignores these side channels will be unsound.

Accounting for the information flowing through dynamic authorization requests is important for the Fabric system implementation as well as for the language design, because the Fabric system itself performs dynamic authorization checks as part of its normal functioning. For example, when objects are requested from stores, the stores perform dynamic authorization checks to determine whether it is safe to return the object to the requesting worker.

Figure 4.1 depicts a situation where authorization requests can cause unsafe flows. Dynamically, w sends an authenticated message to store S_o requesting o. S_o must determine whether w is trusted to enforce the confidentiality of o. If the label on o is $\{p \rightarrow\}$, then $w \succcurlyeq L(o)$ exactly if $w \succcurlyeq p$. To determine whether this is true, S_o must read the object representing p, which may be stored on a different store, say S_p. Furthermore, p may indirectly delegate to w, through principals r or q; in this case S_r and S_q must be contacted as well. In order for the read to be considered safe, all of these stores must be trusted to learn about `secret`.

These concerns are not simply artifacts of the Fabric programming model or system design; these concerns arise in any context involving authorization queries

holding those facts, but does not model distributed proof construction.

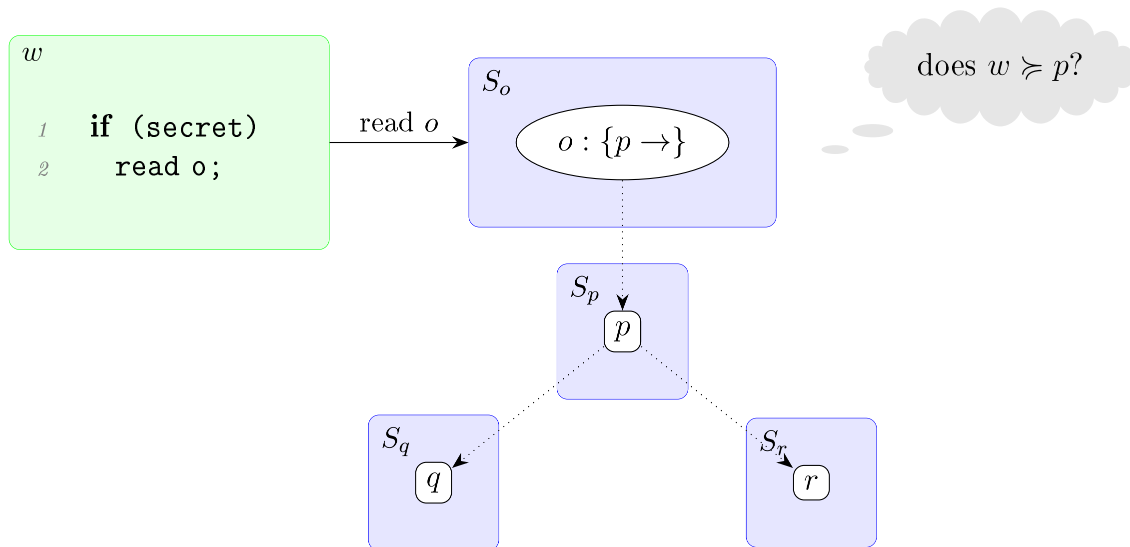

Figure 4.1: Example of read channels in dynamic authorization. To satisfy the request from w, store S_o must determine whether $w \succcurlyeq L(o) = \{p \rightarrow\}$, which may require it to fetch p and even p's delegates q and r.

between partially trusting entities. As a simple example, consider an authentication-as-a-service system such as OAuth [HL11]. The goal of these systems is to allow users to reuse existing accounts (such as Facebook or Google accounts) to identify themselves to third parties. This approach saves application providers from handling the tricky details of implementing authentication properly, while also reducing the proliferation of accounts that each user must manage.

Many users are justifiably reluctant to make use of these services, because they worry about the privacy implications [MH03b]. Some users do not want to allow the authentication providers to learn what websites they are logging into, when they log into them, or how often they do. Others are willing to allow these providers to learn that information, but are worried about that information leaking through the providers to their friends or the public at large.

In a setting like Fabric, where computations involve more complicated communication patterns and a finer-grained notion of trust and authority, side channels caused by authorization queries pose a greater risk. Untrusted parties can create programs that exploit the authorization side channel to leak arbitrary information.

In this chapter we build a formal model of a system and programming language that explicitly account for authorization query side channels. High-level authorization queries are not built into the language model; instead, we provide low-level primitives with types that explicitly account for these side channels. We then show that higher-level authorization queries can be implemented in this low-level language. The soundness of the type system then shows that the higher-level operations are safe.

Existing information flow analyses assume that the set of labels forms a lattice [MPP13]. In particular, given two labels ℓ_1 and ℓ_2, one can join them to form a third label $\ell_1 \sqcup \ell_2$ that is more restrictive than each of them. In a decentralized setting there may be no principal trusted to enforce the policy represented by $\ell_1 \sqcup \ell_2$. Data with such a label could be stored using replication [ZCMZ03], but we take a simpler approach: Our analysis does not assume that labels form a lattice; it merely assumes they form a preorder.

Revocation is an important consideration when considering distributed authorization queries [ALM15]. Our formal analysis follows previous information flow type systems by assuming that the principal hierarchy does not change during the execution of a program [HTHZ05, TZ07]. This assumption is a reasonable approximation of Fabric's transaction mechanism, which limits the side effects of a computation that

depends on revoked authority [SHTZ06]. We discuss the limits of this approximation and suggest a possible method for extending our analysis to handle revocation in Section 4.6.

4.1 System Overview

Our goal is to reason about the interplay between authorization and information flow. Any operation that causes information to flow from label ℓ to label m should only be executed if authorized by the owners of ℓ and m. Our calculus allows us to write programs that check whether these flows have been authorized while ensuring that the authorization query itself only causes authorized flows.

Before diving into the formal presentation of our language, we give an overview of its key design decisions.

Ensuring the integrity of the authorization state is crucially important, but is orthogonal to the problem of covert channels arising through authorization. Therefore, in this chapter we ignore integrity. Ignoring integrity permits a very simple label model: labels are just principals, and information can flow from the label corresponding to principal p to label represented by q whenever p delegates to q.

As in Fabric, principals are first-class objects: they have an identity encoded as a reference, and state defining the set of principals that they directly delegate to. The delegation relation is defined as the reflexive transitive closure of the direct delegation relation.

For simplicity, a principal can only directly delegate to either no principals, to all principals, or to exactly two principals. This is not a significant restriction: a principal can delegate to any finite set of trusted principals by creating a chain of intermediate principals, each delegating to one of the trusted principals and to another intermediate principal. Since delegation is transitive, directly delegating to the first intermediate principal causes indirect delegation to all of the other principals.

We are interested in capturing the information flows caused by dereferencing global references; we therefore explicitly model the distributed state of the system. Objects are stored by principals; we assume the principals that store an object are able to observe accesses to that object. For simplicity, we make no distinction between stores and principals.

The types of references include an access label that constrains the store holding the referenced objects. In addition, delegation sets require an access label that constrains the set of stores that may be contacted while determining the complete delegation set of that principal. This set includes the store holding the principal, as well as the stores of the principal's direct and transitive delegatees.

There is no primitive for querying the acts-for relation, since our goal is to show that such queries are well typed and therefore exhibit no unsafe flows. Instead, the program must compute the acts-for relation by directly traversing the principal data structures. This operation is most naturally expressed as a recursive function in a functional style, so we have chosen to model a functional language based on lambda calculus extended with references. Figure 4.2 shows the implementation of acts-for

```
1  actsfor  :=
2  λaq . λpr . λqr . case pr = qr of
3    | some feq → some (use feq in [pr ≽ qr])
4    | none     → let fqr : q = * qr with [aq ≼ aq] in
5      case fq : q of
6      | P  → some (use fqr, fq in [pr ≽ qr])
7      | ∅  → none
8      | {ar,br} → case (((actsfor
9                        aq with [aq ≼  ⊤ ])
10                       pr with [aq ≼  ⊤ ])
11                       ar with [aq ≼ aq]) of
12       | some fpa → some (use fqr, fq, fpa in [pr ≽ qr])
13       | none     →  case (((actsfor
14                           aq with [aq ≼  ⊤ ])
15                           pr with [aq ≼  ⊤ ])
16                           br with [aq ≼ aq]) of
17         | some fpb → some (use fqr, fq, fpb in [pr ≽ qr])
18         | none     →  none
```

Figure 4.2: Actsfor implementation. We have appended "r" to the names of references: pr, qr, ar and br. The function is parameterized by aq, the access label of the delegatee. The function returns some if pr acts for qr, and none otherwise. It begins by checking whether the two references are equal (line 2), returning true if they are. If not, it fetches q (line 4) and discriminates on the delegation set. In the case that q delegates to two principals (ar and qr), the function recursively compares pr to the two delegatees (lines 8 and 13).

in our calculus. Section 4.4 shows that the function is well typed, and therefore does not leak information.

The key to typing the encoding of acts-for in our model is that the type of the function requires a high pc. This requirement reflects the fact that programs cannot query the trust hierarchy willy-nilly, but must be careful to do so only in contexts that do not reveal secret information.

```
1 int {p→p} x;  int {q→q} y;
2 if (actsfor(p, q))
3   x = y;
```

Figure 4.3: An example Fabric program that uses dynamic acts-for checking

To show that programs in our language comport with policies, our static analysis must be able to connect the dynamic information that the acts-for function learns by traversing the data principal structures to the static information flow checks. For example, the Fabric program in Figure 4.3 is safe because the flow on line 3 only occurs if acts-for function returned true. The Fabric type system is able to do this kind of reasoning because the acts-for operation is built in, but our calculus needs a more general mechanism.

We accomplish this by allowing programs in our calculus to manipulate security proofs as first-class objects. Thus the acts-for function can return not only a boolean indicating whether one principal acts for another, but also a proof that it does; the type system can then reason that since the program was able to construct a proof that a flow is safe that the flow must actually be safe.

Although we avoid revocation for simplicity, we have made a number of design decisions that will make our language suitable for the integration of revocation. We avoid implicit coercion of values with less restrictive labels to values with more restrictive labels, because in the presence of revocation, labels that are more restrictive at one time may be less restrictive in the future. Instead, we require explicit coercion, thereby providing a concrete program point at which information flows between labels.

4.2 Language and System Model

We now turn to the formal model of our language and system. Our language has a number of moving parts, so for clarity we will introduce the syntax, operational semantics and typing feature by feature.

4.2.1 Standard Features

Figure 4.4 shows the standard features of our language. We start with a simply typed lambda calculus. We do not have booleans, opting instead for maybe types.

Everything shown in Figure 4.4 is completely standard, except that we carry some extra context for the features discussed below. Our language has references (discussed in §4.2.3 below), so our typing relation requires a heap context H to track the types of references and our reduction relation requires a representation s of the global state. Our language also keeps track of the side-effects of reads (§4.2.3), which requires a pc parameter to the typing judgment and a log of events $\vec{a}$ in the operational semantics. Finally, our language has a form of dependent types for reasoning about information flow (§4.2.4), for which we require additional annotations on function types and function applications; these annotations are explained in Section 4.2.4.

