PRINCIPLED AND PRACTICAL WEB APPLICATION SECURITY

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Deian Stefan
December 2015

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(David Mazières) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(John C. Mitchell)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alejandro Russo)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Dan Boneh)

Approved for the Stanford University Committee on Graduate Studies

Abstract

Large-scale private user data theft has become a common occurrence on the web. A huge factor in these privacy breaches is that developers specify and enforce data security policies by strewing checks throughout their application code. Overlooking even a single check can lead to vulnerabilities. Unfortunately, even if developers manage to get all the checks right, most web applications rely on third-party code; a vulnerable or malicious third-party library, yet again, puts the user's data at risk.

This dissertation presents a novel approach to protecting sensitive data even when application code is buggy or malicious. The key ideas of this work are to separate the security and privacy concerns of an application from its functionality, and to use language-level information flow control (IFC) to enforce policies throughout the application codebase. The main challenge of this approach is at once to design practical systems that can be easily adopted by average developers, and simultaneously to leverage formal semantics that rule out large classes of design error. To address this challenge, this dissertation presents two systems—Hails and COWL—which respectively address the security issues web applications face on the server and in the browser.

Hails is a server-side web framework that separates the security and privacy concerns of an application from it functionality by following a new paradigm called model–policy–view–controller (MPVC). In the MPVC model, developers specify security policies in a single place, using a declarative policy specification language. Hails then enforces these policies across all application components using language-level IFC. This alleviates the need for application logic code to be intertwined with security checks and ensures that policies are enforced in a mandatory fashion, even across third-party code. Hails has been used by developers with a wide-range of expertise, from a novice high school student to expert web developers to build secure web sites with very small trusted computing bases. Some of these web applications were deployed production.

While Hails ensures that server-side code cannot leak or corrupt sensitive user data, COWL extends this security guarantee to the browser, where JavaScript, typically provided by multiple

disparate parties, computes on the user's sensitive data. COWL is a JavaScript confinement system that extends the browser security model with IFC, while retaining backwards compatibility with the existing Web. Much like Hails, COWL allows developers to associate policy with sensitive data, such as passwords. Within the confines of the browser, COWL then enforces these policies with IFC, prohibiting code from arbitrarily leaking data. This system has been implemented in both Firefox and Chromium, and is currently being standardized at the W3C as a new web specification.

Building practical systems, such as Hails and COWL, using information flow control required new developments in language-level security foundations. This dissertation describes some of the main results which were key to Hails and COWL, including: DC Labels, a simple yet expressive label model based on propositional logic; LIO, a dynamic, language-level IFC system implemented in Haskell; and, IFC-Inside, a generalization of LIO system to arbitrary languages. These foundations explore a new design point in language-level IFC, which addresses many of the shortcomings of previews results, while providing strong security guarantees; this was previously thought to be impractical for purely dynamic IFC enforcement.

Taken together, this dissertation presents practical systems that build on newly developed foundations in language-based security to provide end-to-end security to web applications. In addition to providing a solution to securing today's web applications, however, the strong security provided by these systems also opens up the possibility of deploying applications that, because of security concerns, were not previously practical.

Acknowledgments

I would first like to thank my advisors David Mazières, John C. Mitchell, and Alejandro Russo. They provided me with a model environment for conducting research and always encouraged me to work on things I enjoyed most. David's close mentorship, deep insights, and intuition have greatly shaped my views on research and systems building. John and Alejandro's guidance, encouragement, and perspectives have also highly influenced my research approach, especially in leveraging semantics and programming languages techniques. In addition to my advisors, I thank Brad Karp, Dan Boneh, and Arjen Lenstra for being my mentors and sounding boards at various important times of my career. I am truly thankful and honored to have worked with these great people.

Next, I would like to thank my close collaborators; this dissertation is based on research I performed with many great colleagues. Beyond those already mentioned, I would like to thank Edward Z. Yang, Amit Levy, Stefan Heule, Devon Rifkin, Petr Marchenko, Pablo Buiras, David Terei, Joe Zimmerman, Rahul Sharma, and Daniel B. Giffin. I would also like to thank my Stanford and Chalmers colleagues with whom I've had many great discussions—beyond those mentioned, Diego Ongaro, Adam Beley, Sergio Benitez, Arnar Birgisson, Andrea Bittau, Henry Corrigan-Gibbs, Ali Mashtizadeh, and Dante Zanarini.

I thank my committee—Dan Boneh, John Gill, David Mazières, John C. Mitchell, and Alejandro Russo—for their time, commitment, and feedback on this work.

During my Ph.D. I spent a summer at Mozilla Research working with Dave Herman and Bobby Holley. I thank them both for their help and insights in building adoptable systems for the web. I would also like to thank Brendan Eich for helping me bootstrap at Mozilla. Finally, I thank my Mozilla colleagues—Bebenita, Blake Kaplan, Christoph Kerschbaumer, Ian Melven, Garret Robinson, Brian Smith, Tanvi Vyas, and Boris Zbarsky—for the many fruitful discussions.

For roughly a year now I've been a member of the W3C Web Application Security group. Working on standards has been immensely rewarding, especially with such a supporting and encouraging group. Beyond their support and encouragement, I would like to thank Brad Hill, Wendy Seltzer,

and Daniel Veditz for making whole experience very welcoming.

I would like to thank my co-founders, colleagues, and advisors at GitStar—in particular, Christian Almenar, Devon Rifkin, David Mazières, and Dan Boneh. Working with them to build a company that is putting security research into practice has been immensely rewarding, educational, and full of joy.

Finally, I would like to thank my family and friends for being supportive, understanding, and loving. They've always stood by me and made my life easier. I would not be the same person without their support.

The research in this dissertation was funded by DARPA (CRASH, under contract #N66001-10-2-4088, and PROCEED, under contract #N00014-11-1-0276-P00002), by the NSF, by the AFOSR, by multiple gifts from Google, by a gift from Mozilla, and by the Swedish research agencies VR and STINT. I was supported by the DoD through the NDSEG Fellowship Program.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Web applications are routinely compromised. This is because building secure software applications is an error-prone task; existing programming models make it easy to write insecure code and notoriously difficult to produce secure code. Unfortunately, attempts to address this by creating a culture of good security practices and fixing vulnerabilities post-hoc are not working. Even high-profile web sites with dedicated security teams are vulnerable to application-level attacks—e.g., Github had a vulnerability that allowed a user to set the authentication keys for any project on the site [110], Snapchat was vulnerable to an attack that allowed any user to extract the username and phone number of any other user [89], while Facebook and United were vulnerable to attacks that respectively allowed one user to see and delete another user's private information [100, 88, 59].

Such vulnerabilities arise so frequently because web developers specify and enforce security policy by strewing checks throughout the application code. Today, overlooking even a single check can lead to vulnerabilities. Unfortunately, we cannot expect average developers to write bug-free code; even seasoned Linux kernel developers have committed changes—sometimes as small as a single line—that have led to vulnerabilities [207]. But, even if developers manage to place all the right checks in all the right places, today's applications are largely composed of third-party code which put the whole application at risk—a malicious or vulnerable library can easily leak and corrupt users' private data because, today, such code runs with the privileges of the application itself. How then can we expect average developers to build secure web applications?

This dissertation explores new system design points that change programmer behavior in favor of producing secure code, from the start. It is our thesis that the most effective way to change programmer behavior is to change the programming languages and APIs that developers use when building applications. From the release of Java, to Ruby on Rails, to iOS and Android, history has

shown that programming languages and APIs can have a profound impact on programmer behavior. Hence, by designing languages and APIs with security in mind, we can make it easier for developers to write secure code and restrict the damage that results from inevitable mistakes. Moreover, we can leverage recent advances in verification to apply formal reasoning to these languages and APIs so as to rule out large classes of design error.

One promising approach to designing secure languages and APIs is to base them on *information flow control* (IFC) [167]—in particular, decentralized information flow control [144]. Information flow control is a security mechanism that tracks and controls the flow of information through a system, according to a security policy. With IFC, a web developer can, for instance, specify that one user's private information (e.g., password, photo album, flight information) should not be read or modified by another user. In turn, by ensuring that policies follow data and enforcing them in a mandatory fashion, the IFC enforcement mechanism can guarantee that application components, even when buggy or malicious, cannot leak or corrupt the user's private data.

Information flow control has three general properties that make it especially appealing to the web application setting. First, IFC captures real-world security concerns in a direct, declarative way—policies are specified in terms of who can read and write various data—and enforces these policies throughout an application or system regardless of its structure or the sequence of operations performed. As this dissertation shows, capturing policy concerns in a high-level manner that is substantially divorced from the complex inner workings of an application makes it possible for developers to separate the security concerns of an application from the other code. This, in turn, leads to applications wherein most bugs do not have security implications, i.e., applications wherein most bugs are not vulnerabilities.

Second, IFC can be used to enforce security uniformly across different system layers (e.g., in hardware [230, 14], operating systems [63, 228, 104], programming languages[167, 144], distributed systems [229, 119], and browsers [221, 50, 217]). This is important for web applications, which face security issues on two fronts—server-side and browser-side—in particular, because the evolution of the Web has prioritized functionality over security and has led to the development of brittle, ad-hoc and completely different security solutions on both sides. This dissertation shows that by using a single mechanism—namely, IFC—we can address these limitations and provide end-to-end security against the privacy leaks that plague today's applications, without trading off functionality.

Third, IFC is an *objective* security mechanism, i.e., it provides concrete and, in the event of a design error, falsifiable guarantees. This is in contrast to the ad-hoc security mechanisms used by

today's developers, which often change as applications evolve with new features, and thus make it difficult to evaluate the security of an application in a non-subjective manner. IFC systems typically enforce a well-defined, formal policy called *noninterference* [73]. And, by designing languages with IFC in mind, we can even formally prove that application code cannot violate noninterference and thus leak or corrupt private user data.

Significant research, development, and experimental effort has been devoted to both, *static* and *dynamic*, language-level information flow control mechanisms. Static IFC mechanisms [145, 154, 204, 1, 81, 114, 52, 51], typically implemented as type systems or static analysis tools, prevent insecure programs from running; they have the usual benefits of low runtime overhead and few runtime failures due to security. Dynamic IFC mechanisms [32, 11, 12, 55, 78, 163, 221], on the other hand, typically monitor program executions and prohibit code from performing operations that would violate security policy, at runtime. Compared to static IFC analysis, most dynamic IFC mechanisms are more permissive and better suited for certain scenarios, such as web applications, where users join and leave the system arbitrarily, and where the security policy may depend on data provided by users, at runtime.

1.1 Contributions

This dissertation describes two systems based on dynamic, language-level IFC—Hails [72] and COWL [189]—and new foundations in language-level security that underly them [184, 186, 185, 183, 82]. These systems independently provide developers with mechanisms and APIs for building web applications that have strong security guarantees, server-side and client-side, respectively. More interestingly, however, when used together, Hails and COWL allow developers to build web applications that are secure, end-to-end.

Hails/MPVC Hails [72] is web framework designed to make it easier for average developers to write secure code. To this end, Hails introduces a new paradigm called model–policy–view–controller (MPVC), an extension to the popular model–view–controller (MVC) architecture. In MVC, models represent a program's persistent data structures, views provide a presentation layer for the user, and controllers decide how to handle and respond to particular user requests. Unfortunately, traditional MVC frameworks (e.g., Rails and Django) do not give security and privacy a first-class role. As a consequence, developers sprinkle guards throughout the codebase, around each

point of (data) access—a brittle, error-prone, and unscalable process that has led to many vulnerabilities [155, 100, 88, 59].

Yet, for important and sensitive data, developers typically have a more declarative, high-level security policy in mind—e.g., "a user's credit card number should not be sent to the network," or "a user's email address should only be seen by her friends." Hails makes such security and privacy concerns first-class by allowing developers to specify data access policies alongside data models, where developers already specify the format of data and how it should be stored. In contrast to MVC frameworks, our MPVC framework requires every data model to have an associated security policy that governs how the associated data may be used. Hails then enforces these policies system-wide, in a mandatory fashion, using language-level information flow control. This means that policies follow data (as it leaves the database) through all software components and, even when buggy or malicious, these components cannot leak data.

In Hails, application logic code does not need to be intertwined with security checks. This code solely needs to implement the application functionality; the framework enforces all the security policies. Indeed, the application logic code can even be written by untrusted third-party developers. In addition to separating concerns, this has the important consequence of minimizing the amount of code that is part of the *trusted computing base* (TCB). In Hails, only the code used to define data models and their associated security policy are part of the TCB, i.e., only bugs in the model and policy code amount to vulnerabilities.[1] This is particularly important because, as this dissertation shows, the model and policy code is only a tiny fraction of the application—typically no longer than a few hundred lines of code. This is in contrast to the hundreds of thousands of lines of code used to build application functionality.

Hails has been successfully used by developers with a wide-range of expertise, from a novice high school student to expert web developers to build secure web sites. Its performance is on-par with widely-deployed web frameworks (e.g., Apache/PHP)—Hails (and its underlying IFC mechanism) do not have the performance drawbacks common to dynamic language-level systems [220, 78, 219]. Moreover, the system was design with usability in mind. To our knowledge, Hails is the first IFC system to be used by average developers to build production applications. Some web sites built—notably, GitStar and LearnByHacking—were operated publicly by the core Hails team, but were used by people outside our small Haskell/Hails community. We recently started commercializing a framework highly influenced by the Hails design and our experience running Hails web

[1] Of course, this is in addition to the Hails enforcement mechanism and underlying language and OS runtimes.

sites.

Realizing our goal of designing and building a usable and practical web framework atop dynamic, language-level IFC required solving a number of technical challenges. First, a criticism of past IFC systems has been the perceived difficulty for application programmers to understand the security model. (Indeed, most of these systems have only been used by the security experts that developed them.) Hails addresses this by extending the popular MVC pattern with first-class data access policy; the MPVC paradigm provides developers with a guide for structuring applications which, in turn, makes it easy to understand the security models of their applications. This is in contrast to existing IFC system which provide no guide for structuring applications; from our experience and that of others [191, 219], the lack of structure makes it difficult to reason about the security model since it often leads to sprinkling policy code throughout the application code.

A second criticism of past IFC systems has been the difficulty of policy specification. In existing systems, developers specify policy by manually associating policies, in the form of *labels*, with objects (e.g., program variables, files, channels, database cells). Unfortunately, existing label formats make this challenging—they are either overly complex or not expressive enough for certain scenarios (e.g., web applications) [136]. Worse yet, figuring out precisely what the label of any particular object should be has traditionally been difficult. This dissertation addresses these challenges with DC Labels [184] and a declarative policy specification language for Hails.

DC Labels DC Labels is a new, simple label format designed to express policies that reflect the concerns of multiple stakeholders. A DC label is a pair of positive boolean formulas over *principals*: a *secrecy* formula specifying which principals can read or receive the data, and an *integrity* formula specifying which principals can modify it. For example, the label $\langle \texttt{https://maps.google.com} \vee \texttt{alice}, \texttt{alice} \rangle$ specifies that the data can be observed by Alice or Google Maps, but can only be modified by Alice; such a label may, for example, be used to protect Alice's location information in a web application. On the other hand, a page that integrates the user's bank statements from Chase and HSBC would be labeled $\langle (\texttt{alice} \vee \texttt{https://chase.com}) \wedge (\texttt{alice} \vee \texttt{https://hsbc.com}), \texttt{alice} \rangle$, reflecting the fact that the page contains information sensitive to both banks.

In contrast to most other real-world label models [163, 144, 228, 229, 63], the semantics of DC labels are simple. For example, the checks that an IFC system using DC Labels needs to perform when allowing information flows are just logical implications of the boolean formulas. Indeed, our "model seems to be the simplest model that is an extension to the set-based models [136]." In practice, simple and intuitive semantics are crucial. Indeed, this has allowed us and others to easily

use DC Labels in various systems [72, 189, 217, 218, 149, 36, 86]. Importantly, however, this simplicity does not come at the cost of formal considerations or expressiveness—DC Labels form a *label algebra* [184, 136] and their expressiveness is comparable to Myers and Liskov's DLM [144].

Policy specification/Automatic labeling Hails uses DC Labels to protect data stored in the database by data models. Developers, however, do not manually label this data. (Indeed, doing so would be a daunting task.) Instead, Hails provides a declarative, domain specific language (DSL) for specifying high-level policies alongside data models.

Specifically, when developers define a data model (e.g., for user profiles) they also define a *generic* policy specifying who is allowed to read and write any particular data model object (or field of an object). Each policy is a function from a data model object to a set of readers and writers—the principals allowed to read and write the data. Importantly, however, the policy function may use data contained in the object itself or other parts of the database in specifying the readers and writers. For example, when defining a policy for user profiles, our DSL can be used to express the high-level policies such as "only the user to whom the profile belongs is allowed to modify the profile" and "only the user's friends are allowed to see the email address." In both of these cases, the information necessary for the policy is contained in the data itself—e.g., the user named in the user profile object and the user's list of friends which is retrieved given the name, respectively.

In contrast to existing IFC systems, Hails is distinguished in allowing developers to specify such high-level policies, while still efficiently enforcing them in a mandatory fashion with IFC. This is possible because high-level, generic policies as specified using our DSL are instantiated as low-level DC labels when data is accessed. Indeed, one can understand our policies as way for developers to "automatically" label data. For example, when accessing Alice's email address from the user profile data model, the email field policy—which is a function from the object to a set of readers and writers, i.e., a DC label—is instantiated to produce the label on the email: $\langle \texttt{alice} \vee \texttt{bob} \vee \texttt{claire}, \texttt{alice} \rangle$. This label, as enforced by the language-level IFC mechanism, ensures that only Alice and her friends (Bob and Claire, in this case) can read the email address, and that only Alice can modify it.

LIO Hails relies on language-level information flow control to enforce application-specific policies at runtime. A significant challenge lies in designing an IFC system that is both principled and practical. Most existing language-level dynamic IFC systems are principled but lack features crucial to building real systems (e.g., exceptions, concurrency, policy inspection, and recovery from IFC-monitor failures). In fact, with the exception of the work presented in this dissertation and

the parallel work on Breeze [86], most dynamic language-level IFC systems have a *stop-the-world* semantics, i.e., the IFC mechanism halts programs that attempt to violate policy.