4.2.2 Principals and Delegation

Principals in our language are represented by references ℓ drawn from a set $LocVar$ of global locations. This choice is consistent with Fabric, which also represents principals as references to objects of class `Principal`.

$$
\begin{array}{rcl}
x \in \mathit{TermVar} & & \\
e, f \in \mathit{Expr} & ::= & x \mid () \mid \textbf{let}\, x = e \,\textbf{in}\, e' \mid \lambda x.e \mid e\, v \,\textbf{with}\, f \\
& & \mid \textbf{some}\, e \mid \textbf{none} \mid \textbf{case}\, e \,\textbf{of}\, \bullet\, \textbf{some}\, x \to e' \bullet\, \textbf{none}\, \to e'' \\
v, pc \in \mathit{Value} & ::= & x \mid () \mid \lambda x.e \\
\tau \in \mathit{IFType} & ::= & \textbf{Unit} \mid \textbf{Maybe}\, \tau \mid (x : \tau) \xrightarrow{pc} \tau' \\
E \in \mathit{EvalContext} & ::= & \textbf{let}\, x = [\circ] \,\textbf{in}\, e' \mid \textbf{some}\, [\circ] \mid [\circ] v \,\textbf{with}\, f \mid v\, v' \,\textbf{with}\, [\circ] \\
\Gamma \in \mathit{TypeContext} & ::= & \circ \mid \Gamma[x \mapsto \tau]
\end{array}
$$

Type system: $H; \Gamma; pc \vdash e : \tau$

$$\frac{\Gamma(x) = \tau}{H; \Gamma; pc \vdash x : \tau}\ \textsc{tsId} \qquad \frac{}{H; \Gamma; pc \vdash () : \textbf{Unit}}\ \textsc{tsUnit}$$

$$\frac{H; \Gamma[x \mapsto \tau]; pc \vdash e : \tau \quad H; \Gamma[x \mapsto \tau]; pc \vdash e' : \tau'}{H; \Gamma; pc \vdash \textbf{let}\, x = e \,\textbf{in}\, e' : \tau'}\ \textsc{tsLet}$$

$$\frac{H; \Gamma; pc \vdash e : \tau}{H; \Gamma; pc \vdash \textbf{some}\, e : \textbf{Maybe}\, \tau}\ \textsc{tsSome} \qquad \frac{}{H; \Gamma; pc \vdash \textbf{none} : \textbf{Maybe}\, \tau}\ \textsc{tsNone}$$

$$\frac{H; \Gamma; pc \vdash e : \textbf{Maybe}\, \tau \quad H; \Gamma[x \mapsto \tau]; pc \vdash e_1 : \tau' \quad H; \Gamma; pc \vdash e_2 : \tau'}{H; \Gamma; pc \vdash \textbf{case}\, e \,\textbf{of}\, \bullet\, \textbf{some}\, x \to e_1 \bullet\, \textbf{none}\, \to e_2 : \tau'}\ \textsc{tsMaybeCase}$$

Small-step operational semantics: $\langle \vec{a}, e, s \rangle \to \langle \vec{a}', e', s' \rangle$

$$\frac{\langle \vec{a}, e, s \rangle \to \langle \vec{a}', e', s' \rangle}{\langle \vec{a}, E[e], s \rangle \to \langle \vec{a}', E[e'], s' \rangle}\ \textsc{sosCtx} \qquad \frac{}{\langle \vec{a}, \textbf{let}\, x = v \,\textbf{in}\, e', s \rangle \to \langle \vec{a}, e'[x \mapsto v], s \rangle}\ \textsc{sosLet}$$

$$\frac{}{\langle \vec{a}, \textbf{case}\,\textbf{some}\, v \,\textbf{of}\, \bullet\, \textbf{some}\, x \to e_1 \bullet\, \textbf{none}\, \to e_2, s \rangle \to \langle \vec{a}, e_1[x \mapsto v], s \rangle}\ \textsc{sosCaseSome}$$

$$\frac{}{\langle \vec{a}, \textbf{case}\,\textbf{none}\,\textbf{of}\, \bullet\, \textbf{some}\, x \to e_1 \bullet\, \textbf{none}\, \to e_2, s \rangle \to \langle \vec{a}, e_2, s \rangle}\ \textsc{sosCaseNone}$$

Figure 4.4: Syntax and semantics for standard language features.

$$
\begin{array}{rll}
\ell \in \mathit{LocVar} & & \\
v \in \mathit{Value} & ::= \ldots & \mid \ell \mid \mathbb{P} \mid \emptyset \mid \{v_1, v_2\} \mid \top \mid \bot \\
e \in \mathit{Expr} & ::= \ldots & \mid \ell \mid \mathbb{P} \mid \emptyset \mid \{e_1, e_2\} \mid \top \mid \bot \\
 & & \mid \textbf{case}\, f : v \,\textbf{of}\, \bullet\, \mathbb{P} \to e_1 \bullet \emptyset \to e_2 \bullet \{x_1, x_2\} \to e_3 \\
\tau \in \mathit{IFType} & ::= \ldots & \mid \textbf{Del}\, A \mid \textbf{Prin}
\end{array}
$$

Type system: $H; \Gamma; pc \vdash e : \tau$

$$
\frac{}{H; \Gamma; pc \vdash \mathbb{P} : \textbf{Del}\, A}\ \textsc{tsAllPrin} \qquad \frac{}{H; \Gamma; pc \vdash \emptyset : \textbf{Del}\, A}\ \textsc{tsNoPrin}
$$

$$
\frac{H; \Gamma; pc \vdash e : \textbf{Del}\, A\, \textbf{Ref}\, @A \quad H; \Gamma; pc \vdash e' : \textbf{Del}\, A\, \textbf{Ref}\, @A}{H; \Gamma; pc \vdash \{e, e'\} : \textbf{Del}\, A}\ \textsc{tsMeet}
$$

$$
\frac{\begin{array}{l} H; \Gamma; pc \vdash v : \textbf{Del}\, A \qquad H; \Gamma[f \mapsto \textbf{Proof}\ v = \mathbb{P}]; pc \vdash e_1 : \tau' \\ H; \Gamma[f \mapsto \textbf{Proof}\ v = \emptyset]; pc \vdash e_2 : \tau' \\ H; \Gamma[f \mapsto \textbf{Proof}\ v = \{x_1, x_2\}][x_1 \mapsto \textbf{Del}\, A\, \textbf{Ref}\, @A][x_2 \mapsto \textbf{Del}\, A\, \textbf{Ref}\, @A]; pc \vdash e_3 : \tau' \end{array}}{H; \Gamma; pc \vdash \textbf{case}\, f : v\, \textbf{of}\, \bullet\, \mathbb{P} \to e_1 \bullet \emptyset \to e_2 \bullet \{x_1, x_2\} \to e_3 : \tau'}\ \textsc{tsDelCase}
$$

$$
\frac{}{H; \Gamma; pc \vdash \top : \textbf{Prin}}\ \textsc{tsTop} \qquad \frac{}{H; \Gamma; pc \vdash \bot : \textbf{Prin}}\ \textsc{tsBot}
$$

$$
\frac{H; \Gamma; pc \vdash e : \textbf{Del}\, A\, \textbf{Ref}\, @A}{H; \Gamma; pc \vdash e : \textbf{Prin}}\ \textsc{tsPrin}
$$

Small-step operational semantics: $\langle \vec{a}, e, s \rangle \to \langle \vec{a}', e', s' \rangle$

$$
\frac{}{\langle \vec{a}, \textbf{case}\, f : \emptyset\, \textbf{of}\, \bullet\, \mathbb{P} \to e_1 \bullet \emptyset \to e_2 \bullet \{x_1, x_2\} \to e_3, s \rangle \to \langle \vec{a}, e_2[f \mapsto \textbf{stmt}\, [\emptyset = \emptyset]], s \rangle}\ \textsc{sosCaseTop}
$$

$$
\frac{}{\langle \vec{a}, \textbf{case}\, f : \mathbb{P}\, \textbf{of}\, \bullet\, \mathbb{P} \to e_1 \bullet \emptyset \to e_2 \bullet \{x_1, x_2\} \to e_3, s \rangle \to \langle \vec{a}, e_1[f \mapsto \textbf{stmt}\, [\mathbb{P} = \mathbb{P}]], s \rangle}\ \textsc{sosCaseBot}
$$

$$
\frac{}{\begin{array}{l} \langle \vec{a}, \textbf{case}\, f : \{\ell_1, \ell_2\}\, \textbf{of}\, \bullet\, \mathbb{P} \to e_1 \bullet \emptyset \to e_2 \bullet \{x_1, x_2\} \to e_3, s \rangle \\ \quad \to \quad \langle \vec{a}, e_3[f \mapsto \textbf{stmt}\, [\{\ell_1, \ell_2\} = \{\ell_1, \ell_2\}]][x_1 \mapsto \ell_1][x_2 \mapsto \ell_2], s \rangle \end{array}}\ \textsc{sosCaseMeet}
$$

Figure 4.5: Syntax and semantics for language features for principals and delegation

$$\frac{}{s \vdash \ell \succcurlyeq \ell}\ \textsc{dynAfRefl} \qquad \frac{s \vdash \ell \succcurlyeq \ell' \quad s \vdash \ell' \succcurlyeq \ell''}{s \vdash \ell \succcurlyeq \ell''}\ \textsc{dynAfTrans}$$

$$\frac{\begin{array}{l} s(\ell) = \{\ell_1, \ell_2\} \\ s \vdash \ell' \succcurlyeq \ell_1 \end{array}}{s \vdash \ell' \succcurlyeq \ell}\ \textsc{dynMeetLeft} \qquad \frac{\begin{array}{l} s(\ell) = \{\ell_1, \ell_2\} \\ s \vdash \ell' \succcurlyeq \ell_2 \end{array}}{s \vdash \ell' \succcurlyeq \ell}\ \textsc{dynMeetRight}$$

$$\frac{}{s \vdash \ell' \succcurlyeq \bot}\ \textsc{dynBot} \qquad \frac{s(\ell') = \mathbb{P}}{s \vdash \ell \succcurlyeq \ell'}\ \textsc{dynAll}$$

Figure 4.6: Relation defining $\ell \preccurlyeq \ell'$.

Since principals are references, they must be explicitly dereferenced in order to determine delegation relationships. The typing constraints for references described in Section 4.2.3 prevent read channels for dereferencing in general. Therefore, the type system also rules out read channels caused by authorization queries.

Dereferencing a principal yields a delegation set, which is a value of type $\mathbf{Del}\, A$. As discussed in Section 4.1, a delegation set may be either the set of all principals $\mathbb{P}$, the set containing no principals $\emptyset$, or a set containing exactly two principals $\{\ell_1, \ell_2\}$. The A in $\mathbf{Del}\, A$ is an access label and will be discussed further in Section 4.2.3.

The case statement allows discrimination on delegation sets. Like function application, case statements must manipulate first-class proofs; we therefore defer discussion of the typing and semantics rules to Section 4.2.4.

Principals have type **Prin**. In addition to references to delegation sets, there are two special principals $\top$ and $\bot$. $\top$ and $\bot$, which are useful for specifying access labels. $\top$ and $\bot$ are not references; they contain no delegation state. By definition, $\top$ delegates to no principals (besides itself) while $\bot$ delegates to all principals.

Figure 4.6 contains the formal definition of the acts-for relation. The definition is parameterized on the state; the judgment has the form $s \vdash \ell_1 \succcurlyeq \ell_2$ where ℓ_1 and ℓ_2 are **Prin**s; it is simply a formalization of the reflexive transitive closure of the direct delegation relation, where a principal having the delegation set $\{\ell_1, \ell_2\}$ directly delegates to both ℓ_1 and ℓ_2.

4.2.3 Distributed State

Figure 4.7 shows the syntax and semantics for handling distributed state in our language. Global references ℓ are drawn from a set of location variables *LocVar*; these are intended to model Fabric OIDs.

Like Fabric OIDs, each reference is located on a store, and we assume that each store is operated by a principal. The function *storeof* maps locations to the principals holding those locations. For example, if *LocVar* is the set of Fabric OIDs, then *storeof*(`fab://store/onum`) would give `fab://store/0` since the principal representing the store is stored at onum 0 by convention.

The *new* expression is used to allocate new references; the store on which to create the object is explicitly specified. We assume that there is a deterministic allocation function **alloc** that gives a new reference on a store; we assume that the returned reference depends only on the portion of the state labeled ℓ. This requirement is why we use a cryptographically strong pseudorandom number generator for OID allocation in Fabric.

References are read using the $\textbf{let}\, f : x = *v\, \textbf{with}\, F\, \textbf{in}\, e$ syntax, which binds x to the value referenced by v and then evaluates the expression e. We use this

$$
\begin{array}{rcl}
\ell \in \mathit{LocVar} & & \\
e \in \mathit{Expr} & ::= \ldots & \mid \ell \mid \textbf{wrap}\, e\, \textbf{with}\, F \\
 & & \mid \textbf{new}\, e@v\, \textbf{with}\, F \mid \textbf{let}\, f : x = *v\, \textbf{with}\, F\, \textbf{in}\, e \\
\tau \in \mathit{IFType} & ::= \ldots & \mid \tau\, \textbf{Ref}\, @A \\
H \in \mathit{HeapContext} & ::= & \circ \mid H[\ell \mapsto \tau@A] \\
s \in \mathit{State} & ::= & \circ \mid s[\ell \mapsto v] \\
a \in \mathit{Event} & ::= & \textbf{read}\, \ell \mid \textbf{write}\, \ell\, v
\end{array}
$$

Type system: $H; \Gamma; pc \vdash e : \tau$

$$
\frac{H(\ell) = \tau@A}{H; \Gamma; pc \vdash \ell : \tau\, \textbf{Ref}\, @A}\ \textsc{tsLoc}
$$

$$
\frac{\begin{array}{l} H; \Gamma; pc \vdash e : \langle\tau\rangle_\ell \\ H; \Gamma; pc \vdash A : \textbf{Prin} \\ H; \Gamma; pc \vdash F : \textbf{Proof}\, pc \preccurlyeq A \end{array}}{H; \Gamma; pc \vdash \textbf{new}\, e@A\, \textbf{with}\, F : \langle\tau\rangle_\ell\, \textbf{Ref}\, @A}\ \textsc{tsNew}
$$

$$
\frac{\begin{array}{l} H; \Gamma; pc \vdash v : \tau\, \textbf{Ref}\, @A \\ H; \Gamma; pc \vdash F : \textbf{Proof}\, pc \preccurlyeq A \\ H; \Gamma[f \mapsto \textbf{Proof}\, x = *v][x \mapsto \tau]; pc \vdash e : \tau' \end{array}}{H; \Gamma; pc \vdash \textbf{let}\, f : x = *v\, \textbf{with}\, F\, \textbf{in}\, e : \tau'}\ \textsc{tsRead}
$$

$$
\frac{H; \Gamma; pc \vdash e : \tau\, \textbf{Ref}\, @A' \quad H; \Gamma; pc \vdash F : \textbf{Proof}\, A \preccurlyeq A'}{H; \Gamma; pc \vdash \textbf{wrap}\, e\, \textbf{with}\, F : \tau\, \textbf{Ref}\, @A}\ \textsc{tsWrap}
$$

Small-step operational semantics: $\langle \vec{a}, e, s\rangle \to \langle \vec{a}', e', s'\rangle$

$$
\frac{\ell' = \textbf{alloc}\,(s, \ell, A)}{\langle \vec{a}, \textbf{new}\, \langle v\rangle_\ell @A\, \textbf{with}\, F, s\rangle \to \langle \vec{a} \cdot \textbf{write}\, \ell'\, \langle v\rangle_\ell, \ell', s[\ell' \mapsto \langle v\rangle_\ell]\rangle}\ \textsc{sosNew}
$$

$$
\frac{s(\ell) = v}{\langle \vec{a}, \textbf{let}\, f : x = *\ell\, \textbf{with}\, F\, \textbf{in}\, e, s\rangle \to \langle \vec{a} \cdot \textbf{read}\, \ell, e[f \mapsto \textbf{stmt}\,[v = *\ell]][x \mapsto v], s\rangle}\ \textsc{sosRead}
$$

$$
\frac{}{\begin{array}{l} \langle \vec{a}, \textbf{let}\, f : x = *(\textbf{wrap}\, \ell'\, \textbf{with}\, F')\, \textbf{with}\, F\, \textbf{in}\, e, s\rangle \\ \quad \to \quad \langle \vec{a} \cdot \textbf{read}\, \ell', e[f \mapsto \textbf{stmt}\,[v = *\ell']][x \mapsto v], s\rangle \end{array}}\ \textsc{sosReadWrap}
$$

Figure 4.7: Syntax and semantics for language features for distributed state

extended syntax because we bind a proof f along with the value x that is returned. We defer the explanation of this proof term as well as the various *with* clauses until Section 4.2.4.

Our language does not allow the existing state to be mutated. Our analysis could be extended to reason about mutable state, but this would require keeping track of potential changes to the delegation state as the program executes. This analysis could be handled using a type-and-effect system or by modeling transactions in the operational semantics [SHTZ06], but we felt that this additional language complexity would obscure our handling of read channels and dynamic authorization.

The types of references include an access label A which specifies a bound on the store on which the referenced object resides. Our language avoids subtyping, so references to objects on stores must be explicitly wrapped to be typed with a different access label. The tsLoc rule enforces this constraint, while the sosWrap shows that wrapped references are treated the same way as unwrapped references.

Our goal with this language is to track the information that stores can learn by observing reads and writes to the references they store. To make this analysis easier, we record read and write events in a log $\vec{a}$ that is carried with the state of the system. The sosRead and sosNew rules update the log to record reads and writes respectively.

4.2.4 Dependent Types and Proofs

Figure 4.8 contains the syntax and semantics for the features used to manipulate first-class proofs. The types of proof objects describe the constraints that the proofs

$$
\begin{array}{rll}
e \in \textit{Expr} & ::= \ldots & \mid \mathbf{use}\, e_1, \ldots, e_N\, \mathbf{in}\, [c] \mid \mathbf{stmt}\, [c] \mid v = v' \\
\tau \in \textit{IFType} & ::= \ldots & \mid (x\!:\!\tau) \xrightarrow{pc} \tau' \mid \mathbf{Proof}\; c \\
E \in \textit{EvalContext} & ::= \ldots & \mid e\; v\; \mathbf{with}\, [\circ] \mid [\circ] v\; \mathbf{with}\, f \mid \mathbf{use}\, v_1, \ldots, v_i, [\circ], e_1, \ldots, e_j\, \mathbf{in}\, [c] \\
c \in \textit{Constraint} & ::= \ldots & \mid v = v' \mid v = *v' \mid v_1; \ldots; v_N \preccurlyeq v'_1; \ldots; v'_M
\end{array}
$$

Type system: $H; \Gamma; pc \vdash e : \tau$

$$\frac{H; \Gamma[x \mapsto \tau]; pc' \vdash e : \tau'}{H; \Gamma; pc \vdash \lambda x.e : ((x\!:\!\tau) \xrightarrow{pc'} \tau')} \; \textsc{tsLambda}$$

$$\frac{\begin{array}{c} H; \Gamma; pc \vdash e : ((x\!:\!\tau) \xrightarrow{pc'} \tau') \qquad H; \Gamma; pc \vdash v : \tau[x \mapsto v] \\ H; \Gamma; pc \vdash F : \mathbf{Proof}\; pc \preccurlyeq (pc'[x \mapsto v]) \end{array}}{H; \Gamma; pc \vdash e\; v\; \mathbf{with}\; F : \tau'} \; \textsc{tsApply}$$

$$\frac{\begin{array}{l} H; \Gamma; pc \vdash e_1 : \mathbf{Proof}\; c_1 \quad .. \quad H; \Gamma; pc \vdash e_N : \mathbf{Proof}\; c_N \\ c_1, \ldots, c_N \vdash c \end{array}}{H; \Gamma; pc \vdash \mathbf{use}\, e_1, \ldots, e_N\, \mathbf{in}\, [c] : \mathbf{Proof}\; c} \; \textsc{tsProof}$$

$$\frac{}{H; \Gamma; pc \vdash \mathbf{stmt}\, [c] : \mathbf{Proof}\; c} \; \textsc{tsStmt}$$

$$\frac{H; \Gamma; pc \vdash v_1 : \tau \quad H; \Gamma; pc \vdash v_2 : \tau}{H; \Gamma; pc \vdash v_1 = v_2 : \mathbf{Maybe\,Proof}\; v = v'} \; \textsc{tsEq}$$

Small-step operational semantics: $\langle \vec{a}, e, s \rangle \rightarrow \langle \vec{a}', e', s' \rangle$

$$\frac{}{\langle \vec{a}, (\lambda x.e)\; v\; \mathbf{with}\; F, s \rangle \rightarrow \langle \vec{a}, e[x \mapsto v], s \rangle} \; \textsc{sosApp}$$

$$\frac{}{\langle \vec{a}, v = v, s \rangle \rightarrow \langle \vec{a}, \mathbf{some\,stmt}\, [v = v], s \rangle} \; \textsc{sosEqTrue}$$

$$\frac{v! = v'}{\langle \vec{a}, v = v', s \rangle \rightarrow \langle \vec{a}', \mathbf{none}, s' \rangle} \; \textsc{sosEqFalse}$$

Figure 4.8: Language features for proofs and dependent types

witness; constraints c express the equality of two values ($v = v'$), the state of the global store ($v = *v'$), or a delegation relationship ($v \succcurlyeq v'$).