In practice, we cannot build a web application framework with such semantics. First, this would greatly impede usability: developers would have to write code that never attempted to violate policy since doing so would result in a fatal error.[2] And, second, *stop-the-world* semantics are inherently leaky: a program can leak a bit of information by performing an operation that stops the world (or not) based on the bit. Indeed, it was commonly believed that "fine-grained information flow control is practical only with some static analysis" precisely because "the difficulty with run-time checks is ... the fact of failure (or its absence) can serve as a covert channel. [144]" It is of no surprise that, until this work, practical static language-level IFC tools have far surpassed dynamic ones [145, 119].

This dissertation bridged this gap by borrowing ideas from IFC operating systems [63, 228, 104, 163]. In contrast to dynamic IFC languages, most IFC operating systems are practical. Unfortunately, they lack formal semantics and are not appropriate for web application frameworks—in particular, they cannot be easily adopted by web developers since they typically require installing a new OS and lack support for fine-grained policies common to web applications.

In an attempt to get the best from both worlds, we designed LIO [186, 185, 183], a dynamic language-level IFC system that shares many abstractions with OS-level IFC systems. Like IFC OSes, LIO enforces IFC at a coarse granularity. But, like IFC languages, LIO is more flexible and allows developers to associate labels with fine-grained objects within their program. By exploring a new design point in language-level IFC, LIO supports many modern language features to which developers have grown accustomed, including exceptions and concurrency. Moreover, LIO allows developers to inspect labels—this is important when building real systems, as this dissertation shows—and to recover from IFC violation attempts by encoding failures in terms of exceptions. Most dynamic IFC programming languages lack such features due to the covert channels that can arise, typically, as a result of complex program control flow—LIO eliminates these covert channels by construction [185, 183, 34, 82, 35].

We have formalized the sequential and concurrent implementations of LIO as extensions to the lambda calculus with small-step operational semantics and proved variants of noninterference. For the sequential LIO system, we proved *termination-insensitive noninterference*, i.e., any terminating

[2] Since most IFC systems consider a buggy but not malicious attacker model, i.e., they wish to prevent vulnerabilities due to developer error, the inability to fail non-fatally almost defeats the purpose of using dynamic IFC in the first place.

sequential LIO program cannot leak data by abusing language-level features; this proof was mechanically verified in Coq. While most dynamic language-level IFC systems prove such a theorem, a sequential LIO program can only leak information by diverging—LIO programs do not leak information by, for instance, accidentally trying to write secret data to a public channel. Like Breeze [86], this makes LIO permissive enough to be useful for preventing leaks due to bugs in application code. When considering malicious code, LIO, however, also improves on the state of the art by introducing *clearance* [228] to language-level IFC. Clearance is a form of discretionary access control that allows developers to limit access to data on a "need to know" basis and thus reduce opportunities for code to leak data through covert channels—after all, code that cannot access sensitive data cannot leak it.

For the concurrent LIO system, we proved a much stronger theorem: *termination-sensitive noninterference*. This theorem states that any program written in LIO cannot leak information by abusing language-level features, regardless of its timing (as internally observed within the application) or termination behavior. LIO is the first dynamic IFC language-level system to provide this result without hampering flexibility (e.g., by disallowing branching on secrets). Furthermore, to reduce the gap between language semantics and implementation, which can sometimes allow for real leaks that the semantic model does not capture, we extended LIO's semantics (and, in turn, implementation) to account for subtle implementation details, such as hardware caches [182, 34].

LIO has been implemented as a Haskell library atop which we built Hails. The library approach makes LIO (and, in turn, Hails) more adoptable than a new programming language—developers can keep using a familiar language and the compilers, tools, and libraries accompanying it. Equally important, the library approach has allowed us to evaluate different design points, as discussed in this dissertation, with rapid feedback. Indeed, LIO has been rewritten several times to address usability challenges and limitations that arose when building Hails. To accommodate writing web applications in different languages, we generalized the LIO design to other languages (e.g., JavaScript in Node.js)—we call this extension IFC-Inside [82]. In addition to Hails, however, LIO has been used to build several secure applications and systems. The system has been part of the curricula at Stanford (Functional Systems in Haskell) and UPenn (Advanced Programming), and has served as a research platform at Chalmers, Harvard, MIT Lincoln Labs, and University of Maryland.

COWL Within the confines of the server-side environment, Hails (with LIO) provides strong security guarantees—in particular, Hails ensures that server-side application code cannot leak or corrupt sensitive user data. However, web applications typically ship executable content to the browser,

in the form of JavaScript and HTML. Indeed, large parts of modern web sites are typically implemented in JavaScript that runs in the browser. (Some even completely shifted to the browser and only rely on servers to provide a simple storage layer.) Bugs in these browser-side *applications*, yet again, put the user's privacy at risk.

To make matters worse, these browser-side applications are largely composed of third third-party code in the form of libraries and frameworks. For example, jQuery and Modernizr are respectively used used by over 73% and 34% of the top 10,000 sites [156]. Yet, in the status-quo browser, all code runs with the privilege of the page and must be trusted to not leak the user's sensitive information. Unfortunately, even trustworthy libraries put the user's privacy at risk. For example, in September 2014, jQuery's web servers were compromised and could have been used to serve a malicious library [106]. More recently, Zhu [234] showed how to exploit a bug in UglifyJS, a popular JavaScript minifier, to inject backdoors into libraries that rely on it; amongst others, jQuery and CloudFlare use UglifyJS to minimize libraries before hosting them, in production. It is clear that we need mechanisms that provide security in the presence of untruted code.

Protecting sensitive information in the presence of untrusted code is a natural fit for information flow control. To this end, this dissertation presents COWL [189], a JavaScript confinement system that extends the browser security model with dynamic IFC. Like LIO, COWL allows developers to associate policies—in the form of labels—with sensitive data, such as passwords, within the browser. COWL, however, additionally allows server operators (e.g., a Hails application) to associate policies with data (and code), before shipping it to the browser. Within the confines of the browser, COWL then enforces these policies in a mandatory fashion and prohibits code, even a malicious jQuery, from arbitrarily leaking (or corrupting) data.

As with Hails and LIO, finding abstractions that web developers can use to build secure applications more easily was crucial. To this end, COWL adopted the Hails and LIO abstractions to the browser. Crucially, and unlike previous browser IFC systems [221, 78, 50], COWL did so by both retaining backwards compatibility with the existing model, and retrofitting existing browser concepts and constructs.

For example, in the existing model, developers already express policy in terms of *origins* (the address of a web server) and compartmentalize applications using browsing contexts (e.g., iframes). COWL leverages origins to provide a simple policy model, using DC Labels, that developers can use to protect sensitive data. It then enforces policies at context boundaries, for contexts that opted into using COWL—e.g., when an iframe communicates with the network or another page. As this dissertation describes, this has the added benefit of allowing one to implement IFC by repurposing

existing browser security mechanisms used for enforcing the same-origin policy [16] and content security policy [212].

COWL has been implemented in both Firefox and Chromium. In addition to the Hails applications which rely on COWL to enforce IFC in the browser, several secure browser-side applications have been written using COWL, including a password manager, an encrypted document editor, and a third-party personal finance mash-up (browser-side `mint.com`). Because of security, most of these applications were not previously possible (even with the application of other confinement techniques [221, 50, 78]). To our knowledge, COWL is the first dynamic browser IFC system that addresses practical challenges (e.g., backwards compatibility, usability, performance, multi-browser support, interoperability with existing security mechanisms) and has an underlying formal model [82]. COWL has also been adopted as a W3C web specification [181], and is undergoing standardization.

1.2 Organization

Part I presents our systems solutions that provide end-to-end security to web applications. Chapter 2 describes and evaluates Hails in detail. To demonstrate the flexibility and security of Hails, the chapter describes Hails in the context of *web platforms*, i.e., web sites that are extended by untrusted third-party code. Chapter 3 presents the COWL design and implementations, evaluates its performance, and describes its applicability to browser-side applications, not previously possible. In this part of the dissertation we use the terms "confinement" and "information flow control" interchangeably.

Part II presents new foundations in language-level security that we developed in the process of building Hails and COWL. Chapter 4 describes the DC Labels model, proof of soundness, and various extensions to the model. It also describes several design patterns and two implementations: the dynamic label format used in LIO, and a static implementation that is applicable to static IFC systems, such as SecIO [164] and HLIO [36]. In Chapter 5, we describe the sequential LIO IFC system. The chapter describes the sequential LIO design, formal semantics, and Haskell implementation. It also discusses the different design points considered in the process of building Hails, and presents mostly mechanized proofs of security. Chapter 6 presents the generalization of the LIO design, IFC-Inside, to different languages, using the multi-language approach of Matthews and Findler [125]. In addition, the chapter presents a generalization of LIO to the concurrent setting (originally described for Haskell in [183]), presents several non-interference proofs, and describes implementations for

Haskell, C, and JavaScript.

Finally, chapter 7 presents our conclusions.

Part I

End-to-End Web Application Security with Information Flow Control

Chapter 2

Hails: Protecting Data Privacy in Untrusted Web Applications[1]

Many modern web platforms are no longer written by a single entity, such as a company or individual, but consist of a trusted core that can be extended by untrusted third-party authors. Examples of this approach include Facebook, Yammer, and Salfesforce. Unfortunately, users running a third-party "app" have little control over what it does with their private data. Today's platforms offer only ad hoc constraints on app behavior, leaving users an unfortunate trade-off between convenience and privacy. A principled approach to code confinement could allow the integration of untrusted code while enforcing flexible, end-to-end policies on data access. This chapter presents a new framework, Hails, for building web platforms, that adds mandatory access control and a declarative policy language to the familiar MVC architecture. We demonstrate the flexibility of Hails by building several platforms, including GitStar, a code-hosting web site that enforces robust privacy policies on user data even while allowing untrusted apps to deliver extended features to users.

2.1 Introduction

Extensible web platforms represent an increasingly common way of developing and deploying software. Such platforms provide extensibility by allowing third-party *apps* to integrate, in a restricted manner, with the rest of the platform to offer additional features to users. Facebook popularized this

[1] This chapter is a copy of the extended version of the OSDI 2012 paper [72], which, at the time of this writing, is under submission at the Journal of Computer Security.

extension model for social networking. Others have emulated it: Yammer provides a similar social platform for for enterprises (running behind the firewall), while Dropbox and BitBucket provide a similar extension model for their project management platforms. The functionality users experience on these sites is no longer the product of a single entity. Instead, it is a combination of a core trusted platform, and apps written by less-trusted third-parties.

Many apps are only useful when they are able to manipulate sensitive user data—personal information such as financial or medical details, or non-public social relationships—but, unfortunately, once access to this data has been granted, there is no holistic mechanism to constrain what the app may do with it on today's platforms. This puts users' privacy at risk. For example, the Wall Street Journal reported that some of Facebook's most popular apps, including Zynga's FarmVille game, were transmitting users' account identifiers (sufficient for obtaining personal information) to dozens of advertisers and online tracking companies [180]. While Business Insider reported that nearly 7 million Dropbox credentials were leaked by third-party apps [102].

In a conventional platform model, a user sets a security policy on specific apps, or classes of apps, but these policies either grant or deny access to information, they do not constrain how it can be used. Apps are written to only function with all their access requests granted, giving them unfettered access to sensitive information. This forces users to choose between privacy or functionality. The platform cannot guarantee that the app will not mine private messages for credit card numbers and send this information to the app's developer. Furthermore, third-party apps run on servers outside of the control of the trusted platform, meaning all data the app accesses is exfiltrated. Unfortunately, even if users understand an app's behavior, they are poorly equipped to understand the consequences of exfiltration. In fact, a wide range of sophisticated third-party tracking mechanisms are available for collecting and correlating user information, many based only on scant user data [126].

In order to protect its users, the operator of a conventional web platform is burdened with implementing a complicated security system. These systems are usually ad hoc, relying on access control lists, human audits of app code, and optimistic trust in various software authors. Moreover, while some of these techniques are common, each platform ends up providing a different solution from others.

To address these problems, we have developed an alternate approach for building platforms that need to confine untrusted apps. We demonstrate the system by describing GitStar, a social code hosting web platform inspired by GitHub and BitBucket. GitStar takes a new approach to the app model: we host third-party apps in an environment designed to protect data, rather than allow developers to host them on a arbitrary server. In doing so, we can enforce security policies that

restrict information flow into and out of apps. This, in turn, enables a model wherein users can safely use apps without being asked whether to disclose data.

We built GitStar using a new web framework we developed called Hails. While other frameworks are geared towards monolithic web sites, Hails is explicitly designed for building web *platforms*, where it is expected that a site will comprise many mutually-distrustful components written by various entities. Of course, this also fits the degenerate case where a single entity is building the web site in full—in this case, Hails allows the site developers to protect user data from third-party libraries and bugs in their own application code.

Hails is distinguished by two design principles. First, security policies should be specified declaratively alongside data schemas, rather than spread throughout the codebase as guards around each point of access. Second, security policies should be mandatory even once code has obtained access to data.

The first principle leads to an architecture we call model–policy–view–controller (MPVC), an extension to the popular model–view–controller (MVC) pattern. In MVC, models represent a program's persistent data structures, views provide a presentation layer for the user, and controllers decide how to handle and respond to particular requests. The MVC paradigm does not give security and privacy a first-class role, making it easy for programmers to introduce vulnerabilities [155, 100, 88, 59]. By contrast, MPVC explicitly associates every model with a security policy governing how the associated data may be used.

The second principle, that data access policies should be mandatory, means that policies must follow data throughout the system and be enforced even once code has access to data. Hails uses a form of mandatory access control (MAC) to enforce end-to-end policies on data as it passes through software components with different privileges. While MAC has traditionally been used for high-security and military operating systems, it can be applied effectively to the app-platform model when combined with a notion of decentralized privileges such as that introduced by the decentralized label model [144].

Unlike the access control lists used by today's web platforms, the MAC regime allows a complex system to be implemented by a reconfigurable assemblage of software components that do not necessarily trust each other. For example, when a user browses a software repository on GitStar, a code-viewing component formats files of source code for convenient viewing. Even if this component is flawed or malicious, the access policy attached to the data and enforced by MAC will prevent it from displaying a file to users without permission to see it, or transmitting a private file to the component's author. Thus, the GitStar core component can make repository contents available

to any other component, and users can safely choose third-party viewers based solely on the features they deliver rather than on the trustworthiness of their authors.

A criticism of past MAC systems has been the perceived difficulty for application programmers to understand the security model. By extending the popular MVC pattern by binding security policy to the model, giving us MPVC, Hails offers a new design point that we believe addresses these concerns. To investigate this, we report on our experience and the experiences of other developers in using Hails to both build web platforms and third-party apps for GitStar. While our sample is yet small, our experience suggests MAC security does not impede application development within an MPVC framework.

The remainder of this chapter describes the design of Hails and several applications built on top of Hails, including the GitStar platform. We discuss design patterns used in building Hails web platforms, evaluate our system, provide a discussion, survey related work, and conclude.

2.2 Design

The Hails MPVC architecture differs from traditional MVC web frameworks such as Rails and Django by making security concerns explicit. An MVC framework has no inherent notion of security policy. The effective policy results from an ad hoc collection of checks strewn throughout the application. Unsurprisingly, these checks usually do not extend to third-party code, which constitute large parts of modern web apps. By contrast, MPVC gives security policies a first-class role. Developers specify policies in a domain-specific language (DSL) alongside the data model. The framework then enforces these policies system-wide, regardless of the correctness or intentions of untrusted code, primarily using language-level security.

MPVC applications are built from mutually distrustful components. These components fall into two categories: *MPs*, comprising model and policy logic, and *VCs*, comprising view and controller logic. An MP provides an API through which other components can access a particular database, subject to its associated policies. A VC, on the other hand, interacts with the user, invoking different MPs to fetch and store data. Our language-level confinement ensures that a data-model's policy is respected throughout the system, across the different components. For example, if an MP specifies that "only the user's friends may see his/her email address," then a VC (or another MP) reading the user's email address loses the ability to communicate over the network except to the user's friends (who are allowed to see that email address).

Figure 2.1 illustrates the interaction between different application components in the context of

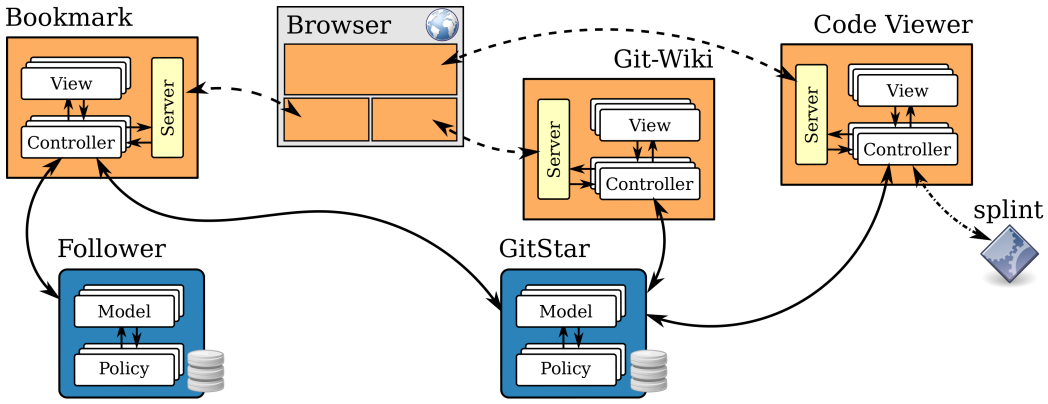

Figure 2.1: Hails platform with three VCs and two MPs. Dashed lines denote HTTP communication; solid lines denote local function calls; dotted lines denote message passing between browsing contexts; dashed-dotted lines denote communication with OS processes. MPs VCs are confined at the programming language level; browser execution contexts (e.g., iframes) are confined by COWL; OS processes are jailed and only communicate with invoking VCs.

GitStar. Two MPs are depicted: GitStar, which manages projects and `git` data; and Follower, which manages a directional relationship between users. Three VCs are shown invoking these modules: a source-code viewer, a `git`-based wiki, and a bookmarking tool. The code viewer presents syntax-highlighted source code and the results of static analysis tools such as splint [109]. Using the same MP, the wiki VC interprets text files using Markdown to transform articles into HTML. Finally, the bookmarking VC leverages both MPs to give users quick access to projects owned by other users whom they follow.

Because an application's components are mutually distrustful, the Hails design directly leads to greater extensibility. For example, anyone who doesn't like GitStar's syntax highlighting is free to run a different code viewer. Indeed, any of the VCs depicted in Figure 2.1 could be developed after the fact by someone other than the author of the MPs; because Hails's MAC security continues to restrict what code can do with data even after gaining access to it, no special privileges are required to access an MP's API.