There are two types of proof values. Base proofs of the form $\textbf{stmt}\,[c]$ are intended to model self-validating statements about the state of the system. Statements of equality are easy to validate; statements of the form $\textbf{stmt}\,[v = *\ell]$ represent signed statements from the authoritative store of ℓ. Composite proofs have the form $\textbf{use}\, v_1, .., v_N\, \textbf{in}\,[c]$. These proofs are valid if the statements proved by the v_i imply c.

A key aspect of a well-formed configuration is that all of the assertions (base proofs) contained in it are valid. For this reason we disallow base proofs in the surface syntax; they are only introduced during the execution of the program. The proof of type preservation shows that this property continues to hold throughout execution.

Proofs are introduced by the dynamic tests of the claims that they witness. Statements of equality are introduced by the equality test in the sosEqTrue rule. This shows why we have chosen maybe types instead of booleans: equality tests return proofs in the positive case.

Similarly, statements about dereferenced locations are bound by the sosRead rule shown previously (Figure 4.7). These are intended to represent the signed packets that are returned from stores when objects are requested.

Statements about the state of the acts-for hierarchy are obtained by traversing delegation sets: the statement $\textbf{case}\, f : v\, \textbf{of}\, \bullet\, \mathbb{P} \rightarrow e_1 \bullet \emptyset \rightarrow e_2 \bullet \{x_1, x_2\} \rightarrow e_3$ binds f to a proof that the delegation set v has the appropriate value.

$$\frac{}{c_1,\,..\,,c_N \vdash c_j}\ \textsc{statId} \qquad \frac{c_1,\,..\,,c_N \vdash x' = *x \quad c_1,\,..\,,c_N \vdash x' = \mathbb{P}}{c_1,\,..\,,c_N \vdash y \succcurlyeq x}\ \textsc{statAllPrincipals}$$

$$\frac{c_1,\,..\,,c_N \vdash v = v'}{c_1,\,..\,,c_N \vdash v \succcurlyeq v'}\ \textsc{statAfRefl} \qquad \frac{c_1,\,..\,,c_N \vdash v \succcurlyeq v' \quad c_1,\,..\,,c_N \vdash v' \succcurlyeq v''}{c_1,\,..\,,c_N \vdash v \succcurlyeq v''}\ \textsc{statAfTrans}$$

$$\frac{\begin{array}{l} c_1,\,..\,,c_N \vdash x' = *x \\ c_1,\,..\,,c_N \vdash x' = \{x_1, x_2\} \\ c_1,\,..\,,c_N \vdash y \succcurlyeq x_1 \end{array}}{c_1,\,..\,,c_N \vdash y \succcurlyeq x}\ \textsc{statMeetLeft} \qquad \frac{\begin{array}{l} c_1,\,..\,,c_N \vdash x' = *x \\ c_1,\,..\,,c_N \vdash x' = \{x_1, x_2\} \\ c_1,\,..\,,c_N \vdash y \succcurlyeq x_2 \end{array}}{c_1,\,..\,,c_N \vdash y \succcurlyeq x}\ \textsc{statMeetRight}$$

Figure 4.9: The static implication relation $c_1,\,..\,,c_N \vdash c$. The compound constraint $v_1; \ldots; v_N \preccurlyeq v'_1; \ldots; v'_M$ is equivalent to the $N \cdot M$ constraints $v_i \preccurlyeq v'_j$. Rules for symmetry, transitivity and reflexivity of $=$ as well as for deconstructing compound acts-for constraints are elided.

Proofs can be combined using the **use** construct, which creates a proof of a derived constraint by combining proofs of the premises. For example, a program can combine proofs that $d = *q$ and $d = \mathbb{P}$ to conclude that $q \preccurlyeq p$ for any p, since q's delegation set says it delegates to all principals. The rules for combining constraints are given in Figure 4.9.

Proofs terms must be provided by the programmer whenever the safety of a statement relies on a delegation relationship. This is the purpose of the **with** clauses on function application (which may raise the pc), allocation and dereferencing (which may leak the pc to the store holding the object being fetched), and wrapped pointers (which must prove that the actual store receiving the information is trusted to enforce the access label).

4.2.5 Labeled Values and Relabeling

The types we have seen so far do not describe confidential values—indeed none of the types described so far have information flow labels. We have been able to encapsulate the features for information flow tracking in one place by adapting the security monad design from the Dependency Core Calculus [ABHR99] to our setting. Figure 4.10 shows the relevant parts of the syntax and semantics.

Bracketed values $\langle v \rangle_\ell$ represent labeled information; they have types that are also bracketed. In order to associate a single label to each part of the state, we prevent nested brackets. To achieve this goal, we do not allow users to insert brackets into their programs directly; instead they must use the **return** construct[2].

The $\mathbf{bind}_{L'}\ x = e\ \mathbf{with}\ F\ \mathbf{in}\ e'$ syntax is used to extract values out of brackets. In this expression, e has a bracketed type, so dynamically it will produce a bracketed value. Within e', the variable x will be bound to the (unbracketed) value produced by e. It can therefore use this value in case statements, function calls, or any of the other expressions that operate over unbracketed values. The catch is that the pc is raised to L' when type checking e', so all of the side-effects of operations that make use of x must be above level L'. Similarly, the result of evaluating e' must also be labeled L'. To ensure that these flows are safe, the programmer must supply a proof F that both the pc and L can flow to L'.

The premise of the sosBind rule deserves mention. Instead of simply performing a substitution of x in e', the bind statement executes e' completely in a single step.

[2]The choice of the keywords **bind** and **return** reflect the fact that these operations almost form a *monad*, a standard design pattern used in functional programming. We say almost because the double-bracketing restriction prevents the monadic functor from being a total function.

$$
\begin{aligned}
e \in \mathit{Expr} \quad &::= \ldots \quad \mid \mathbf{bind}_\ell\ x = e\ \mathbf{with}\ F\ \mathbf{in}\ e' \mid \mathbf{return}_\ell\ e \mid \langle v \rangle_\ell \\
\tau \in \mathit{IFType} \quad &::= \ldots \quad \mid \langle \tau \rangle_\ell
\end{aligned}
$$

Type system: $H; \Gamma; pc \vdash e : \tau$

$$
\frac{\begin{array}{l} H; \Gamma; pc \vdash e : \langle \tau \rangle_L \\ H; \Gamma; pc \vdash L' : \mathbf{Prin} \\ H; \Gamma; pc \vdash F : \mathbf{Proof}\ pc; L \preccurlyeq L' \\ H; \Gamma[x \mapsto \tau]; L' \vdash e' : \langle \tau' \rangle_{L'} \end{array}}{H; \Gamma; pc \vdash \mathbf{bind}_{L'}\ x = e\ \mathbf{with}\ F\ \mathbf{in}\ e' : \langle \tau' \rangle_{L'}}\ \textsc{tsBind}
$$

$$
\frac{H; \Gamma; pc \vdash e : \tau}{H; \Gamma; pc \vdash \mathbf{return}_L\ e : \langle \tau \rangle_L}\ \textsc{tsReturn}
$$

$$
\frac{H; \Gamma; pc \vdash v : \tau \quad H; \Gamma; pc \vdash \ell : \mathbf{Prin}}{H; \Gamma; pc \vdash \langle v \rangle_\ell : \langle \tau \rangle_\ell}\ \textsc{tsProtect}
$$

Small-step operational semantics: $\langle \vec{a}, e, s \rangle \to \langle \vec{a}', e', s' \rangle$

$$
\frac{\langle \vec{a}, e'[x \mapsto v], s \rangle \to^* \langle \vec{a}', v', s' \rangle}{\langle \vec{a}, \mathbf{bind}_{\ell'}\ x = \langle v \rangle_\ell\ \mathbf{with}\ F\ \mathbf{in}\ e', s \rangle \to \langle \vec{a}', v', s' \rangle}\ \textsc{sosBind}
$$

$$
\frac{}{\langle \vec{a}, \mathbf{return}_\ell\ v, s \rangle \to \langle \vec{a}, \langle v \rangle_\ell, s \rangle}\ \textsc{sosReturn}
$$

Figure 4.10: Language features for information flow

This choice enables us to state our definition of flow in terms of single steps in the operational semantics. In a sense, this encodes our decision to ignore timing channels—the number of steps can be thought of as a crude measure of execution time; by making all computations that can observe labeled values execute in a single step, we clobber this timing information. This is the same approach taken in the definition of ($\xrightarrow{\Lambda}\!\!\!\!\rightarrow$) in Chapter 3.

4.3 Security Condition

With our language fully defined, we now turn to the security properties provided by the type system. Typically, information flow security is measured by some form of noninterference, which informally says that if an attacker is not allowed to distinguish between two inputs to a program, then they should not be able to distinguish the between the outputs of the program. This perspective divides the state of the system into things that the attacker is allowed to learn (the public state) and things the attacker is not allowed to learn (the private state).

This perspective assumes that we can ignore flows that are compatible with the flows-to relation, and then proves that there are no (non-ignored) flows. In our setting, the question of whether a principal is allowed to learn something depends on the state of the program. Fortunately, the flow model defined in Chapter 3 is well suited to our task, because it makes all flows explicit, and then allows us to characterize whether flows are safe or unsafe in the environment in which they occur.

In this section we apply the abstract definitions from Chapter 3 to our language model, and prove that there are no unsafe flows. Our language does not include primitives for downgrading, so we do not consider influenced flows.

4.3.1 Definitions

Recall the definition of flow from Chapter 3: we say that information *flows* from h to k if we can start with a configuration C_1 which steps to C_1', alter only the "h part" of C_1 to form C_2, and step to a new configuration C_2' which differs in the "k part". We refer to the 4 configuration $C_i^{(\prime)} = (C_1, C_1', C_2, C_2')$ as the *witness* to the flow. This is summarized in the formal definition:

Definition 4.1 (Flow). *h flows to k with witness $C_i^{(\prime)}$ (written $h \;{}^{C_1}_{C_2}{\leadsto}^{C_1'}_{C_2'}\; k$) if*

- $C_i \to C_i'$,
- *C_1 and C_2 differ* only *at h, and*
- *C_1' and C_2' differ at k.*

This definition requires us to define what it means for two configurations to agree (or differ) at a given label.