2.2.1 Principals and privileges

Hails specifies policy in terms of *principals* who are allowed to read or write data. There are three types of principal. Users are principals, identified by usernames (e.g., `alice`). VCs and remote web sites that a component may communicate with are principals, identified by their origin URL [16] (e.g., `https://wiki.gitstar.org/` and `https://maps.google.com/`). MPs also have unique

principals which, by convention, start with the prefix "_" (e.g., the principal of the GitStar MP of Figure 2.1 is `_GitStar`).

An example policy an MP may want to enforce is "user `alice`'s mailing address can be read only by `alice` or by `https://maps.google.com/`." Such a policy would allow a VC to present `alice` her own address (when she views her profile) or to fetch a Google map of her address and present it to her, but not to disclose the address or map to anyone else. To be be flexible, Hails allows read and write permissions to each be expressed using arbitrary conjunctions and disjunctions of principals.

Enforcing such policies requires knowing what principals an app represents locally and what principals it is communicating with remotely. Remote principals are ascertained as one would expect. Hails uses standard authentication facilities (e.g., Mozilla's Persona [142], and OpenID [158]); a browser presenting a valid session token represents the logged-in user's principal. When code, server-side or in the browser, initiates outgoing HTTP requests to remote web sites, we consider the remote server to act on behalf of the principal of the web site, i.e., the origin URL.

Within the confines of Hails, code itself can act on behalf of principals. In particular, Hails provides unforgeable objects called *privileges* with which code can assert the authority of principals. The trusted Hails runtime passes appropriate privilege objects to MPs and VCs upon loading their code. For example, the GitStar MP is granted the `_GitStar` privilege. Thus, when a user wishes to use GitStar to manage her data, the policy on the data in question must specify `_GitStar` as a reader and writer so as to give GitStar permission to read the data and write it to its database should it chose to exercise its `_GitStar` privileges. In the browser, our trusted COWL runtime grants code running in a page the privilege of the page's origin; the VC serving the page can, however, specify a sub- or weakened-privilege.

2.2.2 Labels and MAC-based confinement

Hails associates a security policy with every piece of data in the system, specifying which principals can read and write the data. Such policies are known as *labels*. The particular labels used by Hails are called *DC labels*. We describe and formalize DC labels in Chapter 4, but, to be self-contend, we give a brief overview of their format and use in MAC.

A DC label is a pair of positive boolean formulas over principals: a *secrecy* formula, specifying who can read the data, and an *integrity* formula, specifying who can write it. For example, a file labeled $\langle \texttt{alice} \vee \texttt{bob}, \texttt{alice} \rangle$ specifies that `alice` or `bob` can read from the file and only `alice` can write to the file. Such a label is subjected on the code viewer VC of Figure 2.1, for example,

when fetching `alice`'s source code. The label allows the VC to present the source code to the project participants, `alice` and `bob`, but not disseminate it to others.

To ensure data protection, the Hails trusted runtime checks that remote principals satisfy any relevant labels before permitting communication. For instance, data labeled $\langle \texttt{alice} \vee \texttt{bob}, \texttt{alice} \rangle$ cannot be sent to a browser whose only principal is `charlie`. The actual checks performed involve verifying logical implications. Data labeled $\langle S, I \rangle$ can be sent to a principal (or combination of principals) p only when $p \Longrightarrow S$. Conversely, remote principal p can write data labeled $\langle S, I \rangle$ only when $p \Longrightarrow I$. Given these checks, $\langle \text{TRUE}, \text{TRUE} \rangle$ labels data readable and writable by any remote principal, i.e., the data is public, while $p = \text{TRUE}$ means a remote party is acting on behalf of no principals (e.g., when a user session is not authenticated).

The same checks would be required for local data access if code had unrestricted network access. Hails could only allow code to access data it had explicit privileges to read. For example, code without the `alice` privilege should not be able to read data labeled $\langle \texttt{alice}, \text{TRUE} \rangle$ if it could subsequently send the data anywhere over the network. However, Hails offers a different possibility: code without privileges can read data labeled $\langle \texttt{alice}, \text{TRUE} \rangle$ so long as it first gives up the ability to communicate with remote principals other than `alice`. Such communication restrictions are the essence of MAC.

To keep track of communication restrictions, the runtime associates a *current label* with each thread. The utility of the current label stems from the transitivity of a partial order called "*can flow to*." We say a label $L_1 = \langle S_1, I_1 \rangle$ *can flow to* another label $L_2 = \langle S_2, I_2 \rangle$ when $S_2 \Longrightarrow S_1$ and $I_1 \Longrightarrow I_2$—in other words, any principals p allowed to read data labeled L_2 can also read data labeled L_1 (because $p \Longrightarrow S_2 \Longrightarrow S_1$) and any principals allowed to write data labeled L_1 can also write data labeled L_2 (because $p \Longrightarrow I_1 \Longrightarrow I_2$).

A thread can read a local data object only if the object's label can flow to the current label; it can write an object only when the current label can flow to the object's label. Data sent over the network is always protected by the current label. (Data may originate in a labeled file or database record but always enters the network via a thread with a current label.) The transitivity of the *can flow to* relation ensures no amount of shuffling data through objects or components can result in sending the data to unauthorized principals.

A thread may adjust the current label to read otherwise prohibited data, only if the old value can flow to the new value. We refer to this as *raising* the current label. Allowing the current label to change without affecting security requires very carefully designed interfaces. Otherwise, labels themselves could leak information. In addition, threads could potentially leak information by not

terminating (so called "termination channels") or by changing the order of observable events (so called "internal timing channels"). As described in Chapters 5 and 6, the MAC enforcement system we developed for Hails, LIO, is the first production system to address these threats at the language level [183, 82].

Hails prevents the current label from accumulating restrictions that would ultimately prevent the VC from communicating back to the user's browser. In MAC parlance, a VC's *clearance* is set according to the user making the request, and serves as an upper bound on the current label. Thus, an attempt to read data that could never be sent back to the browser will fail, confining observation to a "need-to-know" pattern. This is an important design choice for both usability, since it ensures that a VC's current label is never raised to a point where it cannot reply to the end user, and security, since it ensures that a malicious VC can only "leak" data to the remote user (who may be both the VC author and attacker) via a covert channels (e.g., "external timing channels" which encode information by carefully delaying HTTP responses [30, 67]) insofar as it can read the data. Clearance ensures that a component cannot read data more more sensitive than data it could otherwise send to the user directly.

2.2.3 Model-Policy (MP)

Hails applications rely on MPs to define the application's data model and security policies. An MP is a library with access to a dedicated database. Though MPs may contain arbitrary code and can expose an arbitrary API, we encourage using the dedicated database. In doing so, MP code only needs to specify what sort of data may be stored in the database and what access-control policies should be applied to it. This also ensures that other components can access the MP via a common interface—the Hails database API. Better still, it allows MP developers to leverage our DSL, described in Section 2.2.3, for specifying data policies in a concise manner.

The Hails database system provides a common document-oriented interface, similar to MongoDB [43], atop different persistence layers (e.g., MongoDB, file system, or even a REST API). Logically, a Hails *database* consists of a set of *collections*, each storing a set of *documents*. Each document, in turn, contains a set of *fields*, or named values. Some fields are configured as *keys*, which are indexed and identify the document in its collection; all other fields are non-indexed *elements*.

An MP can restrict access to the different database layers using labels. A label is associated with every database, restricting who can access the collections in the database and, at a coarse level, who can read from and write to the database. Similarly, a label is associated with a collection, restricting

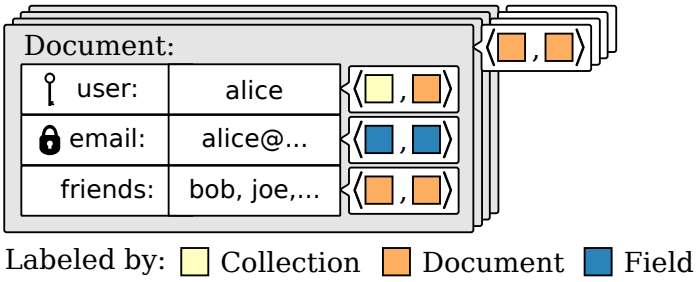

Figure 2.2: GitStar MP user documents and Follower MP friend relationship documents. Each user document is indexed by a key (`user`-name) and contains the user's `email` address and `fullName`. Documents and `email` fields are dynamically labeled using a data-dependent policy; the secrecy of the `user` key and is protected by the static collection label, the document label protects its integrity. The "unlabeled" `city` fields are protected by their corresponding document labels. Each follower relationship document contains two indexable keys—the `user` and `follows`, the name of the user they follow—and a field containing the `date` the user started following their friend.

who can read and write documents in the collection. The collection label additionally serves the role of protecting the keys that identify documents—a computation that can read from a collection can also read all the key values.

Automatic, fine-grained labeling

While static policies on databases and collections are often sufficient, in many web applications, dynamic fine-grained policies on documents and fields are desired. Consider the simplified models shown in Figure 2.2. Each user profile model contains fields corresponding to a user's username, email address, and full name, while each follower model is a mapping from one username to another—the username of the user they are following. In this scenario, the MP may configure usernames as keys in order to, for example, allow VCs to search for `alice`'s profile. Additionally, the MP may specify database and collection labels that restrict access to documents at a coarse grained level. However, these labels are not sufficient to enforce fine grained dynamic policies such as "only `alice` may modify her profile information" and "only her friends (`bob`, `joe`, etc.) may see her email address."

Hails introduces a novel approach to specifying document and field policies by assigning labels to documents and fields as a function of the document contents itself. This approach is based on the observation that, in many web applications, the authoritative source for who should access data is a function of the data itself. For example, in Figure 2.2, the username field values can be used

to specify the user profile document and field policies mentioned above: `alice`'s document is labeled $\langle \text{TRUE}, \texttt{alice} \vee \texttt{\_GitStar} \rangle$, while the `email` field value is labeled $\langle \texttt{alice} \vee \texttt{bob} \vee \texttt{joe} \vee \cdots \vee \texttt{\_GitStar}, \text{TRUE} \rangle$. The document label guarantees that only `alice` or the MP can modify any of the constituent fields. The label on the email-address field additionally guarantees that only `alice`, the MP, or her friends—information available in the followers model—can read her address.

```
database $ do
  -- Set database label:
  access $ do
    readers ==> anybody
    writers ==> anybody

-- Set policy for new "users" collection:
collection "users" $ do
  -- Set collection label:
  access $ do
    readers ==> anybody
    writers ==> anybody
  -- Declare user field as a public indexable key:
  field "user" $ publicIndex
  -- Set document label, given document doc:
  document $ λdoc -> do
    readers ==> anybody
    writers ==> ("user" `from` doc) \/ _GitStar
  -- Set email field label, given document doc:
  field "email" $ labeled $ λdoc -> do
    let user =  "user" `from` doc
    -- Fetch all the user's friends via the Follower MP:
    friends ← run $ withFollowerMP $
       map (get "follows") ∘ findAll $ select ["user" -: user ] "followers"
    readers ==> user \/ (fromList friends) \/ _GitStar
    writers ==> anybody
```

Figure 2.3: DSL-specification for the GitStar users policy. Here, `anybody` corresponds to the boolean formula TRUE; `fromList` converts a list of principals to a disjunction of principals; and, `"x" `from` doc` retrieves the value of field `x` from document `doc`. The `database` and `collection` labels are static. Field `user` is configured as a public indexable key. Finally, each document and `email` field is labeled according to a function from the document itself to a set of `readers` and `writers`. The latter invokes the Follower MP to fetch the user's list of friends (i.e., users they follow).

Hails's data-dependent "automatic labeling" simplifies reasoning about security policies and localizes label logic to a small amount of source code. Figure 2.3 shows the implementation of the GitStar users policy, as described above, using our DSL. Specifying labels on the database

and collections is simply done by setting the respective `readers` and `writers` in the `database` and `collection` sections. Similarly, setting a document or field label is done using a function from the document itself to a pair of `readers` and `writers`.

In contrast to the original DSL [72], which disallowed side effects when computing labels, our current DSL allows MPs to perform arbitrarily complex actions, such as database lookups, using the `run` keyword. Such actions can be used when labeling collections, documents, and fields. For example, in Figure 2.3, the GitStar MP performs a database lookup of the user's list of friends (via the Follower MP) when computing the label of their email address. Importantly, when an MP executes such actions, the Hails runtime ensures that they are confined according to MAC. We remark that, while this added flexibility may appear to make it harder to reason about policy, our experience suggests otherwise; in sections 2.6 and 2.7, we discuss this and the trade-off of this design choice.

Database access and policy application

MP policies are applied on every database operation. For example, when a thread attempts to insert a document into an MP collection, the Hails runtime first checks that that the thread can read from and write to the database and collection, by comparing the thread's current label with that of the database and collection. Subsequently, the field- and document-labeling policy functions are applied to the document and fields. If the policy application succeeds—it may fail if the thread cannot label data as requested—the Hails runtime removes all the labels on the document and performs the write. By removing labels and ensuring that every database operation is mediated according to the policy, we ensure that policy code is localized to MPs.

Hails also allows threads to insert already-labeled documents (e.g., documents retrieved from another MP or directly from the user). As before, when inserting a labeled document, the MP database and collection must be readable and writable at the current label. Different from above, the thread does not need to apply the policy functions; instead, the Hails runtime verifies that the labels on fields and the document agree with those specified by the MP . Finally, if the check succeeds, the hails runtime strips the labels and performs the write.

Application components can also fetch stored data. When performing a fetch, application components specify a query predicate on indexed keys (or TRUE). As with insert, when fetching data, the runtime first checks that that the thread can read from the database and collection. Next, the documents matching the predicate are retrieved from the database. Lastly, the field- and document-labeling policy functions are applied to each document and field; the resultant labeled documents

are returned to the invoking component.

We remark that while the Hails database system supports other standard operations, including (partial) update and delete, the restrictions on most of these operations are similar to those of insert and fetch. We thus elide their details and refer the interested reader to the Hails documentation for details [195].

2.2.4 View-Controller (VC)

VCs interact with users. Specifically, controllers handle user requests, and views present interfaces to the user. However, VCs do not define database-backed models. Instead, a controller invokes one or more MPs when it needs to store or retrieve user data. This data can also be passed on to views when rendering user interfaces.

Each VC is a standalone process, linked against the MP libraries it depends on to provide a data model. The VC author solely provides a definition for a *main* controller, which is a function from an HTTP request to an HTTP response. This function may perform side-effects: it may access a database-backed model by invoking an MP, read files from the labeled file system, etc. Hails relies on LIO's language-level confinement to prevent the VC and MPs it invokes from modifying or leaking data in violation of access permissions; we use OS-level resource management and isolation mechanisms to enforce platform-specific policies not otherwise enforced at the language level.

At the heart of every VC is the Hails HTTP server. The server, a privileged part of the trusted computing base (TCB), receives HTTP requests and invokes the main VC controller to handle them. When a request is from an authenticated user, the server sets the `X-Hails-User` header to the username and attests to the request's contents for the benefit of VCs and MPs that care about request provenance and integrity. The request is also labeled according to the `Sec-COWL` header supplied by the browser (see Section 2.2.5). In turn, the main controller processes the supplied request, by potentially calling into MPs to interact with persistent state, and finally returns an HTTP response. The server returns the provided response to the browser on the condition that it depend only on data the user is permitted to observe.

Enhancing VC functionality with MAC-aware libraries

To carry out their duties, components typically need access to various capabilities. While LIO provides many libraries (e.g., a file system and threads library) "out of the box," we extended LIO with several libraries. Below, we describe two of them: the Hails HTTP client and a library for safely

executing external programs.

Since many VCs (and MPs) rely on communication with external services, usually over HTTP, we implemented an HTTP client on top of LIO. Before establishing a connection, and on each read and write, the HTTP client checks that the current label of the invoking thread is compatible with the remote server principal. In practice, this means VCs can only communicate with external hosts when they have not read any sensitive data or they have only read data *explicitly* labeled for the external server.

However, when communicating with other Hails servers (via HTTPS), the HTTP client can be more flexible by not imposing any read restrictions. Instead, it can return a fine-grained labeled response, which the VC can then inspect at its own will (e.g., when ready to raise its label). The label of the response is supplied by the remote server via a `Sec-COWL` header. Following the COWL spec [181], the HTTP client only considers responses with valid labels, i.e., labels that the principal of the remote server can satisfy. A more flexible, but also more complex, approach to communicating between different MAC-confined domains is to use the DStar protocol [229]; we leave this to future work.

In addition to communicating with external services, real-world applications also rely on external programs to implement different functionality. For example, as highlighted in Figure 2.1, GitStar's code viewer relies on splint, a standalone C program, to flag possible coding errors. Addressing this need, Hails provides a mechanism for spawning confined Linux processes with no network access, no visibility of other processes, and no writable file system shared by other processes. Each such processes is governed by a fixed label, namely the VC's current label at the time the external program was spawned. In turn, labeled file handles can be used to communicate with the process, subject to the restrictions imposed by the current thread's label.

Safely executing code in the browser

When a user examines a private repository through an app such as the GitStar code viewer, Hails prevents the app VC from leaking private contents directly (e.g., using the HTTP client) and indirectly (e.g., via the database) within the confines of the server-side environment. However, VCs typically ship content to the browser where JavaScript or HTML may also attempt to leak data.

Hails prevents code from inappropriately leaking sensitive data on the client-side with COWL [189, 181]. COWL is a language-level confinement system that adopts the MAC-based mechanisms of Section 2.2.2 to the browser, as described in Chapter 3. Most notably, COWL extends the browser with labeled browsing contexts, i.e., labeled pages and iframes. These labeled browsing contexts

are analogous to server-side labeled threads: code running within a context is confined according to MAC, our trusted runtime restricts contexts from communicating with other contexts or web servers according labels. For instance, a GitStar user profile page labeled $\langle$`alice`$\vee$ `https://gitstar.org` $\vee$ `https://gravatar.com`, TRUE$\rangle$ is allowed to fetch `alice`'s image avatar from the Gravatar servers or use the `XMLHttpRequest` (XHR) constructor to fetch data from the main GitStar VC, but it cannot, for example, communicate with `evil.appspot.com`.