Definition 4.2. *(configuration agreement at ℓ). We say that two configurations $C_1 = \langle \vec{a}_1, e_1, s_1 \rangle$ and $C_2 = \langle \vec{a}_2, e_2, s_2 \rangle$* agree at *$\ell$ (written $C_1 =_\ell C_2$), if $\vec{a}_1$ and $\vec{a}_2$ agree at ℓ, s_1 and s_2 agree at ℓ, and e_1 and e_2 agree at ℓ.*

For memories this is easy to define, because each location ℓ has an associated label:

Definition 4.2′. *(memory agreement at ℓ). We say that states s_1 and s_2* agree at ℓ *(written $s_1 =_\ell s_2$) if for all locations ℓ' such that $\ell = labelof(\ell')$, $s_1(\ell') = s_2(\ell')$.*

Similarly, each ℓ' has an associated store that is able to see reads to ℓ', so we can define equality of event streams:

Definition 4.2″. *(event sequence agreement at ℓ). Two sequences of events α_1^* and α_2^* agree at ℓ (written $\alpha_1^* =_\ell \alpha_2^*$) if restricting the α_i^* to reads of locations ℓ' with $store - of\ \ell' = \ell$ yields the same sequences.*

The difficult part of the definition is the agreement of two programs at label ℓ. To allow us to reason about programs that are indistinguishable at a given level, we introduce bracketed values $\langle v \rangle_\ell$. Bracketed values are considered equal to all labels other than ℓ:

Definition 4.2‴. *(expression agreement at ℓ). Two bracketed values $\langle v_1 \rangle_{\ell'}$ and $\langle v_2 \rangle_{\ell'}$ are equal at level ℓ if $\ell' \neq \ell$ or if $v_1 =_\ell v_2$. Other expressions are equal at level ℓ if they are structurally similar and their corresponding subexpressions are equal at level ℓ.*

This notion of agreement at ℓ is different from the usual notion of low-equivalence commonly used to define noninterference. Low-equivalence requires agreement at all labels *lower than* ℓ, whereas our definition requires agreement only at ℓ itself. The reason for this choice is that we consider the label order to be ephemeral, and consider both safe and unsafe flows to be important.

Our definition of flow is stated in terms of single steps in the operational semantics, whereas typical definitions of noninterference allow one of the two configurations to take an arbitrary number of steps to "catch up" to the other. Consider the following program for example:

```
1 P :=  if secret
2  then let x = () in ()
3  else ()
```

Regardless of the value of `secret`, this program evaluates to (), but it may take different numbers of steps to do so. We are deliberately ignoring timing channels, so we want these two evaluations to look equivalent.

Our approach is to set up the semantics so that computations that branch on high data always appear to complete in a single step. This approach is encoded in the sosBind rule: a configuration that binds a value completes the entire high context in a single step.

With the definition of a flow in hand, we can now describe when flows are safe:

Definition 4.3. *(safe flows). We say a flow* $h \; {}^{C_1}_{C_2}{\rightsquigarrow}^{C_1'}_{C_2'} \; k$ *is* safe *if* $C_i^{(\prime)} \vdash h \preccurlyeq k'$*.*

Note that there are four possible configurations in which to evaluate $\ell \preccurlyeq \ell'$ (C_1, C_2, C_1' and C_2'), and we require the delegation relationship to hold in all four of them.

Our security theorem will say that well-formed configurations do not exhibit unsafe flows. This requires a definition of well-formedness. Naturally, well-formed programs must be well typed. In addition, we require that all of the assertions claimed by the program must be true, and that there are no double-bracketed values. These requirements are summarized in the following definition:

Definition 4.4. *(well-formed configuration). We say that a configuration* $C = \langle \vec{a}, e, s \rangle$ *is* well-formed*, written* $pc \vdash C : \tau$ *if there exists some heap context* H *such that*

- $H; \circ; pc \vdash e : \tau$

- *$H \vdash s$, and*
- *all of the statements in both e and s are satisfied by s.*

These extra assumptions are trivially satisfied by programs that a user writes, as we disallow both base statements and bracketed terms. However, we need these conditions to reason about partially evaluated programs.

4.3.2 Proof of Security

We now present a sketch of the proof that our type system enforces security as defined above.

Theorem 4.5. *(Small-step security) Suppose that we have a flow $h \; {}^{C_1}_{C_2}\!\!\rightsquigarrow^{C_1'}_{C_2'} \; k$, and that for some pc, $pc \vdash C_i : \tau$. Then $C_i^{(\prime)} \vdash h \preccurlyeq k$.*

Proof. It suffices to show that $C_1 \vdash h \preccurlyeq k$. $C_2 \vdash h \preccurlyeq k$ follows by symmetry, while $C_i' \vdash h \preccurlyeq k$ follows from the following lemma:

Lemma 4.6. *(Acts-for is preserved) If $C \rightarrow C'$ then $C \vdash h \preccurlyeq k$ implies $C' \vdash h \preccurlyeq k$. Further, if h and k are bound in the state of C, then the converse also holds.*

Proof. This lemma is where we exploit the lack of mutation, as discussed in Section 4.2.3. It follows trivially by induction on the derivation of $C \rightarrow C'$. □

By the definition of flow, we know that $C_1 \rightarrow C_1'$; We proceed by induction on this derivation. We will generalize our inductive hypothesis somewhat: instead of assuming that C_1 and C_2 differ only on a single label h, we allow them to differ on a

set of labels H. We show that there is at least one label $h \in H$ for which $C_1 \vdash h \preccurlyeq k$. The claimed theorem then follows from the case where $H = \{h\}$.

Let $\overline{h}$ be an arbitrary label not in H (the theorem is trivial if no such $\overline{h}$ exists); by the definition of flow, we know that $C_1 =_{\overline{h}} C_2$, and by inspection we can conclude that the derivation $C_2 \to C_2'$ uses the same rule as the derivation $C_1 \to C_1'$.

We now consider the reduction rules in turn:

Context rule We have the following situation:

$$\frac{(\widehat{C}_1 :=) \quad \langle \vec{a}_1, \widehat{e}_1, s_1 \rangle \to \langle \vec{a}_1', \widehat{e}_1', s_1' \rangle \quad (:= \widehat{C}_1')}{(C_1 :=) \quad \langle \vec{a}_1, E_1[\widehat{e}_1], s_1 \rangle \to \langle \vec{a}_1', E_1[\widehat{e}_1'], s_1' \rangle \quad (:= C_1')} \text{ sosContext}$$

$$\frac{(\widehat{C}_2 :=) \quad \langle \vec{a}_2, \widehat{e}_2, s_2 \rangle \to \langle \vec{a}_2', \widehat{e}_2', s_2' \rangle \quad (:= \widehat{C}_2')}{(C_2 :=) \quad \langle \vec{a}_2, E_2[\widehat{e}_2], s_2 \rangle \to \langle \vec{a}_2', E_2[\widehat{e}_2'], s_2' \rangle \quad (:= C_2')} \text{ sosContext}$$

We must check the following fact about evaluation contexts:

Lemma 4.7. *(ℓ-equal evaluation contexts) If $E_1[e_1] =_\ell E_2[e_2]$ then $e_1 =_\ell e_2$ and for all $e_1' =_\ell e_2'$, we have $E_1[e_1'] =_\ell E_2[e_2']$.*

Proof. Straightforward by induction on the structure of E. □

This lemma tells us that for all $\overline{h} \notin H$, $\widehat{C}_1 =_{\overline{h}} \widehat{C}_2$. Moreover, $\widehat{C}_1 \neq_k \widehat{C}_2$. We can therefore apply the inductive hypothesis to conclude that $\widehat{C}_1 \vdash h \preccurlyeq k$ for some $h \in H$. Since C_1 and $\widehat{C}_1$ have the same state, we can conclude that $C_1 \vdash h \preccurlyeq k$ as required.

Straightforward cases The proof for the non-inductive rules (App, Let, New, Read, EqTrue, EqFalse, and the Case rules) all follow immediately. As an

example, we give the details for the Let rule. We have the following situation:

$$\frac{}{(C_1 :=) \quad \langle \vec{a}_1, \textbf{let}\; x = v_1 \textbf{ in } e_1', s_1 \rangle \to \langle \vec{a}_1, e_1'[x \mapsto v_1], s_1 \rangle \quad (:= C_1')} \textsc{sosLet}$$

$$\frac{}{(C_2 :=) \quad \langle \vec{a}_2, \textbf{let}\; x = v_2 \textbf{ in } e_2', s_2 \rangle \to \langle \vec{a}_2, e_2'[x \mapsto v_2], s_2 \rangle \quad (:= C_2')} \textsc{sosLet}$$

Since $C_1 =_{\overline{h}} C_2$, we know that $\vec{a}_1 =_{\overline{h}} \vec{a}_2$, $v_1 =_{\overline{h}} v_2$, $e_1' =_{\overline{h}} e_2'$ and $s_1 =_{\overline{h}} s_2$. From this we conclude that $e_1'[x \mapsto v_1] =_{\overline{h}} e_2'[x \mapsto v_2]$, and thus $C_1' =_{\overline{h}} C_2'$ for all $\overline{h} \notin H$. Since $C_1' \neq_k C_2'$, it must be the case that $k \in H$. By reflexivity of acts-for, $C_1 \vdash k \preccurlyeq k$ so choosing $h := k$ completes this case.

The other straightforward cases have the same structure: show that $C_1' =_{\overline{h}} C_2'$ and then choose $h := k$ using reflexivity.

Bind This is the interesting case. We have

$$\frac{(\widehat{C}_1 :=) \qquad \langle \vec{a}_1, \widehat{e}_1[x \mapsto v_1], s_1 \rangle \to^* \langle \vec{a}_1', v_1', s_1' \rangle \qquad (:= \widehat{C}_1')}{(C_1 :=) \quad \langle \vec{a}_1, \textbf{bind}_{\ell'}\; x = \langle v_1 \rangle_\ell \textbf{ with } f \textbf{ in } \widehat{e}_1, s_1 \rangle \to \langle \vec{a}_1', v_1', s_1' \rangle \quad (:= C_1')} \textsc{sosBind}$$

$$\frac{(\widehat{C}_2 :=) \qquad \langle \vec{a}_2, \widehat{e}_2[x \mapsto v_2], s_2 \rangle \to^* \langle \vec{a}_2', v_2', s_2' \rangle \qquad (:= \widehat{C}_2')}{(C_2 :=) \quad \langle \vec{a}_2, \textbf{bind}_{\ell'}\; x = \langle v_2 \rangle_\ell \textbf{ with } f \textbf{ in } \widehat{e}_2, s_2 \rangle \to \langle \vec{a}_2', v_2', s_2' \rangle \quad (:= C_2')} \textsc{sosBind}$$

Now, there are two possibilities: either $\ell \in H$ or $\ell \notin H$. If $\ell \notin H$ then we know $\langle v_1 \rangle_\ell =_\ell \langle v_2 \rangle_\ell$ so $v_1 =_\ell v_2$. Therefore $\widehat{e}_1[x \mapsto v_1] =_\ell \widehat{e}_2[x \mapsto v_2]$, and thus $\widehat{C}_1 =_\ell \widehat{C}_2$. This implies that $\widehat{C}_1' =_\ell \widehat{C}_2'$, so that $C_1' =_\ell C_2'$. As in the sosLet case above, we see that $k \in H$ and $C_1 \vdash k \preccurlyeq k$, as required.

The other possibility is that $\ell \in H$. In that case, v_1 and v_2 may be completely unrelated. However, we will now show that the fact that these differences influence the output at level k implies that $C_1 \vdash \ell \preccurlyeq k$, so that we can choose $h := \ell$.

Since $C_1' \neq_k C_2'$, we know that either (1) $v_1' \neq_k v_2'$, (2) $\vec{a}_1' \neq_k \vec{a}_2'$, or (3) $s_1' \neq_k s_2'$. In all three cases, we will use the facts that $C_1 \vdash pc \preccurlyeq \ell'$ and $C_1 \vdash \ell \preccurlyeq \ell'$, which come from the existence of the proof object $f : \mathbf{Proof}\, pc; \ell \preccurlyeq \ell'$.

In case (1), note that values v_1' and v_2' both have type $\langle \tau \rangle_{\ell'}$, and thus have the forms $\langle \widehat{v}_1' \rangle_{\ell'}$ and $\langle \widehat{v}_2' \rangle_{\ell'}$. This means that $v_1' = \langle \widehat{v}_1' \rangle_{\ell'} \neq_k \langle \widehat{v}_2' \rangle_{\ell'} = v_2'$, which is only possible if $k = \ell'$. Since $C_1 \vdash \ell \preccurlyeq \ell'$, we have $C_1 \vdash \ell \preccurlyeq k$ as required.

In cases (2) and (3), the evaluation of one of $\widehat{e}_1$ or $\widehat{e}_2$ must have caused side effects at k. However, we know that $\widehat{e}_1$ and $\widehat{e}_2$ are both well-typed with pc ℓ', which intuitively means that evaluating them should only have side effects at labels above ℓ'. Thus, the side effects at k should mean that $C_1 \vdash \ell' \preccurlyeq k$. Combining this with the fact that $C_1 \vdash \ell \preccurlyeq \ell'$ gives $C_1 \vdash \ell \preccurlyeq k$ as required.

The intuition in the preceding paragraph is justified by the following lemma:

Lemma 4.8. *(No low side-effects) Suppose that $pc \vdash C$ and that $C \to^* C'$. Then if the store of C' differs from the store of C at label k then $C \vdash pc \preccurlyeq k$. Similarly, if the trace of C' differs from the trace of C' on k then $C \vdash pc \preccurlyeq k$.*

Proof. Induction on the length of $C \to^* C'$ and the derivation of $C \to C'$. □

This concludes the proof of Theorem 4.5. □

4.4 Actsfor Revisited

The implementation of acts-for in figure 4.2 has the type

$$(aq\!:\!\mathbf{Prin}) \xrightarrow{\top} (pr\!:\!\mathbf{Prin}) \xrightarrow{\top} (qr\!:\!\mathbf{Del}\, aq\, \mathbf{Ref}\ @aq) \xrightarrow{aq} \mathbf{Maybe}\, \mathbf{Proof}\, pr \succcurlyeq qr$$

According to theorem 4.5, this means that if executed, the function produces no side effects below aq. However, it does require that the caller provide a proof that the caller's pc may flow to aq.

This implementation of acts-for has a few shortcomings. In particular, while our definitions allow circular delegation relations, the function shown will loop infinitely if executed on a principal with a cyclic delegation structure. Because acts-for is not primitive, implementing a search procedure that keeps track of visited principals is a simple matter of programming; the correctness of the implementation follows from the typing. Fabric's implementation of acts-for is even more complicated, because it performs caching of intermediate results and reuses facts it has already derived. Although we have not done this exercise, our calculus should be sufficiently expressive to encode such a function.

Another possible extension would be to allow dynamic exploration of the acts-for hierarchy. In our current formulation, the access label on principal objects is statically determined, and principals are not able to delegate to principals on more public stores. An alternative approach would be to supply the program counter label to the acts-for implementation, and allow it to dynamically explore as much of the hierarchy as it can while avoiding unsafe read channels. This would require extending our calculus to provide a way to dynamically determine the store of a reference, but this should not present any fundamental difficulty. The advantage to this approach is that acts-for could be queried in more circumstances, but the result it returns would only be a conservative approximation of the true acts-for relation. Exploring the limitations of these contrasting approaches is planned for future work.

```
1 final Label l1, l2;
2 if (l1 ⊑ l2) { /* l1 ⊑ l2 is assumed here */ }
3
4 final Principal p1, p2;
5 if (p1 ≼ p2) { /* p1 ≼ p2 is assumed here */ }
```

Figure 4.11: Dynamic label and principal tests in Fabric.

4.5 Lessons for Fabric

In this section we discuss the ramifications of this Chapter's formal development on the Fabric system. The most direct impact will be on the typing of the dynamic label and principal comparison operations. Fabric contains primitive syntax for dynamically comparing labels and principals, shown in Figure 4.11.

These constructs are translated to Java by generating calls to the `LabelUtil.flowsTo` and `PrincipalUtil.delegatesTo` methods respectively. These methods, which are implemented in Java, are responsible for traversing the hierarchy defined by the `Principal.delegatesTo` methods.

In order to properly account for read channels through delegation, we must impose additional constraints on the use of these constructs. In our formal model, we parameterized the **Del** type with an access label A; in Fabric we should parameterize the `Principal` and `Label` classes with an access label. This can be handled using Fabric's existing support for parameterized classes. In our formal model, the type for `actsfor` required the caller's pc to flow to the access label of the delegator; the corresponding requirement in Fabric will be to require the pc at a dynamic check to flow to the access label of the delegator.

In fact we could go a step further by porting the Fabric implementation of the `PrincipalUtil.delegatesTo` and `LabelUtil.flowsTo` functions into the Fabric language, and type checking the dynamic delegation primitives as if they were method invocations. This approach would ensure that we do not make security-critical mistakes while implementing them.

More generally, we believe the approach used in this formal development can be applied more broadly to the analysis of the Fabric system. In particular, by implementing high-level Fabric primitives in lower-level calculi with information flow tracking, we can effectively reason about the information flow behavior of our system implementation. In the limit, we could implement the Fabric runtime system in a language like Jif, which would be an interesting research project in its own right.

First-class proofs are not necessary to fix the dynamic authorization side channel in Fabric, but they would be useful to add for a number of other reasons. The most natural way to integrate them into the Fabric programming model would be as representation invariants on Fabric classes. Instances of these classes could then be used to tie together a collection of principals and other objects and act as a witness that the expected relationships between those principals hold.

To see the benefits of this feature, consider the airline example from Chapter 2. In that example, there were principals representing an airline, a Broker, and a customer, and we assumed that the airline and customer both delegated to the Broker. In the implementation, each of the methods that made use of these assumptions had to use **where** clauses indicating the assumptions, and these where clauses had to be threaded through all of the implementation methods. Similarly, the FriendMap

```
1  class ProblemInstance where airline ≼ broker, customer ≼ broker {
2    public final Airline airline;
3    public final Broker  broker;
4    public final Customer customer;
5
6    public ProblemInstance(Airline a, Broker b, Customer c)
7    throws AssertionFailedException
8    {
9      assert (a ≼ b); assert (c ≼ b);
10   }
11 }
12
13 void runAuction(nonnull ProblemInstance pi) {
14   /* this code can assume that pi.airline ≼ pi.broker */
15 }
```

Figure 4.12: Broker example using class **where** constraints.

application discussed in Chapter 2 was factored into a number of methods for adding a pin to a map, for creating a public map with a given set of users, for creating a private map, and so on. Each of these methods required explicit declarations of the expected relationships, leading to verbose and redundant annotations.

With representation invariants, we could encode these expectations into a class which checks the invariants in the constructor. The methods that rely on the relationships can then replace the list of where constraints with a single argument representing the problem instance. In the airline example, we would create a `ExampleInstance` class as shown in Figure 4.12. The compiler ensures that the assertions in the **where** clauses hold at the end of the constructor invocation, allowing the `runAuction` method to assume them.

These features should be easy to implement by building on features already present in Jif and Fabric. In particular, the Jif compiler already performs static analysis to determine whether references may be **null**, and they already perform static information flow analysis relative to an environment of known label and principal relationships. Adding class invariants should be a matter of extending the environment used for label checking with facts taken from the classes of in-scope non-null references. However, implementation and evaluation of these designs is left for future work.

First class proof terms can also inform the runtime implementation of systems like Fabric. Fabric stores already distribute signed object groups to workers and the dissemination layer. These are very similar to the $\textbf{stmt}\,[v = *v']$ objects that are produced by **read**s in our calculus. Thinking of these as first-class proof objects can simplify the way we reason about them. This idea is closely related to current work on warranties [LMA+14]; we expect that the implementation of first-class proofs in Fabric will mesh elegantly with ongoing efforts to implement warranties in Fabric.

4.6 Revocation

Our soundness result depends on the assumption that when the program reads a reference, the value that is read does not subsequently change. Some kind of consistency constraint is necessary to rule out an execution in which a delegation set is read by a worker w, then is changed, and then w performs an action based on the stale delegation set. This would result in an impermissible flow, and is an example of a “time-of-check/time-of-use” (TOCTOU) vulnerability.

The consistency constraint we have used in our model is extremely strong—that once something is read it never changes. This choice is inspired by the transaction abstraction, in which programs may operate under the illusion that they are completely isolated from all concurrent mutations, and the runtime system is responsible for resolving conflicts between transactions. This approach is somewhat justified by the results in [SHTZ06], which proves a noninterference result in a language with mutable delegation using an explicit transaction system.

However, just as the metatheory in prior work has hidden the information flows required to implement dynamic delegation checks, we have masked the information flows required to implement transactions by making this strong transactional assumption. Although we leave formalization of information flow through transaction implementations to future work, in this section we will pull back the curtain somewhat and examine some of the issues at a high level.

Consider an optimistic transaction implementation such as Fabric's (see Chapter 2). Optimistic concurrency control isolates transactions by speculatively executing them and rolling them back if they cause conflicts. Although this scheme keeps the persistent data consistent, it does not prevent information leaks caused by inconsistent trust configuration: the transaction will only be rolled back when it tries to commit, but it may communicate sensitive information before that. Put another way, communication with untrusted nodes is an external action that cannot be rolled back, and such actions are incompatible with optimistic concurrency control.

Since the problem with the optimistic approach is that we may perform external actions before checking consistency, we may consider performing a consistency check

before each externally visible action. This approach fails for two reasons. The first is that performing these checks may introduce new read channels: the fact that the consistency check is being made indicates that an externally visible action is about to happen.

The second problem with performing extra consistency checks prior to performing externally visible actions is that they do not solve the TOCTOU vulnerability. Although it would require delegation *immediately before* an information flow, it does not require delegation *while* the action is being performed. While this may be "good enough" for many applications, it is hard to characterize exactly what security property is offered. The root of the problem is that reading a delegation object gives a snapshot at a zero-dimensional point in time while performing actions that may leak information takes a one-dimensional range of time.