To ensure that code running in the browser is appropriately confined, the server-side Hails HTTP server supplies a `Sec-COWL` HTTP response header with every VC response. The response header value specifies the initial label (and privilege) of the browsing context, which is in turn set by the COWL runtime. By default, a page's label is set to the label of the VC thread that produced the response. But, code running in the browser can also raise the context label to subsequently read more sensitive information. For instance, in the example above, the initial profile page may have been public (e.g., because it did not yet fetch any sensitive data). But in fetching `alice`'s profile information via XHR, the page's label was raised to protect her sensitive data.

In addition to setting context labels, the `Sec-COWL` header is used to communicate labels on data between the server and client. For example, when responding to an XHR request for profile data, the GitStar VC may specify (explicitly or implicitly, via its current label) that the label of the response is $\langle$`alice`$\vee$ `https://gitstar.org` $\vee$ `https://gravatar.com`, TRUE$\rangle$; of course, in most cases such labels are specified by the MP using our DSL.

Within the confines of the browser, VCs can execute arbitrary code and use most of the Web platform APIs [189, 181], subject to confinement. COWL, however, provides additional APIs for labeling data client-side and thus imposing restrictions on how such data is used by other contexts (e.g., an iframe of a different VC, or a third-party service such as Google maps). This allows much of the VC implementation to be client-side and even consider scenarios not previously possible (see [189]).

2.2.5 Life-cycle of an application

In this section, we use GitStar's deployment model to illustrate the life-cycle of a Hails application from development, through deployment, to a user-request.

Application development and deployment

A third-party application developer may introduce a new data model to the GitStar platform by writing an MP. For example, the Follower MP shown earlier specifies a data-model for storing a relation between users, as well as a policy specifying who is able to read, create and modify those relationships. Once written, the developer uploads the library code to the GitStar servers where it is compiled and installed. The platform administrator generates a unique privilege for the new MP and associates it with a specific database in a globally-accessible configuration file. Subsequently, any Hails code may import the MP, which when invoked, will be loaded with its privilege and database handle.

The third-party developer may build a user interface to the newly-created model by writing a VC and registering a subdomain with the platform. As with MPs, developers upload their VC code to the GitStar servers where it is compiled and linked against any MPs it depends on. Thereafter, the platform administrator generates a privilege for the new VC, which corresponds to the hostname of the VC, and uses a program called `hails`, which contains the Hails runtime and HTTP server, to dynamically load the main VC controller and service user requests on the dedicated subdomain.

While in this example both the VC and MP were implemented by a single developer, third-party developers can implement applications consisting solely of a VC that interacts with MPs created by others. In fact, in GitStar, most applications are simply VCs that use the GitStar MP to manage projects and retrieve `git` objects. For example, the `git`-based wiki application, as shown in Figure 2.1, is simply a VC that displays formatted text from a particular branch of a `git` repository.

An example user request

When an end-user request is sent to the GitStar platform, an HTTP proxy routes the request to the appropriate VC HTTP server based on the hostname in the request.

The Hails server receiving the forwarded request invokes the main controller of the corresponding VC in a newly spawned thread. The controller is executed with the VC's privileges and sanitized request. The HTTP server sanitizes the incoming request by removing sensitive headers such as `Cookie`; it also sets the `X-Hails-User` header to the user-name, if the request is from an authenticated user. To ensure that sensitive data from the browser is not leaked server-side, the request is labeled according to the `Sec-COWL` HTTP request header, which encodes the label of the browsing context that performed the request.

The main controller may be a simple request handler that returns a basic HTML page without

accessing any sensitive data (e.g., an index or about page). A more interesting VC may access sensitive user data from an MP database before computing a response. In this case, the VC invokes the MP by performing a database operation such as insert or fetch. The invocation consists of several steps. First, the Hails runtime instantiates the MP with its privilege and establishes a connection to the associated database. Then, the MP executes the database operations supplied by the VC, and, in coordination with the Hails runtime, labels the data according to its policies. While some database operations are not sensitive (e.g., accessing a public `git` repository in GitStar), many involve private information. In such cases, the database operation will also raise the current label of the VC thread, and thereby affect all its future communication.

When a VC produces an HTTP response, the runtime checks that the current label, which reflects all data accesses or other sensitive operations, is still compatible with the end-user's browser. For example, if `alice` has sent a request to the code viewer VC asking for code from a private repository, the response produced by code viewer will only be forwarded by the Hails server if the final label of code viewer VC thread can flow to $\langle \texttt{alice}, \text{TRUE} \rangle$; otherwise, the Hails server responds with an error message. Recall that clearance ensures that the VC label ends up being compatible with the user's browser label in most cases.[2] Indeed, clearance ensures that such failures are not delayed and occur early into the VC computation (e.g., when the VC attempts to read overly sensitive data).

To prevent leaks client-side, the Hails HTTP server associates a `Sec-COWL` HTTP response header with every response. This header conveys the label of the response to browsers that support COWL [181]. When the response is data (e.g., JSON) the COWL-enabled browser ensures that the data cannot be leaked by code running in the browser. On the other hand, when the response is active content (e.g., an HTML page), the browser ensures that the content code is confined according to the label.

As detailed in Section 2.3.3, our client-side confinement system, COWL [189, 181], restricts all incoming responses and outgoing requests according to the response label. For example, if the code viewer returns a response labeled $\langle \texttt{alice} \vee \texttt{https://code.google.com}, \text{TRUE} \rangle$, the rendered page may retrieve scripts for prettifying code from `https://code.google.com`, but not retrieve images from `https://haskell.org`. On the other hand, a publicly labeled response imposes no restrictions on the requests triggered by the page.

[2] It is possible for an MP (or VC) to raise the clearance using its privilege as to allow the thread to read data more sensitive than what the end-user is allowed to see. However, this data should not be sent client-side and thus the reason for performing this check. MPs must explicitly declassify such data.

2.2.6 Trust assumptions

The Hails runtime, including the confinement mechanisms, HTTP server, and libraries are part of the TCB. Parts of the system, namely our labels and confinement mechanisms (LIO and COWL), have been formalized in [187, 184, 183, 136, 82]; Part II of this dissertation presents these formal details. We remark that different from other work, our server-side language-level concurrent confinement system is sound even in the presence of termination and internal timing covert channels [183, 82]. However, similar to other MAC systems (e.g., [119]), we assume that the remaining Hails components are correct and that the underlying OS, browser, and network are not under the control of an attacker.

By visiting a web page, the MPs invoked by the VC presenting the page are trusted by users to preserve the confidentiality and integrity of their data. This is a consequence of MPs being allowed to manage all aspects of their database. However, one MP cannot declassify data managed by another, and thus users can choose to use trustworthy MPs. To facilitate this choice, platforms should make MP policies and dependency relationships between VCs and MPs available for inspection.

Since a user can choose to invoke a VC according to the MPs it depends on, VCs are *mostly* untrusted. On the server-side, VCs cannot exfiltrate user data from the database without collusion from an MP the user has trusted. Nevertheless, VCs cannot be considered completely untrusted since they directly interact with users through their browser. Unfortunately, in today's browsers, even with COWL, a malicious VC can coerce a user to declassify sensitive data (e.g., by tricking users to execute code in their browser console).

2.3 Implementation

Hails employs a combination of application-level, OS-level, and browser-level confinement mechanisms spread across all layers of the application stack to achieve its security goals. Most notably, we use language-level information flow control (IFC) server-side and in the browser to enforce fine-grained policies. This section describes some of the implementation details of these language-level mechanisms and our OS sandbox.

2.3.1 Server-side language-level confinement

Hails applications are written in Haskell. Haskell is a statically- and strongly-typed, memory-safe language. Crucially, Haskell's type system distinguishes operations involving side-effects (such as

potentially data-leaking IO) from purely-functional computations. As a consequence, for example, compiling a VC's main controller with an appropriately specified type is sufficient to assert that the VC cannot perform arbitrary network communication.

Hails relies on the safety of the Haskell type system when incorporating untrusted code. However, like other languages, Haskell "suffers" from a set of features that allow programmers to perform unsafe, but useful, actions (e.g., type coercion). To address this, we extended the Glasgow Haskell Compiler (GHC) with `Safe Haskell` [197]. `Safe Haskell`, deployed with GHC as of version 7.2, guarantees type safety by removing the small set of language features that otherwise allow programs to violate the type system and break module boundaries.

With this change, Haskell permits the implementation of language-level dynamic IFC as a library. Accordingly, we implemented LIO [187, 183], which employs the label-tracking and confinement mechanisms of Section 2.2.2. Despite sharing many abstractions with OS-level IFC systems, such as HiStar [228] and Flume [104], LIO is more fine-grained (e.g., it allows labels to be associated with values, such as documents and email addresses) and thus better suited for web applications.

We believe the Hails architecture is equally realizable in other languages, though possibly with less backward compatibility. For example, JiF [145], Aeolus [41], and Breeze [86] provide similar confinement guarantees and are also good choices. However, to use existing libraries JiF and Aeolus typically require non-trivial modifications, while Breeze requires porting libraries to a new language. Conversely, about 12,500 modules in Hackage (34%), a popular Haskell source distribution site, are currently safe for Hails applications to import. Of course, the functions that perform arbitrary IO are not directly useful, and, like in JiF, must be modified to run in LIO. Nevertheless, many core libraries require no modifications.

2.3.2 OS-level confinement

Hails uses Linux isolation mechanisms to confine processes spawned by application components. These techniques are not novel, but it is important that they work properly. Using *clone* with the various `CLONE_NEW*` flags, we give each confined process its own mount table and process ID namespace, as well as a new network stack with a new loopback device and no external interfaces. Using a read-only bind-mount and the tmpfs file system, we create a system image in which the only writable directory is an empty `/tmp`. Using cgroups, we restrict the ability to create and use devices and consume resources. With `pivot_root` and `umount`, we hide file systems outside of the read-only system image. The previous actions all occur in a setuid root wrapper utility, which finally calls

`setuid` and drops capabilities before executing the confined process.

2.3.3 Browser-side language-level confinement

As mentioned in Section 2.2.4, Hails prevents code from inappropriately leaking sensitive data on the client-side with COWL [189, 181]. COWL is a language-level confinement system that adopts the confinement mechanisms of Section 2.2.2 to the browser, while retaining backward compatibility with the existing Web. We implemented COWL as modifications to the Chromimum and Firefox browsers, largely reusing existing mechanisms which are already in place for the Same-origin Policy [16] and Content Security Policy (CSP) [209]. We refer the interested reader to [189] for a full description of COWL and its implementation. Here, we instead remark on the challenge of browser-side confinement deployment.

Different from server-side, where platform developers can ensure that apps are implemented using Hails, we cannot impose that users download and use our custom browsers to reap the benefits of COWL. However, COWL is undergoing standardization and on the roadmap to be incorporated in browsers, by default [181]. Nevertheless, it is important to support legacy browsers, when possible.

When communicating with older browsers—browsers that do not support COWL but do implement CSP—Hails can provide some confinement guarantees at the cost of flexibility. Specifically, Hails can confine the content of a page according to the response label by associating a CSP header that whitelists the origins that content can communicate with [209]. Unfortunately, this weakens our trust model since CSP whitelists do not encompass navigation (or in-browser message passing), so even with CSP, a malicious VC could leak sensitive user data by navigating to a URL that encodes the sensitive information. Moreover, it also constrains application functionality: CSP is a discretionary access control mechanism [217] and thus not suitable for certain scenarios, such as mashups, where one needs to share sensitive data with a page whose label (and thus CSP header) is not compatible with the label of the data (see [189]).

2.4 Applications

In this section we describe several applications built with Hails, focusing primarily on the GitStar platform and its apps.

2.4.1 GitStar platform

We built and deployed GitStar, a Hails platform centered around source code hosting and project management. We and others have authored a number Hails apps for the GitStar platform. Below we detail some of these applications including the core management application, a code viewer, a follower/bookmarking application, a wiki, and a messaging system.

GitStar At its core, GitStar includes a basic MP and VC. The MP manages users' SSH public-keys, project membership, and project meta-data such as the project name and description; the VC provides a simple user interface for managing such projects and users.

Since Hails does not have built-in support for `git` or SSH, the GitStar platform includes an SSH server (and `git`'s transport utilities) as an external service. Our modified SSH server queries the GitStar VC via HTTP when authenticating users and determining access control permissions for repositories. Conversely, the GitStar MP communicates with an HTTP service atop this external `git`-repository server to access `git` objects.

GitStar allows users to create projects to which they can push files via `git`. Projects may be public (anyone can view or checkout repository contents) or private, in which case only specific users identified as *readers* or *collaborators* may access the project. In both cases, only collaborators may *push* contents to the project repository. GitStar provides an interface for managing these settings.

The rest of the platform functionality is provided by separately-administered, mutually-distrustful Hails applications, some of which were written by third-party developers. Each application is independently accessible through a unique subdomain. When a user "installs" an application in a project, GitStar creates a link on the project page that embeds an `iframe` pointing to the application. This gives third-party applications a first-class role in extending the user experience.

Code viewer One of the most useful features of source-code hosting sites is the ability to browse a project's code. We have implemented a code-viewing VC that allows users to navigate to different branches in a project's repository, view syntax-highlighted code, etc. Source code markup is done on the client-side using Google's prettify JavaScript library [74]. Additionally, if the source file is written in C or Haskell, the VC provides the user with an option to run static-analysis tools—respectively, splint [109] and hlint [134]—on the checked-in code.

Like all third-party applications, the code viewer is considered untrusted and accesses repository contents through the GitStar MP. When accessing objects in a private repository, the GitStar MP raises the VC's current label to restrict communication to authorized readers of the repository. Note

that this may also restrict the VC from subsequently writing to the database.

`git`-based wiki The `git`-based wiki displays Markdown files from the "wiki" branch of a project repository as formatted HTML. It uses the pandoc library [122] to convert Markdown to HTML. Like the code viewer, the wiki VC accesses source files through the GitStar MP, meaning it cannot show private wiki pages to the wrong users. This application leverages functionality originally intended for the code viewer for different purposes, demonstrating the power of separating policies from application logic.

Standalone wiki The standalone wiki is similar to the `git`-based wiki, except that pages are stored directly in a database rather than in files checked into `git`. To accomplish this, the developer wrote both an MP and a VC. The MP stores a mapping between project names and wiki pages. Wiki pages are labeled dynamically to allow project readers and collaborators—fetched via the GitStar MP—to read and write wiki pages. This is different from the `git`-based wiki in that it allows a more relaxed policy: readers can create and modify wiki pages.

Follower GitHub introduced the notion of "social coding," which combines features from social networks with project collaboration. This requires that a user be able to "follow" other users and projects. GitStar does not provide this feature natively, but a follower MP has been developed to manage such relationships. Users may now add the "bookmark" application (implemented as a VC that interacts with the follower MP) to their project pages, which allows other users to add the project to their list of followed repositories. This same application can be used to allow one user to follow other users.

Messenger The messenger application provides a simple private-messaging system for users. Its MP, as implemented by the developer, defines a message model and policies on the messaging data. The policy allows any user to create a message, but restricts the reading of a message to the sender and intended recipient. Interfacing with the MP, the messenger VC provides a page where users may compose messages, and a separate page where they may read incoming messages.

2.4.2 LearnByHacking platform

We built and deployed LearnByHacking, a Hails blogging platform in the style of School of Haskell [68]. LearnByHacking allows users to write articles (e.g., blog posts, tutorials, or lectures) that contain *active code* snippets, i.e., code that end users can modify and execute in the browser, without installing any software on their machine. Below we describe the LearnByHacking core and a commenting application built by another developer.

LearnByHacking At its core, LearnByHacking contains an MP and VC for managing user profiles, articles, and tags. The application allows users to collaborate on articles; until published, the MP ensures that articles can only be read and edited by its authors. Tags are public keywords that authors can associate with their articles, as to allow users to more easily subscribe to articles of interests.

The LearnByHacking VC provides authors with an interface for managing articles (e.g., collaborators, tags, published/draft state) and an interface for editing the article Markdown source, atop the CodeMirror in-browser editor [47]. The VC also renders articles, using the pandoc library [122], which users can read and interact with. Users can also subscribe to RSS/ATOM feeds (according to tags, users, etc.) to read articles offline.

To support active code, we extended Markdown code blocks with *directives*. Authors use these directive to specify the language of the code block, whether it is executable, a name for the block, and any other code blocks it depends on. The latter two directives allow authors to chain different code blocks to, for example, illustrate different execution sub-paths of a program to users (e.g., failure and success). The VC relies on our OS-level confinement mechanism to safely execute active code. We currently support code written in C, C++, JavaScript, Bash, and Haskell. However, extending LearnByHacking with additional languages is straight forward and, importantly, does not rely on any modifying any trusted code, i.e., the MP.

Commenter The commenter application provides a simple way for users of LearnByHacking to comment on published articles. To provide this feature, the developer defined an MP for storing comments. The MP policy ensures that the author of a comment is the only user allowed to modify their comment; comments are publicly readable. The commenter VC is embedded in published LearnByHacking articles as an iframe, providing users with ways for writing comments (both in response to an article and as replies to other comments), editing their comments, and viewing all the comments associated with the article. We remark that while the developer implemented the

application for LearnByHacking, the commenting system can easily be used by other applications as an alternative to Disqus commenting system [57]. Indeed a more tightly-integrated commenting system would have reused LearnByHacking's MP policy to allow for comments to unpublished articles.

2.4.3 λChair

We built a simple conference management system called λChair, inspired by EasyChair. λChair is a Hails application that can be used by conference chairs to manage paper submissions and reviews. The application defines data models and policies for handling users, papers, and reviews. After a user creates an account, the chair can add them to the committee, assign them to review papers, and manage their conflicts of interests. Logged-in users can only upload, view, manage conflicts, and edit their uploaded papers. Modifications are only allowed during the submission period. During the review period, committee members can participate in the review process by reading and writing comments and reviews for papers they are not in conflict with. Finally, after the review period, authors can additionally see their papers' reviews.

While λChair's functionality is relatively straight forward, the application is particularly interesting because of its security policy. Hails must ensure that only the *right* users are allowed to read and modify a particular paper or review, and this changes according to assignments, conflicts of interests, and conference state (in submission, review, or done). Indeed, we introduced side-effects into our policy DSL because of this use case; as further detailed in Section 2.7, the MP of our first λChair implementation was overly complex because the policy language was not flexible enough.

2.4.4 Taskr

Taskr is a task and project management application. Taskr was built by a group of three undergraduate students over a summer research internship. The students learned Haskell and Hails at the beginning of the summer, and built Taskr in the last month of their internship.

The application defines data models and policies for users to collaborate on projects through task assignment and a commenting system. Each project has one or more project leaders, who can change project settings, as well as project members, who may contribute to the project. Any member may add a task or comment on a project, but only leaders can make administrative changes to a project, such as changing the membership list.

Projects may be public, in which case anyone can view them, including non-members. Alternatively, private projects are only visible to their members.

Hails enforces all of these policies in a single place: the MP. We found that, although the students building the application were novices in both Haskell and Hails, once the policy was well-defined, they were able to build the rest of the application rapidly. In Section 2.6.3 we discuss our improvements to Hails based on feedback from these students.

2.5 Design Patterns

In this section, we detail the applicability of some existing security patterns within Hails, and various design patterns that we have identified in the process of building the applications described in Section 2.4.

Minimizing privilege usage Since MPs are trusted by users to protect the confidentiality and integrity of their data, a well-designed MP should be programmed defensively. For example, an MP should treat all invoking VCs as untrusted, including ones written by the same author. This minimizes the damage that any VC—whether malicious or vulnerable—could have on user data.

The easiest way to program defensively is to minimize the use of MP privileges and avoid granting privilege, for example, to other components unnecessarily, i.e., follow the principles of least privilege and privilege separation [170]. When doing so, VCs that access the MP's database will only be able to fetch data that the end user can observe. This is in contrast to having MP privileges, which provide unfettered access to its data. Without privileges, VCs are also restricted to inserting already-labeled documents (see Section 2.2.3) on behalf of users, i.e., without privileges VCs can typically only insert data endorsed by the Hails HTTP server on behalf of the user, as opposed to arbitrary data (when using the MP privilege).

Trustworthy user input Since VCs can craft arbitrary HTTP requests, VC-constructed documents should not always be trusted to represent the user's intentions. Hence, MPs should ensure that VCs cannot arbitrarily insert or modify data on behalf of users. To this end, MPs should, as discussed above, minimize privilege usage and, moreover, they should set policies that disallow code from acting on behalf of users. An example of such a policy is the policy on user documents given in Figure 2.3. The policy specifies that only `alice` (or the GitStar MP) is allowed insert and modify documents with the username set to `alice`. Since Hails does not grant user privileges to any code,

a VC, even one handling a request from `alice`, is disallowed from constructing and inserting a document with the username set to `alice` (e.g., the document of Figure 2.2), without `alice` or the MP first endorsing it.

We, however, need to allow VCs to insert, modify, and delete user data when requested by the user to do so. To this end, the Hails HTTP server endorses requests on behalf of users before invoking a VC's main controller. This reflects the fact that requests may convey the user's intent. Since VCs cannot directly manipulate requests (e.g., to transform them into arbitrary database actions) without stripping off their integrity labels, Hails also provides a library for transforming labeled requests into labeled documents. MPs may use this library to expose transformers to VCs. These transformers take, as input, user-endorsed requests and return MP-endorsed documents that VCs may, in turn, insert into the database (potentially in place of an existing document).

In practice, MPs typically inspect requests before transforming them into labeled documents. This avoids the need to completely trust VCs to construct HTTP requests that reflect the user's intentions. Indeed an MP may choose to only transform requests from VCs it trusts, or from VCs the user has approved. Nevertheless, we recognize this as a limitation of our approach—since VCs ultimately interface with users they cannot be considered completely untrusted. We, however, remark that policies, such as that of Figure 2.3 prevent a VC trusted only by `bob` from modifying `alice`'s data. Moreover, using this design pattern further proved to be useful in reducing and reasoning about the attack surface of Hails applications: a VC cannot perform arbitrary actions that may result in data corruption without going through an MP filter.

Partial update The trustworthy user input pattern is suitable for inserting and updating documents in whole; it is not, however, directly applicable to partially updating documents. And, in many cases, it is desirable to update only certain elements of a document. For example, in GitStar users sometimes need to (only) update their SSH keys, while in λChair authors need to be able to update a paper's title and abstract without uploading a new PDF. Updating a whole document is both error prone, since it requires developers to include all document fields with every HTML form, and inefficient, since it requires sending all the data from the server to the browser with every form.

Fortunately, the HTTP request PATCH method precisely addresses this issue [60]; in contrast to PUT, which is used by browsers to indicate that a document should be replaced with the supplied resource, PATCH is used to convey partial modifications to a stored document. Naturally, Hails VCs use HTTP request methods to convey the user's intent to MPs (GET for fetching a document, POST for creating a new document, PUT for updating a document, PATCH for partially updating a

document, and DELETE for deleting a document). However, since many browsers still only support a subset of these methods, we still rely on request bodies to convey this information. In particular, when partially updating a document, we found that a partial document that contains the newly-updated fields, the document keys, and a token `$hailsDbOp` indicating the operation (PATCH, in this case) is sufficient for the MP to update an existing document. This partial document must be endorsed by the user or MP, by, for example, applying the previous pattern, before the VC attempts to update the corresponding persistent document. Whenever a VC invokes an MP to carry out a partial update, the MP first verifies that the user is aware of the update by checking the presence of the operation token `$hailsDbOp`. Next, the MP uses the keys to fetch the stored document. Finally, it merges the newly-updated fields into the original document and writes the document to the database, imposing restrictions similar to those of Section 2.2.3.

Delete We have found that most applications require a pattern similar to the partial update pattern when deleting documents. To delete a document, a VC invokes the appropriate MP with a document containing the target-document's keys and an operation token indicating a delete, i.e., `$hailsDbOp` set to DELETE. As in the partial update, this document must be endorsed by the user or MP by applying the trustworthy input pattern. When invoking an MP with such a labeled document, the MP simply inspects it and removes the target document.

Privilege delegation Hails provides a call-gate mechanism, inspired by [228], with which code can authenticate itself to a called function, i.e., prove possession of privileges, without actually granting any privileges to the called function. One use of call gates is to delegate privileges. For instance, an MP can provide a gate that simply returns its own privilege, on the condition that it was called by a particular VC.

An early version of GitStar relied on privileged delegation to allow the GitStar SSH server to read the SSH keys stored in the GitStar MP database. Specifically, the GitStar (project management) VC used a call gate to retrieve the GitStar MP privilege when looking up project readers and collaborators on behalf of the GitStar SSH server. We've since refactored this code to not use privilege delegation. Instead of relying on the MP privilege, we created a dedicated user account for the SSH server and added this principal as a reader to the GitStar project collection policy.

While we have managed to avoid privilege delegation in most applications we developed, changing policies as in GitStar is not always possible and privilege delegation may prove necessary. We

especially foresee this being useful for non-platform applications, i.e., applications written by a single team, at the cost of placing more trust in VCs. Indeed, we've built several simple applications, similar to those of Section 2.4, but using the call gate mechanism to retrieve a privilege corresponding to the current user;[3] in most cases, the code turned out to be simpler than the corresponding code without privilege—but, of course, at the cost of trusting VC code to properly utilize the user's privilge.

2.6 Evaluation

We evaluate Hails on three dimensions:

1. **Performance**: we compare the performance of the Hails framework against existing web frameworks.
2. **Security**: we give measurements of the TCB sizes of Hails applications we and other have built as rough estimates of the applications' attack surfaces.
3. **Usability**: we report on our experience and the experience of application authors not involved in the design and implementation of the framework in building Hails applications.

Below we describe our methodology and evaluation results. We refer the interested reader to [189] for a detailed performance evaluation of our client-side confinement system, COWL.

2.6.1 Performance Benchmarks

To demonstrate how Hails performs in comparison to other widely-used frameworks, we present the results of four micro-benchmarks that reflect basic operations common to web applications. Figure 2.4 shows the performance of Hails, compared with:

- Ruby Sinatra framework [175] on the Unicorn web server. Sinatra is a common application framework for small Ruby applications and APIs (e.g., the GitHub API is written using Sinatra).

[3] Since MPs do not have access to user's privilege, the privilege corresponding to the current user is actually an MP subdivded privilege (e.g., `_GitStar` $\vee$ `alice`).

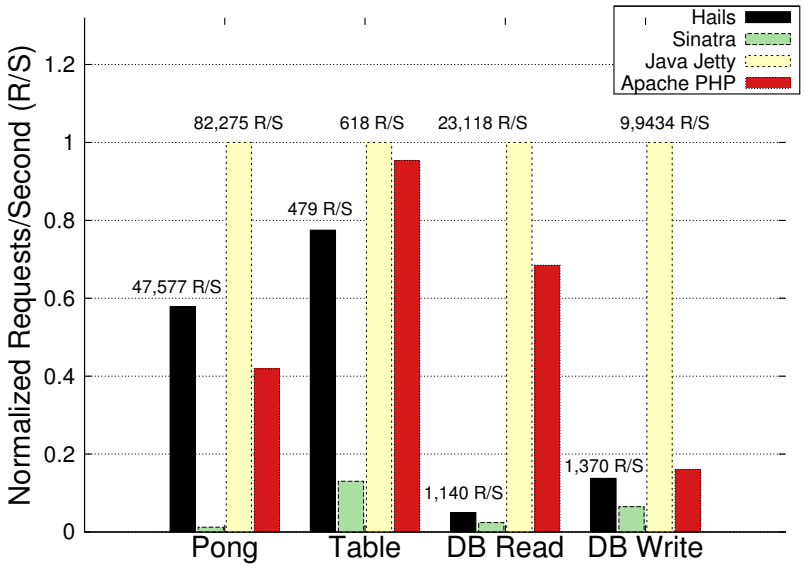

Figure 2.4: Micro-benchmarks of basic web application operations. The measurements are normalized to the Java Jetty throughput. All database operations are on MongoDB.

- PHP on the Apache web server with `mod_php`. Apache+PHP is one of the most widely deployed technology for web applications, including WordPress blogs, Wikipedia, and earlier versions of Facebook.

- Java on the Jetty web server [139]. Jetty is a container for Oracle's Java Servlet specification, and is widely used in production Java web-applications including Twitter's streaming API, Zimbra and Google AppEngine.

We use `httperf` [140] to measure the throughput of each server setup when 100 client connections continuously make requests in a closed-loop—we report the average responses/second. The client and server were executed on separate machines, each with two Intel Xeon E5620 (2.4GHz) processors, and 48GB of RAM, connected over a Gigabit local network.

In the Pong benchmark the server simply responds with the text "PONG". This effectively measures the throughput of the web server itself and overhead of the framework. Hails responds to 1.7× fewer requests/second than Jetty. However, the measured throughput of 47,577 requests/second is roughly 28% and 47× higher than Apache+PHP and Sinatra, respectively.

In the Table benchmark, the server dynamically renders an HTML table containing 5,000 entries, effectively measuring the performance of the underlying language. Hails respectively responds to 30% and 23% fewer requests/second than Jetty and Apache+PHP, but 6× more than Sinatra. Hails

Application	Trusted MP code	Untrusted app code	Third-party library code
GitStar manager	251	1,590	54,109
GitStar code viewer	0	1,454	66,786
GitStar `git`-wiki	0	859	66,258
LearnByHacking	224	1,352	96,727
Commenter	34	212	46,685
Taskr	90	894	37,879
λChair	132	613	33,769

Table 2.1: Application line count, broken down into the amount of code that is essentially trusted (the MP code), the application code that is not trusted (i.e., essentially the VC code), and third-party libraries. We only give the line count for the subset of applications that have been deployed. We count every line in a Haskell source file, whether it is ultimately execute in the application or not.

is clearly less performant than Jetty and Apache+PHP for such workloads, even though Haskell should be faster than PHP at CPU workloads. We believe that this is primarily because Hails does not allow pipelined HTTP responses, so a large response body must be generated in memory and sent in its entirety at once (as opposed to sent in chunks as output is available). Nonetheless, Hails responds to 6$\times$ more requests/second than Sinatra.

The DB Read and DB Write benchmarks compare the performance of the read and write database throughput. Specifically, for the DB Read benchmark the server responds with a document stored in the MongoDB, while for the DB Write the server inserts (with MongoDB's `fsync` and `safe` settings on) a new document into a database collection and reports success. Like the Ruby library, the Haskell MongoDB library does not implement a connection pool, so we loose significant parallelism in the DB Read workload when compared to Jetty and Apache+PHP. In the DB Write workload, this effect is obviated since the `fsync` option serializes all writes.

2.6.2 Evaluating the attack surface of Hails applications

Our confinement mechanisms have underlying theoretical foundations with accompanying proofs of strong security properties. For example, we proved that programs written in LIO and COWL satisfy non-interference [187, 185, 82], i.e., they cannot leak or corrupt user-sensitive data. In contrast to most dynamic language-level confinement systems, our formal models consider real-world language features (e.g., exceptions and threads) and our theorems hold even in the presence of typical covert channels, such as the termination, internal timing [183, 82], and even hardware cache-timing

channels [182].

Unfortunately, these results do not trivially extend to Hails applications. This is primarily because MPs (and VCs) rely on privileges to accomplish their tasks and our formal language models, like most results in this area, do not account for privileges. We consider such extensions to our formal models as part our future work. Here, we evaluate the attack surface of typical Hails applications more qualitatively, under the trust assumptions of Section 2.2.6.

In particular, we report the TCB line count of some of the Hails applications described in Section 2.4. Unlike traditional web frameworks, where the TCB of an application is essentially the whole application codebase, including the third-party libraries it depends on, the TCB of a Hails application is limited to the MP policy code and any code that uses MP privileges. Since a bug in any of this code could potentially be a vulnerability, we consider this to be the attack surface the application. This is precisely the reason we encourage developers to use the privilege separation patterns of Section 2.5.

Table 2.1 gives the line counts (of Haskell code) for our applications; these numbers include the application TCB size, i.e., the trusted MP code, and the untrusted application code, which entails the VC code and third-party libraries. We remark that in general the amount of code that developers must get right is relatively low—on the oder of a few hundred lines of Haskell—especially when considering the rest of the application code, which orders on tens of thousands of lines of Haskell. More interestingly, we remark on the evolution of the GitStar and LearnByHacking platforms. In both cases the applications grew with new features and functionalities, respectively adding over 100,000 and 40,000 lines of code, but the attack surface grew sub-linearly and remained under 300 lines of code.

2.6.3 Experience report on building Hails applications

When building systems such as Hails it is important to also consider its usability. Indeed, it is important to consider both, usability and security, jointly when designing systems, since addressing traditional usability concerns alone can negatively impact security and vice versa [96]. We iterated on the Hails design and the underlying confinement mechanisms to address usability issues that arose when we and other developer were building applications. In the process, we gathered experience reports from seven developers, four of which we interviewed as part of a small, mostly informal, usability analysis. We remark that none of the developers had experience building web applications in Haskell. Their reflections validate some of the design choices we made in Hails, as well as highlight some ways in which we could make Hails more usable. Below we summarize these

reports and our own experience building platforms and applications. We refer the interested reader to [185] for details on the LIO design evolution, which was largely influence by our work on Hails.

Understanding the security model We conjectured that MAC and the separation of code into MPs and VCs leads to building applications for which it is easier to *understand* and reason about security. To measure understanding we compared the mental model of the developers with our expert model, in the style of [96]. Beyond validating their understanding though discussion, their understanding of policies was also reflected in the code they wrote code. For example, their VC code often gracefully handled failures due to policy.

More broadly, we found that developers also understood the implications of the MPVC paradigm on security, i.e., that MP code is security critical and that VCs need not trusted to enforce policy. This was validated by the application authors who remarked that although "experienced developers [need to] write the tough [MP] code and present a good interface," when compared to frameworks such as Rails, not having to "sprinkle [security] checks in the controller" made it easier to be sure that "a check was not missing." With Hails, they, instead, "spent time focusing on developing the [VC] functionality." Indeed, we found this to also be reflected in the code they produced—most developers, ourselves included, did not include checks in VCs that are typical to web apps (e.g., *can these actions be performed by the current user?*). In fact, most applications written in Hails would traditionally be susceptible to the mass-assignment vulnerability that affected GitHub [155, 110], or the access control check vulnerabilities that affected Facebook [100] and United [59]. But, because our developers specified MP policies that would disallow one user from impersonating another, and because policies are enforced in a mandatory fashion, such "bugs" have no security implications in Hails—exploiting such bugs typically revealed UI bugs, which arguably, should not even be addressed.

Policy specification usability Implementing MPs using an early version of Hails proved to be challenging for most of the developers. In this version, we did not have the declarative DSL of Section 2.2.3; instead, the "policy specification" entailed imperatively labeling the different database components (collections, documents, etc.). While developers were typically effective in devising policies for a model, the API for implementing policy modules was difficult to learn, inefficient, and error prone. Unsurprisingly, the policy code was hard to understand, also negatively impacting the previous usability factor—understanding the security model.

To address this, we designed the DSL described in the original paper [72], similar to a subset

of the DSL given in Section 2.2.3. We found that this DSL, while less flexible than imperative code, makes policy specification simpler and more efficient—developers were able to specify correct policies in less time. Equally important, developers found it easier to understand what policy an MP was enforcing and thus make a more informed decision when deciding to use the library. Moreover, this also improved Hails' satisfiability—developers were no longer dreading writing MP code.

Unfortunately, since [72], we found the DSL to still have some usability-security concerns. First, we found that new developers misunderstood the security implications of declaring fields to be keys. For example, one blog developer marked the body fields of articles as keys, despite specifying a relatively strict document (article) policy. This was motivated by their want to implement a feature that would allow full-text searching over articles. Unfortunately, this was not an isolated instance; we found that developers that do not (yet) understand the intricacies of our database model and are mostly motivated to implement features, are unlikely to be cautious when declaring a field to be indexable, especially if they've already specified a policy for a document. To address this, we modified our DSL to use the modifier `publicIndex` instead of `key` when declaring a field to be a key; this small change imposes fewer assumptions on the developer, i.e., that they know that declaring fields as keys has security implications, and makes it easier for them to be vigilant about not declaring fields "public."[4]

Second, we found that restricting DSL policies to pure functions negatively affects efficiency and understandability. In many cases purely-functional policies are sufficient and easy to reason about. However, when a policy relies on data not present in the document supplied as an argument, developers would have to resort to using a level indirection to accomplish their goal. Consider an example from our λChair implementation. In λChair, we needed to specify that a paper can be read by any committee member not in conflict with the paper. But, since committee members and conflict of interest relationships are stored in different collections, we didn't have access to the data and ended up encoding this information in the labels themselves. For example, `alice`'s submission, whose ID is 1, would initially be labeled $\langle \texttt{alice} \vee \texttt{\#paper} : 1 \vee \texttt{\_Chair}, \texttt{alice} \vee \texttt{\_Chair} \rangle$. The λChair MP would then use a transformer to relabel papers by expanding group principals (e.g., `#paper:1`, above) into label formulae (e.g., `john` ∨ `claire`, the PC members not in conflict with `alice`) before returning control to the invoking VC. This proved to be both inefficient and overly complicated—most developers had a hard time understanding the additional level of indirection.