An alternative approach to consistency control is to use pessimistic concurrency control, or locking. Under this approach, objects are locked whenever they are read, and the stores reject any conflicting updates until the lock is released. Similarly, if one is willing to assume that nodes have access to synchronized or loosely synchronized clocks, then it is possible to issue warranties [LMA+14] that guarantee that the value of an object will not change for a certain time period after it is read. Either of these approaches give a worker certainty about the delegation state if it has read a delegation set sufficiently recently.

The problem with locks, leases, and warranties in our setting is that they introduce additional covert channels. A worker should be able to fetch an object from a highly trusted store in a high context, even if the object itself is public. However, if

reading the object causes it to be locked or to have a promise generated for it, then untrusted workers can learn that the object is being accessed by observing whether updates to that object are accepted.

One way to prevent these channels from leaking information is to make their effects depend only on public information. For example, instead of generating leases on an as-needed basis, a store could adopt a public and fixed schedule of when updates are allowed to occur. As long as a worker is executing within one of the epochs in which delegation information will not change, it can safely assume that the values it has read are consistent, but untrusted hosts can not learn anything more than the public schedule. This is a simplified example of the technique used by Askarov et al. [AZM10] to prevent timing channels.

One further twist to consider is that in our model, proof terms can be persisted on stores. Therefore we must have some mechanism to ensure that when a proof term is dereferenced that the fact that it proves is still valid. One way to ensure this to simply check whether the facts within the proof are still valid, but this means that proof terms are not very durable. An alternative approach is to have stores that hold proof terms "subscribe" to the facts referenced by those proof terms, and rebuild the proofs whenever the epochs for those references ends.

We think that this scheme—epoch based concurrency control with subscriptions for proof terms—can be implemented soundly. However, as this section shows there are many subtleties involved, and significant future research must be done to formalize this approach and to validate its practical feasibility. Just as we analyzed acts-for tests by encoding them in a more primitive language, encoding transaction

management in a simple language containing basic message passing primitives is a potential technique for analyzing these concurrency control schemes.

4.7 Related Work

The problem of confidentiality leaks through policy changes has been widely studied. Becker [Bec12] examines information flow in declarative policy languages. In the context of language-based information flow analysis, Rx [SHTZ06] uses *metapolicies* that bound the information that can be learned by querying label relationships. Zheng et al. [ZM07] allow labels to be represented dynamically by objects that have their own information flow labels. Kozyri et al. [KSB+19] present a general framework for reasoning about chains of metapolicies, each describing the next.

The concern addressed in this chapter is lower-level. Instead of bounding the information learned by observing the policy, we are focused on the covert channels caused by observing requests about the policy. These information flows are examples of read channels. Our use of access labels to control read channels is based on K. Vikram's approach [Vik15].

Automated trust negotiation (ATN) protocols [WSJ00] use access control policies to prevent information leakage through credential requests. In the ATN setting, parties iteratively exchange credentials in an attempt to demonstrate that they are authorized to view the credentials presented in the next round.

The flow-limited authorization model (FLAM) [ALM15] also considers information flow through dynamic authorization queries in the context of an authorization

logic. Judgments in FLAM are predicated on an information flow label bounding the principals holding the information used in the derivation. The authors present a proof search algorithm that is similar to our implementation of `actsfor`, but they do not study the information flow properties of their algorithm.

Our language can be thought of as a domain specific language for tracking information flow in a reference monitor. The flow-limited authorization calculus (FLAC) [AM16] takes a very similar approach. FLAC has a "delegation value" type that serves the same purpose as our proof objects. Like our calculus, FLAC uses the DCC monad to track information flow. FLAC does not track read channels.

Chapter 5
Conclusions

Our goal in this dissertation was to bridge the divide between the strong security assurance provided by decentralized information flow control and the software engineering features required by modern distributed applications. This agenda has raised interesting problems on both ends of the divide, and we have presented contributions to both areas.

5.1 Contributions to Distributed Software Platforms

On the software-engineering side, Fabric demonstrates that language-based information flow control is a compelling approach to building secure distributed applications that operate in a federated environment. Fabric succeeds in offering both a simple, general abstraction for building secure systems and an implementation that can be used to build real applications with stronger security assurance than any previous platform for distributed computing.

The Fabric language supports the construction of secure distributed applications by integrating support for information flow control, object shipping, function shipping, mobile code, and transaction-based concurrency control. Fabric provides these features in the context of a fully featured object-oriented language, allowing developers to build software using familiar design patterns and idioms.

Implementing the Fabric language and system securely and efficiently required several technical innovations. The Fabric system includes novel static and dynamic

type-checking constraints to support data shipping, function shipping, and mobile code while respecting confidentiality and integrity requirements. Novel implementation techniques, including writer maps, distributed transaction logging, and our hierarchical two-phase commit protocol enable a strong transactional consistency model in the presence of mutual distrust.

The example applications we have implemented show that Fabric is an expressive language for building complex distributed applications, and that the Fabric system provides good performance. The FriendMap, multiuser calendar, and bidding agent examples model complex interactions between mutually distrusting entities, while the OO7 and CMS examples show that our implementation provides acceptable performance.

5.2 Contributions to Information Flow Analysis

Reasoning about Fabric's information flow properties has raised several interesting challenges for the formal analysis of decentralized information flow control systems, and we have presented important contributions in this area as well.

We have articulated a decentralized security principle that is suitable for evaluating a federated system like Fabric: a principal should only be harmed by someone that they trust. The definitions of flow and influenced flow given in Chapter 3 provide a useful language for rigorously defining the DSP.

Our definitions make minimal assumptions about the label model and the system model, and are therefore broadly applicable. They generalize existing semantic

security conditions, while accounting for the potential untrustworthiness of parts of the computational infrastructure.

Using these definitions, we were able to formally state and prove two forms of the decentralized security principle: a strong form that describes systems that aim to enforce noninterference, and a weak form that describes systems with downgrading.

Trust, authority, and integrity are closely linked, and the label model abstraction presented in Chapter 3 goes beyond previous frameworks by making these relationships explicit. The formal statement and proof of the DSP require an "is-trusted-to-enforce" relation and an "authority" function describing the intended semantics of labels. The extended decentralized label model instantiates the label model axioms and demonstrates the utility of these relations.

These contributions are a solid foundation for the mathematical modeling of distributed information flow control systems.

5.3 Contributions to Dynamic Distributed Authorization

Federated systems require a decentralized mechanism for authorization; systems that provide strong security guarantees must implement those authorization mechanisms without violating information flow restrictions. Since trust statements in federated systems are distributed, determining whether a computation is safe may require a separate distributed computation, which may in turn introduce potentially unsafe information flows.

We addressed this problem in Chapter 4 by constructing a calculus for distributed authorization queries that tracks information flow. We showed that programs in our language do not leak information, and that the language is sufficiently expressive to model nontrivial authorization queries.

Types in our calculus include information flow labels, and therefore type checking requires resolving authorization queries. Authorization queries are implemented as programs in the calculus, which must themselves be well-typed. Our authorization calculus resolves this apparent circularity using a novel dependent type system based on first-class proof objects: in order to evaluate authorization queries, programs must first construct proofs that the evaluation will not inappropriately leak information.

5.4 Future work

Fabric provides fertile ground for research into the interactions between information flow control and distributed systems, and there are several natural directions to extend the work presented in this dissertation.

One avenue for further exploration would be to integrate the theoretical frameworks presented in Chapters 3 and 4 into the Fabric implementation. The label model requirements given in Chapter 3 can be naturally encoded as a collection of Fabric interfaces; particular label models such as the DLM and the EDLM would be implemented by Fabric code that instantiates those interfaces.

This transformation would have several benefits. Implementing the principal and label infrastructure in Fabric provides assurance that those implementations do not

harm the confidentiality or integrity of the system. In particular, the authorization leaks discussed in Chapter 4 would be ruled out by Fabric's access label checking. This assurance would come at the cost of a more sophisticated type system: dependent types similar to those used in Chapter 4 would likely be necessary to typecheck label model implementations.

Another advantage of abstracting away the label model would be the potential to integrate Fabric applications with applications written for other DIFC platforms that use different label models (such as DStar or Aeolus). The ability to integrate software built using different technologies is another important feature of the modern software ecosystem that we have not addressed; a label model abstraction would be an important component in a principled analysis of integrated DIFC systems.

A related avenue for further research would be to investigate the theoretical relationships between different label models using the abstract label model framework given in Chapter 3. Montagu et al. [MPP13] have defined morphisms between label algebras, allowing them to formally characterize the fundamental differences between different label models. Their definition of a label model does not consider the is-trusted-to-enforce relation or the relationship between integrity and authority; extending their definitions in light of the DSP would shed further light on the relationships between different label models.

The authority function introduced in Chapter 3 provides an interesting parameter for exploring security conditions that are more restrictive than nonmalleability but less restrictive than noninterference. An authority function requiring high integrity for downgrading would allow a principal to delegate the ability to handle data without

delegating the ability to make decisions about the policy on the data. A more public authority function would encode a requirement that downgrading decisions must be observable, a kind of auditability requirement.

The calculus presented in Chapter 4 contains the low-level features necessary to implement the high-level authorization primitives that are implemented as trusted code in Fabric. A similar approach could be used to analyze the information-flow properties of other components of the Fabric implementation.

For example, although we have taken care to implement the distributed transaction management subsystem in a way that respects the information flow policies on data, the implementation is complex. An implementation in a calculus with information flow types and low-level communication primitives would increase the level of assurance that the Fabric system is secure. The formal properties of such a calculus may also shed further light on the interactions between transactions and revocation, as discussed in Section 4.6.

In the limit, a complete implementation of the Fabric system in either the Fabric language or a suitable lower-level language would be an interesting project. Such an implementation would allow the formal properties of the implementation language to be used to reason about the security and correctness of the system implementation. This would reduce the trusted computing base of Fabric programs and increase assurance that Fabric enforces applications' confidentiality and integrity requirements.

Chapter 4 analyzes the information flows in a fairly simple authorization scheme. Expanding the calculus to a more fully-featured language would enable implementation of more complex automated trust negotiation schemes. Using a domain-specific

language for implementing distributed reference monitors with information flow control would increase assurance in the trustworthy implementation of these security-critical components.

Experience writing complex applications in Fabric has inspired the design of new programming language mechanisms. For example, the design of first-order proof objects sketched in Section 4.5 was driven by a desire to encapsulate the security requirements of the FriendMap application and thereby reduce the annotation burden. Implementing first-order proof objects and investigating their formal properties is a promising avenue for future work. Moreover, this experience suggests that implementing more applications in Fabric could inspire other useful programming language features for secure distributed computation.

5.5 Summary

Information security is a critical challenge for the modern applications. We have shown that language-based information flow control can be extended to address the complex requirements of today's software ecosystem. My hope is that these contributions help move us towards a world where users can expect their data to be used appropriately, and where software providers have the tools to meet those expectations.