To address this, we extended the DSL with the `run` keyword which allows an MP to execute

[4] Key are protected by the collection label. But, collection labels in many cases correspond to the public label.

arbitrary Hails code in the policy. When used to compute document or field labels, `run` restricts the computation clearance to to the MP collection label. (Similarly, `run` sets the clearance of its computation to the database label when computing collection labels as such.) This ensures that the labels are not computed from overly sensitive data. We imposed these semantics to demotivate MP developers from using sensitive data in labels.[5]

We remark that our primary reason for originally making DSL policies pure functions was to make it easy for developers to specify and understand policies; the extension with `run` is very much still in line with this reasoning. Indeed `run` makes it easy to specify complex policies in a straight forward manner. Moreover, it extends the usage of the DSL to use cases where imperative code would otherwise be required: specifying dynamic policies on collections. This, for example, makes it possible for λChair to support multiple conference while ensuring isolation between the different conferences' content.

Usability of framework libraries Given the task of building VCs and MPs, we learned that developers found Hails to be an effective framework, i.e., they were able to build the applications at hand. But, in addition to addressing the above usability concerns, we refactored several APIs, which are primarily used by VCs, to address various usability concerns (e.g., efficiency and satisfaction) that arose during the development process. For example, to improve the efficiency of VC development, we extended our database APIs to provide developers with functionality that was necessary in common tasks—e.g., filtering results based on the VC's clearance and unlabeling documents before returning them. Similarly, we introduced support for template frameworks to address concerns raised by developers who found that our lack of "scaffolding tools for generating boiler-plate code [and] a template framework" impedes the development process, when compared to frameworks such as Rails. We are still working on improving the Hails development experience as a whole, currently focusing on developers' want for better documentation, recipes, debugging information, and scaffolding tools that will make it easier to build both VCs and MPs.

We remark that, while we believe that Hails is a usable framework, i.e., *it can be used to achieve specified goals with effectiveness, efficiency, and satisfaction* [92, 173], our conclusion is drawn from external adoption of our systems and the short usability analysis we conducted.[6] We believe

[5] They can still read sensitive data, but have to go through the additional step of raising the clearance.

[6] Our MAC-based confinement mechanisms are seeing some adoption in both academia (e.g., LIO and COWL have been used in several university courses) and industry (e.g., COWL is a spec at the W3C while Hails and LIO are used in a commercial product at GitStar, Inc.). We are actively trying to incorporate feedback from the different use cases to

that a more formal and extensive usability analysis (e.g., one that defines small tasks for users to accomplish) is an important future research goal that can lead to a more usably-secure framework. In retrospect, we should have consulted HCI experts throughout the design process.

2.7 Discussion and Limitations

In this section, we discuss the ramifications of the design and implementation of Hails and suggest solutions to some of its limitations.

OS-level confinement Since [72], both Docker and CoreOS have released products that use Linux isolation mechanisms, namespacing, and cgroups to isolate Linux applications from each other. Their underlying approach is very similar to ours (see Section 2.3.2). Until recently, however, neither was an appropriate replacement for our OS-level confinement (e.g., Docker did not originally provide different user namespaces [231, 150]). Indeed, because of their focus on providing app-deployment solutions, both Docker and CoreOS/rkt are more complicated than our solution (and the reason for some of their vulnerabilities [118]) and not well-suited to to be used as throw-away containers.[7] Nevertheless, we believe that using one of these solutions in place of our current OS-level confinement mechanisms can have numerous benefits. Usability is the most obvious. But, we can also leverage their recent support for AppArmor and SELinux (see [71, 58]) to make the confinement more flexible (e.g., to allow an OS-level confined process to communicate with some remote servers); our OS-level confinement is unnecessarily restricting in disallowing any network communication or shared file system access.

Browser-level confinement In Section 2.2.4 we discussed the current status of our browser confinement system, COWL, and an alternative approach to providing a limited form of confinement with CSP for older browsers. A different, but complimentary, approach to both would be to rewrite VC output at the server-side before sending it to the client, neutralizing data-exfiltration risks. While such content-rewriting used to be considered a dangerous proposition (e.g., because tools implemented by Google [127], Yahoo [49], Facebook [64], and Microsoft [91] proved to all have vulnerabilities [124]), most browsers now have JavaScript engines that support ECMAScript 5 (ES5)

improve the systems.

[7] Their typical use case is long-lived applications, i.e., servers. This is different from ours, wherein containers are typically short lived, usually for the duration of a request.

strict mode, which makes the prospect of safe re-writing far more tractable. In addition, client-side tools with solid theoretical foundations, such as SES [193], which builds on ES5 strict mode, and DJS [21], have shown promise in confining and isolating increasingly complex applications which contain untrusted code. We leave the exploration of using such tools and output re-writing to future work.

Query interface Hails queries are predicates on keys. By separating keys from the other fields, the decision to permit a query is simple: if a Hails component can read from the database collection, it may perform a key-based query. This limited interface is sufficient for many VCs, which may perform more complex queries on other, labeled fields by inspecting them in their own execution contexts.

For large datasets, better performance would result from filtering on all relevant fields in the underlying database system itself. Additionally, this would obviate the need to reason about the security semantics of keys. However, providing this more-general interface to a Hails application would require sensitivity to label policies inside the query engine. Since most of the persistence layers we consider (e.g., MongoDB and PostgresSQL) allow developers to plug in custom logic in their engines, we believe that compiling policy to code to run in the database layer is viable and a useful improvement on our implementation.

We, remark, however, that since our policy DSL allows developers to execute arbitrary code in `run` blocks, this may introduce some challenges. For example, if the MP code was fetching data from a remote server in the policy code, it may not be sensible for such code to execute in the query engine. Luckily, Haskell's monads makes it easy to define sub-languages within Haskell. This allows us to, for example, restrict (certain) `run` blocks to only be composed of pure code and database queries as to facilitate compilation to query engines.

Side-effecting policy code The ability to execute arbitrary code in policy code has several trade-offs, as as discussed in Section 2.6.3. For example, with the introduction of `run`, it is now possible for one developer to define a policy that depends on data from another MP, the policy of which in turn depends on the first developer's MP. Unlike non-terminating pure code, debugging such a policy may prove to be more difficult. Similarly, it is possible for developers to carelessly increase an application's latency by performing network requests (without short timeouts) and other IO within the policy code. While we have not run into such cases in practice, we foresee the need for static analysis to help eliminate such bugs in policy code. Indeed, even for purely-functional policy code,

static analysis (e.g., in the form of refinement types [162]) can be used to eliminate certain kinds of bugs (e.g., VCs trying to read from a collection whose static label restricts read access to the MP).

Policy as code When code wishes to access data sored in the database, the Hails database layer will always invoke the corresponding MP's policy code to compute labels. An alternative approach could have serialized labels alongside data (e.g., as in HiStar [228] and LIO's file system). This has the benefit of allowing code—and, in particular, code written in another language—to access labeled data without invoking MP code. Unfortunately, this also has many drawbacks that Hails' approach does not. For example, in taking this approach Hails policies can be specified in a single location in a high-level, generic fashion that applies to all the documents in the collection. This not only makes it easy and less error prone to specify policy, but also easier to inspect and understand an application's policy—reasoning about security policy in terms of the many serialized labels is very difficult. More importantly, changing a policy simply amounts to changing the MP policy specification code, updating the MP's version in the platform's global configuration file, and re-linking affected apps—it importantly does not require relabeling data stored in the database, a process that is both less efficient and safe. Nevertheless, allowing applications written in different languages to access labeled is important, especially in enterprise scenarios. Hence, the compilation of queries to an intermediate form that can be interpreted in different languages (e.g., by JavaScript in MongoDB) seems like an interesting balance.

Deployment We have described the deployment of mutually-untrusted software components on a particular web "platform," GitStar, which simply provides the minimal policy modules and apps necessary to coordinate shared data and to unify the user interface. But there is nothing to prevent a constellation of such platforms being deployed together, providing a rich ecology of functionality in which third-party apps could thrive; for example, social-networking apps could operate on data managed by disparate platforms. Services such as App Engine and Heroku have already shown that many developers are willing to trust a centralized hosting provider to run their applications. Indeed, an interesting direction for future work may be to implement a Hails platform wherein MPs simply re-expose users' Facebook, Twitter, etc. data, which is available via their platform APIs, by attaching labels to it. Such a hosting platform would empower users by allowing them to use third-party apps that Hails ensures to protect their privacy.

DoS attacks The Hails web framework does not address address denial-of-service (DoS) attacks in the form of resource exhaustion or heavy network load, except when apps execute external programs. For example, we do not restrict an app from participating in a distributed DoS attack by sending many HTTP requests to a target host. Such concerns are platform-specific and outside the scope of our web framework. We, however, remark that Hails composes well with existing OS-level isolation, resource management and workload distribution mechanisms since each Hails app is run as a separate process.

2.8 Related Work on Server-side Web Application Security

Information flow control and web applications A series of work based on Jif addresses security in web applications. Servlet Information Flow (SIF) [45] is a framework that allows programmers to write their web applications as servlets in Jif. Swift [44], based on Jif/split [226, 233], compiles Jif-like code for web applications into JavaScript code running on the client-side and Java code running on the server by applying a clever partitioning algorithm. SIF and Swift do not support information flow control involving databases or untrusted executables as Hails does. However, we believe that their partitioning approach could potentially be used by Hails to confine JavaScript in legacy browsers.

Ur/Web [42] is a domain specific language for web application development that includes a static information flow analysis called UrFlow. Policies are expressed in the form of SQL queries and while statically enforced, can depend on dynamic data from the database. Security can also be enforced on the client side in a similar manner to Swift, with Ur/Web compiling to both the server and client. A crucial difference from Hails is that Ur/Web does not aim to support a platform architecture consisting of mutually distrustful applications. Moreover, Hails is more amendable to extensions such as executing untrusted binaries or scaling to a distributed setting.

Logical attestation [176] allows specifying a security policy in first-order logic and the system ensures that the policy is obeyed by all server-side components. This system was implemented as a new OS, called Nexus [176]. Hails's DC labels are similar to Nexus' logical attestation, but based on a simpler logic: propositional logic. A crucial difference between Nexus and Hails is that Hails provides fine grained labeling and a framework for separating data-manipulating code from other application logic at the language level. For a web framework, fine grained policies are desirable; the language-level approach used by Hails also addresses the limitations of cobufs used in Nexus, as discussed in [176]. Moreover, requiring users to install a new OS as opposed to a library is not

always feasible. However, Nexus is very much complimentary: a Hails platform could potentially use Nexus to execute untrusted executables in an environment that is less restricting that our OS sandbox (e.g., it could have network access as directed by Nexus).

Laminar [163] combines operating systems and programming languages IFC techniques to jointly provide application and OS end-to-end guarantees. At the application level, Laminar enforces IFC within certain code regions named *security regions*—where labeled data can be accessed. This is similar to our underlying confinement system—LIO, as described in Chapter 6—which enforces IFC at the thread level. Unlike Hails, Laminar does not extend enforcement to the browser nor does it address a challenge that underlies most IFC systems: how to specify policy. However, their OS confinement approach is considerably more flexible than ours; as with Nexus, using Laminar's OS-level confinement in Hails would be an interesting direction.

Another closely related work is Jeeves [220]. Jeeves is a language-level IFC system that separates policy specification from the rest of the program that implements functionality. This is similar to our approach of separating applications into MPs, where policy and data models are specified, and VCs, where the app functionality is implemented. They, however, impose this design principle in the language design, whereas Hails imposes this in the framework, atop LIO. This has allowed us to change the policy specification language over time to address usability factors—and, importantly, still allows others to use wholly different policy languages. Moreover, this design choice has allowed us to iterate on the design of LIO itself; in contrast to Jeeves, and most dynamic language-level IFC systems, LIO proves a stronger security theorem (*termination-sensitive noninterference*), supports advanced language features (e.g., threads, exceptions, recovery from IFC failures, etc.), provides good performance (as demonstrated by Hails) and does not impose new language semantics.

In Jeeves, programmers that implement functionality do so by writing policy-agnostic code. This is a desirable property since it would mean, in the context of Hails, that a VC developer would not need to be aware of the policies specified by an MP. And indeed, a subset of the Hails database API does handle labels transparently (e.g., `findOne` unlabels documents) to hide some details from the developer. However, our experience implementing web applications so far suggests that programmers *need* to be aware of and able to inspect labels. This is needed, for instance, to cleanly handle policy violations when they occur. It would, however, be an interesting exploration to extend Hails to support a programming model closer to Jeeves and refining what the balance is between exposing and abstracting policy mechanisms. This could be done by building on the work of Austin *et al.* [13].

Jeeves was recently used in the Jacqueline web framework [219], which allows Jeeves policies

to be specified on data stored in a database. As discussed in Section 2.7, Hails does not yet support this—all application code must interface with an MP. Jacqueline's guarantees, however, do not extend to the browser or OS.

Another closely related work is W5 [105]. Similar to Hails, they propose a separation of user data and policies (MPs), from from the application logic (VCs). Moreover, they propose an architecture that, like Hails, uses IFC to address issues with current web site architectures. W5's design is structured around OS-level IFC systems. This approach is less flexible in being coarser grained, but, like Nexus, complimentary. A distinguishing factor from W5 is our ability to report on the implementation and evaluation of a system that has been used in production.

Secure web frameworks OKWS [103], Diesel [66], and Radiatus [40] are web frameworks that use privilege separation and least privilege to reduce damages due vulnerable app code. OKWS applies privilege separation to application services, Radiatus to users, and Diesel focuses on separating database access rights. Unlike IFC, the mechanisms underlying these frameworks provide weaker guarantees. For example, they assume that applications are written by developers that have the user's best interests at heart and thus cannot protect against untrusted code or code injection attacks.

Resin [222] allows developers to specify data flow policies that the runtime then enforces on application code. This is sufficient for many applications, but like the aforementioned (non-IFC) secure frameworks, Resin provides weaker guarantees and is not appropriate for platform deployments.

Passe [25] isolates applications (into controllers) and enforces data and control-flow dependencies between the app, browser and database. The system's guarantees are not as strong as Hails' IFC guarantees and its somewhat complex training phase make its difficult for Passe to be applied to platforms. However, Passe can be used to secure existing applications, whereas Hails and most other frameworks require developers to change their applications. Moreover, Passe infers an application's policy from tests and does not require developers to a-priory specify policies. We believe that it is possible for us to infer policies from applications traces (with most-permissive policies) in much the same way, but we leave this to future work.

Mylar [153] is a web framework that provides data confidentiality and authenticity even when a server is under the control of an attacker. Mylar relies on CryptDB [152] to protect data stored in the database and assumes that code running in the browser is trusted. Mylar is mostly complimentary to our approach. In particular, we believe that COWL can be used to address the deficiencies

of Mylar—that of leaking data client-side—while Mylar and CryptDB can be used to reduce (or eliminate) the need to trust MPs in Hails, which for certain applications is very desirable.

Trust management Trust Management is an approach to distributed access control and authorization, popularized in [27]. Related work includes [2, 26, 54, 112, 111]. One central idea in trust management, which we follow in the present chapter, is to separate policy from other components of the system. However, trust management makes access control decisions based on policy supplied by multiple parties; in contrast, our approach draws on information flow concepts, avoiding the need for access requests and grant/deny decisions.

Persistent storage Li and Zdancewic [113] enforce information flow control in PHP programs that interact with a relational database. They statically indicate the types of the input fields and the results of a predetermined number of database queries. In contrast, Hails allows arbitrary queries on keys and automatically infers the security levels of the returned results.

Extending Jif, Fabric [119] is an IFC language that is used to build distributed programs with support for data stores and transactions. Fabric safely stores objects, with exactly one security label, into a persistent storage consisting of a collection of objects. Different from Fabric, Hails store units (documents) can have different security labels for individual elements. Like Fabric, Hails can only fetch documents based on key fields.

BStore [38] separates application and data storage code in a similar fashion to Hails's separation of code into VCs and MPs. Their abstraction is at the file system granularity, enforcing policies by associating labels with files. Our main contribution provides a mechanism for associating labels with finer grained objects—namely Haskell values. We believe that BStore is complimentary since they address similar issues, but on the client side.

SeLINQ [171], IFDB [172], and [120, 121] enforce information flow control in database systems. Most of these works are complimentary. One important aspect of making Hails usable is the policy DSL (which is tied to the data model); the dependent types approach of [120, 121] have a similar data model and policy application approach to ours, but in a static setting. Using database systems such as these would potentially allow Hails applications to fetch and store data without invoking MP code and thus facilitate multi-language platforms.

2.9 Conclusion

Ad hoc security and privacy mechanisms based on access control lists are an awkward fit for modern web frameworks that must protect sensitive user data while incorporating third-party apps. To address this, we developed Hails, a framework for building web platforms that applies confinement mechanisms at the language, OS, and browser levels, allowing mutually-untrusted apps to interact safely. Because the framework promotes information flow policies to first-class status, platform and app authors may specify policy concisely in one place and be assured that the desired constraints on confidentiality and integrity are enforced in a mandatory fashion across all components in the system, whatever their quality or provenance.

As a demonstration of the expressiveness of Hails, we built a production system, GitStar, whose central function of hosting source-control repositories with user-configurable sharing is enriched by various third-party apps. Beyond GitStar, we built several other platforms using Hails and enlisted several third-party developers to build VCs and MPs for these platforms. These experiences demonstrate the ability of Hails to support a platform consisting of mutually-distrustful apps written by numerous authors, where flexible security policies, as required by real-world users, can nevertheless be enforced.