BIBLIOGRAPHY

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 147–160, January 1999.

[AFM05] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Publishing Company, 2005.

[ALM15] Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF)*, pages 569–583, July 2015.

[AM10] Aslan Askarov and Andrew C. Myers. A semantic framework for declassification and endorsement. In *Proceedings of the 19th European Symposium on Programming (ESOP)*, March 2010.

[AM11] Aslan Askarov and Andrew C. Myers. Attacker control and impact for confidentiality and integrity. *Logical Methods in Computer Science*, 7(3), September 2011.

[AM16] Owen Arden and Andrew C. Myers. A calculus for flow-limited authorization. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF)*, pages 135–147, June 2016.

[AMS+07] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st ACM Symposium on Operating System Principles (SOSP)*, pages 159–174, October 2007.

[AZM10] Aslan Askarov, Danfeng Zhang, and Andrew C Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th*

ACM conference on Computer and communications security, pages 297–307, 2010.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison Wesley Publishing Company, April 2003.

[Bec12] Moritz Y Becker. Information flow in trust management systems. *Journal of Computer Security*, 20(6):677–708, 2012.

[BFI14] Pierre Bourque, Richard E. Fairley, and IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK®): Version 3.0.* IEEE Computer Society Press, 3rd edition, 2014.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of the 1st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 78–86, November 1986.

[BMH06] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0.* O'Reilly Media, Inc., 5 edition, 2006.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[BS04] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW)*, pages 139–154, 2004.

[BS06] Niklas Broberg and David Sands. Flow locks – towards a core calculus for dynamic flow policies. In *Proceedings of the 15th European Symposium on Programming (ESOP)*, pages 180–196, 2006.

[BS11] A. Birgisson and A. Sabelfeld. Capabilities for information flow. In *Proceedings of the 6th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2011.

[BvDS13] Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 217–232, 2013.

[CAL97] M. Castro, A. Adya, and B. Liskov. Lazy reference counting for transactional storage systems. Technical Memo MIT/LCS/TM=567, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1997.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. IACR Cryptotololgy ePrint ArchiveReport 2016/086, 2016.

[CDE+07] Philip Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform clustered computing. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, May 1993.

[CDP+00] DeQing Chen, Sandhya Dwarkadas, Srinivasan Parthasarathy, Eduardo Pinheiro, and Michael L. Scott. InterWeave: A middleware system for distributed shared state. In *Proceedings of the 5th Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, May 2000.

[Chl10] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[CJD+18] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We still don't have secure cross-domain re-

quests: an empirical study of CORS. In *Proceedings of the 27th USENIX Security Symposium*, pages 1079–1093, 2018.

[CLM+07] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating System Principles (SOSP)*, pages 31–44, October 2007.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the 34th ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 494–503, 2002.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 198–209, October 2004.

[CM06] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 242–253, July 2006.

[CMA17] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable information flow control. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 1875–1891, October 2017.

[CMJL09] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2009.

[CPS+12] 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 *Proceedings of the 2012 USENIX Annual Technical Conference*, June 2012.

[CS08] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, pages 51–65, June 2008.

[CVM07] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium*, August 2007.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 202–215, 2001.

[EGC+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 393–407, 2010.

[EJM+14] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 1092–1104, November 2014.

[EKV+05] 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 *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP)*, October 2005.

[FF98] Matthew Flatt and Matthias Felleisen. Units: cool modules for HOT languages. In *Proceedings of the '98ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1998.

[For07] Bryan Ford. Structured streams: a new transport abstraction. In *Proceedings of the ACM SIGCOMM2007Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 361–372, August 2007.

[Fou] Free Software Foundation. GNU Classpath.

[GB01] Robert Grimm and Brian N. Bershad. Separating access control policy, enforcement, and functionality in extensible systems. *ACM Transactions on Computer Systems*, 19(1):36–70, February 2001.

[GLS+12] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60, 2012.

[HBBS14] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 1663–1671, New York, NY, USA, 2014. Association for Computing Machinery.

[HL11] E. Hammer-Lahav. The OAuth 2.0 authorization protocol. Network Working Group Internet-Draft, September 2011.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 48–60, 2005.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15:287–317, 1983.

[HTHZ05] Michael Hicks, Stephen Tse, Boniface Hicks, and Steve Zdancewic. Dynamic updating of information-flow policies. In *Proceedings of the 2005 Foundations of Computer Security (FCSW)*, 2005.

[HW87] M. Herlihy and J. Wing. Avalon: Language support for reliable distributed systems. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 90–94, July 1987.

[JL78] Anita K. Jones and Barbara Liskov. A language extension for expressing constraints on data access. *Communications of the ACM*, 21(5):358–367, May 1978.

[JVM+08] 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 *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2008.

[KAMS19] Elisavet Kozyri, Owen Arden, Andrew C. Myers, and Fred B. Schneider. Jrif: Reactive information flow control for java. In Joshua D. Guttman, Carl E. Landwehr, José Meseguer, and Dusko Pavlovic, editors, *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows*, pages 70–88. Springer International Publishing, Cham, 2019.

[KS20] Elisavet Kozyri and Fred B. Schneider. RIF: Reactive information flow labels. *J. Comput. Secur.*, 28:191–228, 2020.

[KSB+19] E. Kozyri, F. B. Schneider, A. Bedford, J. Desharnais, and N. Tawbi. Beyond labels: Permissiveness for dynamic information flow enforcement. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 351–35115, 2019.

[KYB+07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer,

M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating System Principles (SOSP)*, 2007.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 318–329, June 1996.

[Lis85] B. Liskov. The Argus language and system. In Goos and Hartmanis, editors, *Distributed Systems: Methods and Tools for Specification; An Advanced Course*, pages 343–430. Springer-Verlag, 1985.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[LMA+14] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 513–517, April 2014.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison Wesley Publishing Company, 1999.

[LZ05] Peng Li and Steve Zdancewic. Downgrading Policies and Relaxed Noninterference. In *Symposium on Principles of Programming Languages (POPL)*, 2005.

[MCCL07] Daniel Myers, Jennifer Carlisle, James Cowling, and Barbara Liskov. MapJAX: Data structure abstractions for asynchronous web applications. In *Proceedings of the 2007USENIX Annual Technical Conference*, June 2007.

[MH03a] Lynette Millett and Stephen Holden. Authentication and its privacy effects. *IEEE Internet Computing*, 7:54–58, November 2003.

[MH03b] Lynette Millett and Stephen Holden. Authentication and its privacy effects. *IEEE Internet Computing*, 7:54–58, 11 2003.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[ML85] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. *ACM SIGOPS Operating Systems Review*, 19(2):40–52, April 1985.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 129–142, 1997.

[ML00] 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.

[ML10] Leo A. Meyerovich and Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (Oakland)*, May 2010.

[MMN+04] John MacCormick, Nick Murph, Marc Najor, Chandramohan A. Thekkat, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[Mos90] J. E. B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, March 1990.

[MPP13] Benoˆ Montagu, Benjamin C. Pierce, and Randy Pollack. A theory of information-flow labels. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF)*, pages 3–17, June 2013.

[MSL+08] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript, 2008.

[MSZ06] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security (JCS)*, 14(2):157–196, 2006.

[MWC10] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A security-oriented subset of Java. In *Proceedings of the 2010 Network and Distributed System Security Symposium (NDSS)*, 2010.

[Mye99a] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[Mye99b] Andrew C. Myers. Mostly-static decentralized information flow control. Ph.D. thesis Report MIT/LCS/TR-783, Massachusetts Institute of Technology, Cambridge, MA, January 1999.

[MZZ+06] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, , July 2006.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction (ICCC)*, pages 138–152, April 2003.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 106–119, January 1997.

[NLV11] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Cloud Computing Security Workshop (CCSW)*, pages 113–124, 2011.

[NMB+16] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. TrustZone explained: Architectural features and use cases. In *Proceedings of the 2nd IEEE International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, 2016.

[OBDA08] Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn. Programming with live distributed objects. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP)*, volume 5142, pages 463–489, July 2008.

[OBS99] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the 1999 USENIX Annual Technical Conference*, 1999.

[Ora99] Oracle Corp. JAR file specification, 1999. http://download.oracle.com/javase/1.4.2/docs/guide/jar/jar.html.

[PC00] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 46–57, New York, NY, USA, 2000. Association for Computing Machinery.

[PRZB11] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP)*, pages 85–100, 2011.

[RD01a] A. Rowstron and P. Druschel. Storage management and caching in PAST a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, October 2001.

[RD01b] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[REG+03] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben

Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the 2nd IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust (FAST)*, pages 1–14, 2003.

[RFS+00] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 347–360, October 2000.

[RG05] Michael F. Ringenburg and Dan Grossman. AtomCaml: First-class atomicity via rollback. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, page 92–104, 2005.

[RPB+09] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[SHTZ06] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 202–216, July 2006.

[Sim03] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[SL13] David A. Schultz and Barbara Liskov. IFDB: Decentralized information flow control for databases. In *Proceedings of the 8th ACM European Conference on Computer Systems (EUROSYS)*, 2013.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[SRB+12] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 201–214, 2012.

[SRMM11] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell (HASKELL)*, September 2011.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 255–269, June 2005.

[STT08] Liuba Shrira, Hong Tian, and Doug Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/USENIX/IFIP International Conference on Distributed Systems Platforms (Middleware)*, December 2008.

[TS09] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Transactions on Software Engineering and Methodology*, 19(1):1:1–1:40, August 2009.

[TZ07] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. *ACM Transactions on Programming Languages and Systems*, 30(1), 2007.

[Vik15] K. Vikram. *Building distributed systems with information flow control.* PhD thesis, Cornell University Department of Computer Science, August 2015.

[Wil91] Paul R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *ACM SIGARCH Computer Architecture News*, 19(4):6–13, July 1991.

[WSJ00] William H Winsborough, Kent E Seamons, and Vicki E Jones. Automated trust negotiation. In *DARPA Information Survivability Confer-*

ence and Exposition, 2000. DISCEX'00. Proceedings, volume 1, pages 88–102, January 2000.

[WZB05] Marianne Winslett, Charles C Zhang, and Piero A Bonatti. PeerAccess: A logic for distributed authorization. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 168–179, 2005.

[YSRG06] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. Hilda: A high-level language for data-driven web applications. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 32–43, April 2006.

[YYSL12] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 85–96, 2012.

[Zal09] Michal Zalewski. Browser security handbook, part 2, 2009.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[ZBWM08] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–308, 2008.

[ZCMZ03] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (Oakland)*, pages 236–250, May 2003.

[ZM01] Steve Zdancewic and Andrew C. Myers. Robust declassification. In

Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW), pages 15–23, June 2001.

[ZM03] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 29–43, June 2003.

[ZM07] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 2007.

[ZSM19] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. Using information flow to design an ISA that controls timing channels. In *Proceedings of the 32nd IEEE Computer Security Foundations Symposium (CSF)*, June 2019.

[ZZNM02] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.