Chapter 3

Protecting Users by Confining JavaScript with COWL[1]

Modern web applications are conglomerations of JavaScript written by multiple authors: application developers routinely incorporate code from third-party libraries, and *mashup* applications synthesize data and code hosted at different sites. In current browsers, a web application's developer and user must trust third-party code in libraries not to leak the user's sensitive information from within applications. Even worse, in the status quo, the only way to implement some mashups is for the user to give her login credentials for one site to the operator of another site. Fundamentally, today's browser security model trades privacy for flexibility because it lacks a sufficient mechanism for *confining untrusted code.* This chapter presents COWL, a robust JavaScript confinement system for modern web browsers currently under standardization at the W3C. COWL introduces label-based mandatory access control to browsing contexts in a way that is fully backward-compatible with legacy web content. We use a series of case-study applications to motivate COWL's design and demonstrate how COWL allows both the inclusion of untrusted scripts in applications and the building of mashups that combine sensitive information from multiple mutually distrusting origins, all while protecting users' privacy. Measurements of two COWL implementations, one in Firefox and one in Chromium, demonstrate a virtually imperceptible increase in page-load latency.

[1] This chapter is a copy of the OSDI 2014 paper [189].

3.1 Introduction

Web applications have proliferated because it is so easy for developers to reuse components of existing ones. Such reuse is ubiquitous. jQuery, a widely used JavaScript library, is included in and used by over 77% of the Quantcast top-10,000 web sites, and 59% of the Quantcast top-million web sites [156]. While component reuse in the venerable desktop software model typically involves libraries, the reusable components in web applications are not limited to just JavaScript library code—they further include network-accessible content and services.

The resulting model is one in which web developers cobble together multiple JavaScript libraries, web-based content, and web-based services written and operated by various parties (who in turn may integrate more of these resources) and build the required application-specific functionality atop them. Unfortunately, some of the many contributors to the tangle of JavaScript comprising an application may not have the user's best interest at heart. The wealth of sensitive data processed in today's web applications (*e.g.,* email, bank statements, health records, passwords, *etc.*) is an attractive target. Miscreants may stealthily craft malicious JavaScript that, when incorporated into an application by an unwitting developer, violates the user's privacy by leaking sensitive information.

Two goals for web applications emerge from the prior discussion: *flexibility* for the application developer (*i.e.,* enabling the building of applications with rich functionality, composable from potentially disparate pieces hosted by different sites); and *privacy* for the user (*i.e.,* to ensure that the user's sensitive data cannot be leaked from applications to unauthorized parties). These two goals are hardly new: Wang *et al.* articulated similar ones, and proposed new browser primitives to improve isolation within *mashups,* including discretionary access control (DAC) for inter-frame communication [206]. Indeed, today's browsers incorporate similar mechanisms in the guises of HTML5's iframe sandbox and postMessage API [214]. And the *Same-Origin Policy* (SOP, reviewed in Section 3.2.1) prevents JavaScript hosted by one principal from reading content hosted by another.

Unfortunately, in the status-quo web browser security architecture, one must often sacrifice privacy to achieve flexibility, and vice-versa. The central reason that flexibility and privacy are at odds in the status quo is that the mechanisms today's browsers rely on for providing privacy—the SOP, Content Security Policy (CSP) [209], and Cross-Origin Resource Sharing (CORS) [213]—are all forms of discretionary access control. DAC has the brittle character of either denying or granting untrusted code (*e.g.,* a library written by a third party) access to data. In the former case, the untrusted JavaScript might *need* the sensitive data to implement the desired application functionality—hence, denying access prioritizes privacy over flexibility. In the latter, DAC exercises no control over what

the untrusted code does with the sensitive data—and thus prioritizes flexibility over privacy. DAC is an essential tool in the privacy arsenal, but *does not fit cases where one runs untrusted code on sensitive input,* which are the norm for web applications, given their multi-contributor nature.

In practice, web developers turn their backs on privacy in favor of flexibility because the browser doesn't offer primitives that let them opt for both. For example, a developer may want to include untrusted JavaScript from another origin in his application. All-or-nothing DAC leads the developer to include the untrusted library with a `script` tag, which effectively bypasses the SOP, interpolating untrusted code into the enclosing page and granting it unfettered access to the enclosing page's origin's content.[2] And when a developer of a mashup that integrates content from *other* origins finds that the SOP forbids his application from retrieving data from them, he designs his mashup to require that the user provide the mashup her login credentials for the sites at the two other origins [133]—the epitome of "functionality over privacy."

In this chapter, we present COWL (Confinement with Origin Web Labels), a mandatory access control (MAC) system that confines untrusted JavaScript in web browsers. COWL allows untrusted code to compute over sensitive data and display results to the user, but prohibits the untrusted code from exfiltrating sensitive data (*e.g.,* by sending it to an untrusted remote origin). It thus allows web developers to opt for *both* flexibility and privacy.

We consider four motivating example web applications—a password strength-checker, an application that imports the (untrusted) jQuery library, an encrypted cloud-based document editor, and a third-party mashup, none of which can be implemented in a way that preserves the user's privacy in the status-quo web security architecture. These examples drive the design requirements for COWL, particularly MAC with *symmetric and hierarchical confinement* that supports *delegation.* Symmetric confinement allows *mutually* distrusting principals each to pass sensitive data to the other, and confine the other's use of the passed sensitive data. Hierarchical confinement allows any developer to confine code she does not trust, and confinement to be nested to arbitrary depths. And delegation allows a developer explicitly to confer the privileges of one execution context on a separate execution context. No prior browser security architecture offers this combination of properties.

We demonstrate COWL's applicability by implementing secure versions of the four motivating applications with it. Our contributions include:

- We characterize the shared needs of four case-study web applications (Section 3.2.2) for

[2] Indeed, jQuery *requires* such access to the enclosing page's content!

which today's browser security architecture cannot provide privacy.

- We describe the design of the COWL label-based MAC system for web browsers (Section 3.3), which meets the requirements of the four case-study web applications.
- We describe designs of the four case-study web applications atop COWL (Section 3.4).
- We describe implementations of COWL (Section 3.5) for the Firefox and Chromium open-source browsers; our evaluation (Section 3.6) illustrates that COWL incurs minimal performance overhead over the respective baseline browsers.

The contributions of this chapter led to the standardization of COWL at the W3C; we refer the interested reader to working draft for more details [181].

3.2 Background, Examples, & Goals

A single top-level web page often incorporates multiple scripts written by different authors.[3] Ideally, the browser should protect the user's sensitive data from unauthorized disclosure, yet afford page developers the greatest possible flexibility to construct featureful applications that reuse functionality implemented in scripts provided by (potentially untrusted) third parties. To make concrete the diversity of potential trust relationships between scripts' authors and the many ways page developers structure amalgamations of scripts, we describe several example web applications, none of which can be implemented with strong privacy for the user in today's web browsers. These examples illustrate key requirements for the design of a flexible browser confinement mechanism. Before describing these examples, however, we offer a brief refresher on status-quo browser privacy polices.

3.2.1 Browser Privacy Policies

Browsing contexts Figure 3.1 depicts the basic building blocks of the current web security architecture. A *browsing context* (*e.g.,* a page or frame) encapsulates presentable content and a JavaScript execution environment (heap and code) that interacts with content through the *Document Object Model (DOM)* [214]. Browsing contexts may be nested (*e.g.,* by using iframes). They also

[3]Throughout we use "web page" and "web application" interchangeably, and "JavaScript code" and "script" interchangeably.

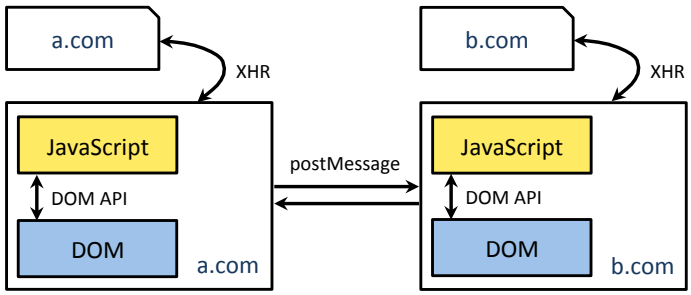

Figure 3.1: Simplified browser architecture.

may read and write persistent storage (*e.g.,* cookies), issue network requests (either implicitly in page content that references a URL retrieved over the network, or explicitly in JavaScript, using the `XMLHttpRequest` (XHR) constructor), and communicate with other contexts (IPC-style via `postMessage`, or, in certain cases, by sharing DOM objects). Some contexts such as Web Workers [211] run JavaScript but do not instantiate a DOM. We use the terms *context* and *compartment* interchangeably to refer to both browsing contexts and workers, except when the more precise meaning is relevant.

Origins and the Same-Origin Policy Since different authors may contribute components within a page, today's status quo browsers impose a security policy on interactions among components. Policies are expressed in terms of *origins*. An origin is a source of authority encoded by the protocol (*e.g.,* `https`), domain name (*e.g.,* `fb.com`), and port (*e.g.,* `443`) of a resource URL. For brevity, we elide the protocol and port from URLs throughout.

The same-origin policy specifies that an origin's resources should be readable only by content from the same origin [16, 232, 200]. Browsers ensure that code executing in an `a.com` context can only inspect the DOM and cookies of another context if they share the same origin, *i.e.,* `a.com`. Similarly, such code can only inspect the response to a network request (performed with XHR) if the remote host's origin is `a.com`.

The SOP does not, however, prevent code from *disclosing* data to foreign origins. For example, code executing in an `a.com` context can trivially disclose data to `b.com` by using XHR to perform a network request; the SOP prevents the code from inspecting responses to such cross-origin XHR requests, but does not impose any restrictions on sending such requests. Similarly, code can exfiltrate data by encoding it in the path of a URL whose origin is `b.com`, and setting the `src` property of an `img` element to this URL.

Content Security Policy (CSP) Modern browsers allow the developer to protect a user's privacy by specifying a CSP that limits the communication of a page—*i.e.,* that disallows certain communication ordinarily permitted by the SOP. Developers may set individual CSP directives to restrict the origins to which a context may issue requests of specific types (for images or scripts, XHR destinations, *etc.*) [209]. However, CSP policies suffer from two limitations. They are *static:* they cannot change during a page's lifetime (*e.g.,* a page may not drop the privilege to communicate with untrusted origins before reading potentially sensitive data). And they are *inaccessible:* JavaScript code cannot inspect the CSP of its enclosing context or some other context, *e.g.,* when determining whether to share sensitive data with that other context.

postMessage and Cross-Origin Resource Sharing (CORS) As illustrated in Figure 3.1, the HTML5 `postMessage` API [210] enables cross-origin communication in IPC-like fashion within the browser. To prevent unintended leaks [17], a sender always specifies the origin of the intended recipient; only a context with that origin may read the message.

CORS [213] goes a step further and allows controlled cross-origin communication between a browsing context of one origin and a remote server with a different origin. Under CORS, a server may include a header on returned content that explicitly whitelists other origin(s) allowed to read the response.

Note that both `postMessage`'s target origin and CORS are purely discretionary in nature: they allow static selection of which cross-origin communication is allowed and which denied, but enforce no confinement on a receiving compartment of differing origin. Thus, in the status-quo web security architecture, a privacy-conscious developer should only send sensitive data to a compartment of differing origin if she completely trusts that origin.

3.2.2 Motivating Examples

Having reviewed the building blocks of security policies in status-quo web browsers, we now turn to examples of web applications for which strong privacy is not achievable today. These examples illuminate key design requirements for the COWL confinement system.

Password Strength Checker Given users' propensity for choosing poor (*i.e.,* easily guessable) passwords, many web sites today incorporate functionality to check the strength of a password selected by a user and offer the user feedback (*e.g.,* "too weak; choose another," "strong," *etc.*).

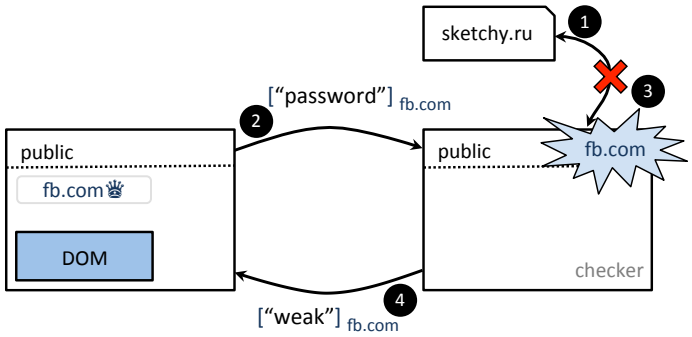

Figure 3.2: Third-party password checker architecture under COWL.

Suppose a developer at Facebook (origin `fb.com`) wishes to re-use password-checking functionality provided in a JavaScript library by a third party, say, from origin `sketchy.ru`. If the developer at `fb.com` simply includes the third party's code in a `script` tag referencing a resource at `sketchy.ru`, then the referenced script will have unfettered access to both the user's password (provided by the Facebook page, which the library *must* see to do its job) and to write to the network via XHR. This simple state of affairs is emblematic of the ease with which naïve web developers can introduce leaks of sensitive data in applications.

A more skilled web developer could today host the checker script on her *own* server and have that server specify a CSP policy for the page. Unfortunately, a CSP policy that disallows scripts within the page from initiating XHRs to any other origins is *too inflexible,* in that it precludes useful operations by the checker script, *e.g.,* retrieving an updated set of regular expressions describing weak passwords from a remote server (essentially, "updating" the checker's functionality). Doing so requires communicating with a remote origin. Yet a CSP policy that permits such communication, even with the top-level page's same origin, is *too permissive:* a malicious script could potentially carry out a *self-exfiltration attack* and write the password to a public part of the trusted server [221, 39].

This trade-off between flexibility and privacy, while inherent to CSP, need not be fundamental to the web model. The key insight is that it is entirely safe and useful for an untrusted script to communicate with remote origins *before* it reads sensitive data. We note, then, the requirement of a confinement mechanism that allows code in a compartment to communicate with the network *until it has been exposed to sensitive data.* MAC-based confinement meets this requirement.

Figure 3.2 shows how such a design might look. In this and subsequent examples, rectangular frames denote compartments, arrows denote communication (either between a compartment and the

network, or IPC-style between compartments), and events during execution are numbered sequentially in time. As we have proposed previously [217], compartments may be *labeled* (Section 3.3.1) with the origins to whose sensitive data they have been exposed. A compartment that has not yet observed sensitive data is denoted `public`; however, when it wishes to incorporate sensitive data, the compartment *raises* its label (at the cost of being more restricted in where it can write). We illustrate the raising of a label with a "flash" connoting the sensitivity of data being integrated. A compartment's *privilege* (Section 3.3.3), which specifies the origins for which a script executing in that compartment is trusted, is indicated by a crown. Here, a top-level page at `fb.com` encapsulates a password-checker script from a third-party origin in a new compartment. The label of the new compartment is initially `public`. First, in step (1), the checker script is free to download updated regular expressions from an arbitrary remote origin. In step (2), the top-level page sends the user's password to the checker script's worker using `postMessage`; the password is *labeled* `fb.com` to indicate that the data is sensitive to this origin (Section 3.3.2). In step (3) the checker raises its label to reflect that the context is about to be exposed to sensitive data from `fb.com` and inspects the password. When the label is raised, COWL atomically denies the context further access to the network in step (3).[4] However, the checker script is free to compute the result, which it then returns via `postMessage` to the top-level page in step (4); the result carries the label `fb.com` to reflect that the sender may be sending data derived from sensitive data owned by `fb.com`. Since the top-level page has the `fb.com` privilege, it can simply read the data (without raising its label).

Encrypted Document Editor Today's web applications, such as in-browser document editors backed by cloud-based storage (*e.g.,* Google Docs), typically require the user to trust the app developer/cloud-based storage provider (often the same principal under the SOP) with the data in her documents. That is, the provider's server observes the user's data in cleartext. Suppose an organization wished to use an in-browser document editor but did *not* want to reveal its users' document data to the editor provider's server. How might the provider offer a privacy-preserving editor app that would satisfy the needs of such a privacy-conscious organization? One promising approach might be for the "customer" privacy-sensitive organization to implement a trusted document encryption service hosted at its own origin, distinct from that which hosts the editor app. The editor app could allow the user to specify a JavaScript "plugin" library she trusts to perform cryptography correctly.

[4] For clarity, we use `fb.com` as the label on the data. This label still allows the checker to send XHR requests to `fb.com`; to ensure that the checker cannot communicate with *any* origin, COWL provides fresh origins (see Section 3.3.3).

In this design, one origin serves the JavaScript code for the editor app (say, `gdocs.com`) and a different origin serves the JavaScript code for the cryptography library (say, `eff.org`). Note that these two origins may be *mutually distrusting.* `gdocs.com`'s script must pass the document's cleartext to a script from `eff.org` for encryption, but would like to confine the execution of the encryption script so that it cannot exfiltrate the document to any origin *other* than `gdocs.com`. Similarly, `eff.org` 's cryptography library may not trust `gdocs.com` with the cleartext document—it would like to confine `gdocs.com`'s editor to prevent exfiltration of the cleartext document to `gdocs.com` (or to any other origin). This simple use case highlights the need for *symmetric confinement:* when two mutually distrusting scripts from different origins communicate, *each must be able to confine the other's further use of data it provides.*

Third-Party Mashup Some of the most useful web applications are *mashups*; these applications integrate and compute over data hosted by multiple origins. For example, consider an application that reconciles a user's Amazon purchases (the data for which are hosted by `amazon.com`) against a user's bank statement (the data for which are hosted by `chase.com`). The user may well deem both these categories of data sensitive and will furthermore not want data from Amazon to be exposed to her bank or vice-versa, nor to any other remote party. Today, if one of the two providers implements the mashup, its application code must bypass the SOP to allow sharing of data across origin boundaries, *e.g.,* by communicating between iframes with `postMessage` or setting a permissive CORS policy. This approach forfeits privacy: one origin sends sensitive data to the other, after which the receiving origin may exfiltrate that sensitive data at will. Alternatively, a third-party developer may wish to implement and offer this mashup application. Users of such a *third-party mashup* give up their privacy, usually by simply handing off credentials, as again today's browser enforces no policy that confines the sensitive data the mashup's code observes within the browser. To enable third-party mashups that do not sacrifice the user's privacy, we note again the need for an untrusted script to be able to issue requests to multiple remote origins (*e.g.,* `amazon.com` and `chase.com`), but to lose the privilege to communicate over the network once it has read the responses from those origins. Here, too, MAC-based confinement addresses the shortcomings of DAC.

Untrusted Third-Party Library Web application developers today make extensive use of third-party libraries like jQuery. Simply importing a library into a page provides no isolation whatsoever between the untrusted third-party code and any sensitive data within the page. Developers of applications that process sensitive data want the convenience of reusing popular libraries. But such reuse

risks exfiltration of sensitive data by these untrusted libraries. Note that because jQuery requires access to the content of the entire page that uses it, we cannot isolate jQuery in a separate compartment from the parent's, as we did for the password-checker example. Instead, we observe that jQuery demands a design that is a mirror image of that for confining the password checker: we place the *trusted* code for a page in a separate compartment and deem the rest of the page (including the untrusted jQuery code) as untrusted. The trusted code can then communicate with remote origins and inject sensitive data into the untrusted page, but the untrusted page (including jQuery) cannot communicate with remote origins (and thus cannot exfiltrate sensitive data within the untrusted page). This refactoring highlights the need for a confinement system that supports *delegation* and *dropping privilege*: a page should be able to create a compartment, confer its privileges to communicate with remote origins on that compartment, and then give these privileges up.

We note further that any *library* author may wish to reuse functionality from another untrusted library. Accordingly, to allow the broadest reuse of code, the browser should support *hierarchical confinement*—the primitives for confining untrusted code should allow not only a single level of confinement (one trusted context confining one untrusted context), but arbitrarily many levels of confinement (one trusted context confining an untrusted one, that in turn confines a further untrusted one, *etc.*).

3.2.3 Design Goals

We have briefly introduced four motivating web applications that achieve rich functionality by combining code from one or more untrusted parties. The privacy challenges that arise in such applications are unfortunately unaddressed by status-quo browser security policies, such as the SOP. These applications clearly illustrate the need for robust yet flexible confinement for untrusted code in browsers. To summarize, these applications would appear to be well served by a system that:

- Applies mandatory access control (MAC);
- Is *symmetric, i.e.,* it permits two principals to *mutually* distrust one another, and each prevent the other from exfiltrating its data;
- Is *hierarchical, i.e.,* it permits principal A to confine code from principal B that processes A's data, while principal B can independently confine code from principal C that processes B's data, *etc.*
- Supports *delegation* and *dropping privilege, i.e.,* it permits a script running in a compartment

with the privilege to communicate with some set of origins to confer those privileges on another compartment, then relinquish those privileges itself.

In the next section, we describe COWL, a new confinement system that satisfies these design goals.

3.3 The COWL Confinement System

The COWL confinement system extends the browser security model while leaving the browser fully compatible with today's "legacy" web applications.[5] Under COWL, the browser treats a page exactly like a legacy browser does unless the page executes a COWL API operation, at which point the browser records that page as running in *confinement mode,* and all further operations by that page are subject to confinement by COWL. COWL augments today's web browser with three primitives, all of which appear in the simple password-checker application example in Figure 3.2.

Labeled browsing contexts enforce MAC-based confinement of JavaScript at the granularity of a context (*e.g.,* a worker or iframe). The rectangular frames in Figure 3.2 are labeled contexts. As contexts may be nested, labeled browsing contexts allow hierarchical confinement, whose importance for supporting nesting of untrusted libraries we discussed in Section 3.2.2.

When one browsing context sends sensitive information to another, a sending context can use *labeled communication* to confine the potentially untrusted code receiving the information. This enables symmetric confinement, whose importance in building applications that compose mutually distrusting scripts we articulated in Section 3.2.2. In Figure 3.2, the arrows between compartments indicate labeled communication, where a subscript on the communicated data denotes the data's label.

COWL may grant a labeled browsing context one or more *privileges,* each with respect to an origin, and each of which reflects trust that the scripts executing within that context will not violate the secrecy and integrity of that origin's data, *e.g.,* because the browser retrieved them from that origin. A privilege authorizes scripts within a context to execute certain operations, such as *declassification* and *delegation*, whose abuse would permit the release of sensitive information to unauthorized parties. In COWL, we express privilege in terms of origins. The crown icon in the left compartment in Figure 3.2 denotes that this compartment may execute privileged operations on

[5] In prior work, we described how confinement can subsume today's browser security primitives, and advocated replacing them entirely with a clean-slate, confinement-based model [217]. In this chapter, we instead prioritize incremental deployability, which requires coexistence alongside the status quo model.

data labeled with the origin `fb.com`—more succinctly, that the compartment holds the privilege for `fb.com`. The compartment uses that privilege to remain unconfined by declassifying the checker response labeled `fb.com`.

We now describe these three constructs in greater detail.

3.3.1 Labeled Browsing Contexts

A COWL application consists of multiple labeled contexts. Labeled contexts extend today's browser contexts, used to isolate iframes, pages, *etc.*, with MAC *labels*. A context's label specifies the security policy for all data within the context, which COWL enforces by restricting the flow of information to and from other contexts and servers.

As described in Chapter 4, and in [217, 184], a label is a pair of boolean formulas over origins: a *secrecy* formula specifying which origins may read a context's data, and an *integrity* formula specifying which origins may write it. For example, only Amazon or Chase may read data labeled $\langle$`amazon.com` $\vee$ `chase.com`, `amazon.com`$\rangle$, and only Amazon may modify it.[6] Amazon could assign this label to its order history page to allow a Chase-hosted mashup to read the user's purchases. On the other hand, after a third-party mashup hosted by `mint.com` (as described in Section 3.2.2) reads *both* the user's Chase bank statement data *and* Amazon purchase data, the label on data produced by the third-party mashup will be $\langle$`amazon.com` $\wedge$ `chase.com`, `mint.com`$\rangle$. This secrecy label component specifies that the data may be sensitive to both parties, and without both their consent (see Section 3.3.3), it should only be read by the user; the integrity label component, on the other hand, permits only code hosted by Mint to modify the resulting data.

COWL enforces label policies in a MAC fashion by only allowing a context to communicate with other contexts or servers whose labels are at least as restricting. (A server's "label" is simply its origin.) Intuitively, when a context wishes to send a message, the target must not allow additional origins to read the data (preserving secrecy). Dually, the source context must not be writable by origins not otherwise trusted by the target. That is, the source must be at least as trustworthy as the target. We say that such a target label "subsumes" the source label. For example, a context labeled $\langle$`amazon.com`, `mint.com`$\rangle$ can send messages to one labeled $\langle$`amazon.com` $\wedge$ `chase.com`, `mint.com`$\rangle$, since the latter is trusted to preserve the privacy of `amazon.com` (and `chase.com`). However, communication in the reverse direction is not possible since it may violate the privacy

[6] $\vee$ and $\wedge$ denote disjunction and conjunction. A comma separates the secrecy and integrity formulas.

of `chase.com`. In the rest of this chapter, we limit our discussion to secrecy and only comment on integrity where relevant; we refer the interested reader to Chapter 4 for a full description of the label model.

A context can freely *raise* its label, *i.e.,* change its label to any label that is more restricting, in order to receive a message from an otherwise prohibited context. Of course, in raising its label to read more sensitive data from another context, the context also becomes more restricted in where it can write. For example, a Mint context labeled ⟨`amazon.com`⟩ can raise its label to ⟨`amazon.com` ∧ `chase.com`⟩ to read bank statements, but only at the cost of giving up its ability to communicate with Amazon (or, for that matter, any other) servers. When creating a new context, code can impose an upper bound on the context's label to ensure that untrusted code cannot raise its label and read data above this *clearance*. This notion of clearance is well established [63, 228, 186, 183, 86]; we discuss its relevance to covert channels in Section 3.7.

As noted, COWL allows a labeled context to create additional labeled contexts, much as today's browsing contexts can create sub-compartments in the form of iframes, workers, *etc.* This functionality is crucial for compartmentalizing a system hierarchically, where the developer places code of different degrees of trustworthiness in separate contexts. For example, in the password checker example in Section 3.2.2, we create a child context in which we execute the untrusted checker script. Importantly, however, code should not be able to leak information by laundering data through a newly created context. Hence, a newly created context implicitly inherits the current label of its parent. Alternatively, when creating a child, the parent may specify an initial current label for the child that is *more* restrictive than the parent's, to confine the child further. Top-level contexts (*i.e.,* pages) are assigned a default label of `public`, to ensure compatibility with pages written for the legacy SOP. Such browsing contexts can be restricted by setting a `COWL-label` HTTP response header, which dictates the minimal document label the browser must enforce on the associated content.

COWL applications can create two types of context. First, an application can create standard (but labeled) contexts in the form of pages, iframes, workers, *etc.* Indeed, it may do so because a COWL application is merely a regular web application that additionally uses the COWL API. It thus is confined by MAC, in addition to today's web security policies. Note that to enforce MAC, COWL must mediate all pre-existing communication channels—even subtle and implicit channels, such as content loading—according to contexts' labels. We describe how COWL does so in Section 3.5.

Second, a COWL application can create labeled contexts in the form of *lightweight labeled workers (LWorkers)*. Like normal workers [211], the API exposed to LWorkers is minimal; it consists only of constructs for communicating with the parent, the XHR constructor, and the COWL API.

Unlike normal workers, which execute in separate threads, an LWorker executes in the same thread as its parent, sharing its event loop. This sharing has the added benefit of allowing the parent to give the child (labeled) access to its DOM, any access to which is treated as both a read and a write, *i.e.,* bidirectional communication. Our third-party library example uses such a *DOM worker* to isolate the trusted application code, which requires access to the DOM, from the untrusted jQuery library. In general, LWorkers—especially when given DOM access—simplify the isolation and confinement of scripts (*e.g.,* the password strength checker) that would otherwise run in a shared context, as when loaded with `script` tags.

3.3.2 Labeled Communication

Since COWL enforces a label check whenever a context sends a message, the design described thus far is already symmetric: a source context can confine a target context by raising its label (or a child context's label) and thereafter send the desired message. To read this message, the target context must confine itself by raising its label accordingly. These semantics can make interactions between contexts cumbersome, however. For example, a sending context may wish to communicate with multiple contexts, and need to confine those target contexts with different labels, or even confine the same target context with different labels for different messages. And a receiving context may need unfettered communication with one or more origins for a time before confining itself by raising its label to receive a message. In the password-checker example application, the untrusted checker script at the right of Figure 3.2 exhibits exactly this latter behavior: it needs to communicate with untrusted remote origin `sketchy.ru` before reading the password labeled `fb.com`.

Labeled Blob Messages (Intra-Browser) To simplify communication with confinement, we introduce the *labeled Blob,* which binds together the payload of an individual inter-context message with the label protecting it. The payload takes the form of a serialized immutable object of type Blob [214]. Encapsulating the label with the message avoids the cumbersome label raises heretofore necessary in both sending and receiving contexts before a message may even be sent or received. Instead, COWL allows the developer sending a message from a context to specify the label to be attached to a labeled Blob; any label as or more restrictive than the sending context's current label may be specified (modulo its clearance). While the receiving context may receive a labeled Blob with no immediate effect on the origins with which it can communicate, it may only inspect the

label, not the payload.[7] Only after raising its label as needed may the receiving context read the payload.

Labeled Blobs simplify building applications that incorporate distrust among contexts. Not only can a sender impose confinement on a receiver simply by labeling a message; a receiver can delay inspecting a sensitive message until it has completed communication with untrusted origins (as does the checker script in Figure 3.2). They also ease the implementation of integrity in applications, as they allow a context that is not trusted to modify content in some other context to serve as a passive conduit for a message from a third context that *is* so trusted.

Labeled XHR Messages (Browser–Server) Thus far we have focused on confinement as it arises when two browser contexts communicate. Confinement is of use in browser-server communication, too. As noted in Section 3.3.1, COWL only allows a context to communicate with a server (whether with XHR, retrieving an image, or otherwise) when the server's origin subsumes the context's label. Upon receiving a request, a COWL-aware web server may also wish to know the current label of the context that initiated it. For this reason, COWL attaches the current label to every request the browser sends to a server.[8] As also noted in Section 3.3.1, a COWL-aware web server may elect to label a response it sends the client by including a `COWL-label` header on it. In such cases, the COWL-aware browser will only allow the receiving context to read the XHR response if its current label subsumes that on the response.

Here, again, a context that receives labeled data—in this case from a server—may wish to defer raising its label until it has completed communication with other remote origins. To give a context this freedom, COWL supports *labeled XHR* communication. When a script invokes COWL's labeled XHR constructor, COWL delivers the response to the initiating script as a labeled Blob. Just as with labeled Blob intra-browser IPC, the script is then free to delay raising its label to read the payload of the response—and delay being confined—until after it has completed its other remote communication. For example, in the third-party mashup example, Mint only confines itself once it has received all necessary (labeled) responses from both Amazon and Chase. At this point it processes the data and displays results to the user, but it can no longer send requests since doing so may leak information.[9]

[7] The label itself cannot leak information—COWL still ensures that the target context's label is at least as restricting as that of the source.

[8] COWL also attaches the current privilege; see Section 3.3.3.

[9] To continuously process data in "streaming" fashion, one may partition the application into contexts that poll Amazon

3.3.3 Privileges

While confinement handily enforces secrecy, there are occasions when an application must eschew confinement in order to achieve its goals, and yet can uphold secrecy while doing so. For example, a context may be confined with respect to some origin (say, `a.com`) as a result of having received data from that origin, but may need to send an encrypted version of that data to a third-party origin. Doing so does not disclose sensitive data, but COWL would normally prohibit such an operation. In such situations, how can a context *declassify* data, and thus be permitted to send to an arbitrary recipient, or avoid the recipient's being confined?

COWL's *privilege* primitive enables safe declassification. A context may hold one or more privileges, each with respect to some origin. Possession of a privilege for an origin by a context denotes trust that the scripts that execute within that context will not compromise the secrecy of data from that origin. Where might such trust come from (and hence how are privileges granted)? Under the SOP, when a browser retrieves a page from `a.com`, any script within the context for the page is trusted not to violate the secrecy of `a.com`'s data, as these scripts are deemed to be executing on behalf of `a.com`. COWL makes the analogous assumption by granting the privilege for `a.com` to the context that retrieves a page from `a.com`: scripts executing in that context are similarly deemed to be executing on behalf of `a.com`, and thus are trusted not to leak `a.com`'s data to unauthorized parties—even though they can declassify data. Only the COWL runtime can create a new privilege for a valid remote origin upon retrieval of a page from that origin; a script cannot synthesize a privilege for a valid remote origin.

To illustrate the role of privileges in declassification, consider the encrypted Google Docs example application. In the implementation of this application atop COWL, code executing on behalf of `eff.org` (*i.e.,* in a compartment holding the `eff.org` privilege) with a current label $\langle$`eff.org` $\wedge$ `gdoc.com`$\rangle$ is permitted to send messages to a context labeled $\langle$`gdoc.com`$\rangle$. Without the `eff.org` privilege, this flow would not be allowed, as it may leak the EFF's information to Google.

Similarly, code can declassify information when unlabeling messages. Consider now the password checker example application. The left context in Figure 3.2 leverages its `fb.com` privilege to declassify the password strength result, which is labeled with its origin, to avoid (unecessarily) raising its label to `fb.com`.

COWL generally exercises privileges *implicitly:* if a context holds a privilege, code executing

and Chase's servers for new data and pass labeled responses to the confined context that processes the payloads of the responses.

in that context will, with the exception of sending a message, always attempt to use it.[10] COWL, however, lets code control the use of privileges by allowing code to get and set the underlying context's privileges. Code can drop privileges by setting its context's privileges to `null`. Dropping privileges is of practical use in confining closely coupled untrusted libraries like jQuery. Setting privileges, on the other hand, increases the trust placed in a context by authorizing it act on behalf of origins. This is especially useful since COWL allows one context to *delegate* its privileges (or a subset of them) to another; this functionality is also instrumental in confining untrusted libraries like jQuery. Finally, COWL also allows a context to create privileges for *fresh* origins, *i.e.,* unique origins that do not have a real protocol (and thus do not map to real servers). These fresh origins are primarily used to *completely* confine a context: the sender can label messages with such an origin, which upon inspection will raise the receiver's label to this "fake" origin, thereby ensuring that it cannot communicate except with the parent (which holds the fresh origin's privilege).

3.4 Applications

In Section 3.2.2, we characterized four applications and explained why the status-quo web architecture cannot accommodate them satisfactorily. We then described the COWL system's new browser primitives. We now close the loop by demonstrating how to build the aforementioned applications with the COWL primitives.

Encrypted Document Editor The key feature needed by an encrypted document editor is symmetric confinement, where two mutually distrusting scripts can each confine the other's use of data they send one another. Asymmetrically conferring COWL privileges on the distrusting components is the key to realizing this application.

Figure 3.3 depicts the architecture for an encrypted document editor. The editor has three components: a component which has the user's Google Docs credentials and communicates with the server (`gdoc.com`), the editor proper (also `gdoc.com`), and the component that performs encryption (`eff.org`). COWL provides privacy as follows: if `eff.org` is honest, then COWL ensures that the cleartext of the user's document is not leaked to any origin. If only `gdoc.com` is honest, then `gdoc.com` may be able to recover cleartext (*e.g.,* the encryptor may have used the null "cipher"),

[10] While the alternative approach of explicit exercise of privileges (*e.g.,* when registering an `onmessage` handler) may be safer [228, 186, 132], we find it a poor fit with existing asynchronous web APIs.

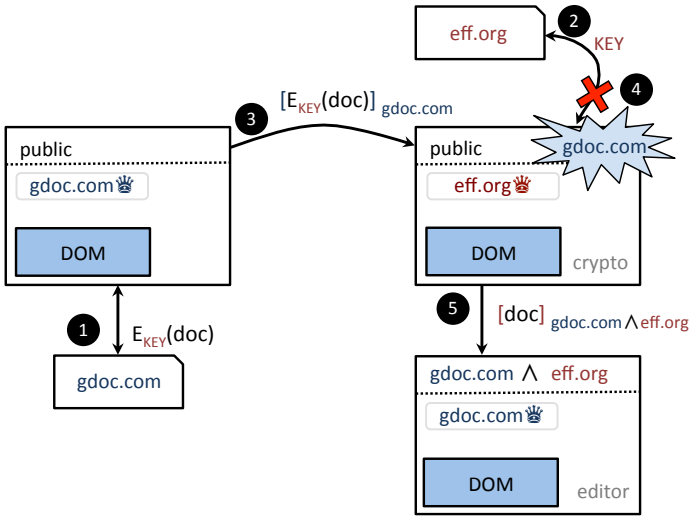

Figure 3.3: Encrypted document editor architecture under COWL.

but the encryptor should not be able to exfiltrate the cleartext to anyone else.

How does execution of the encrypted document editor proceed? Initially, `gdoc.com` downloads (1) the encrypted document from Google's servers. As the document is encrypted, it opens an iframe to `eff.org`, with initial label `public` so it can communicate with the `eff.org` server and download the private key (2) which will be used to decrypt the document. Next, it sends the encrypted document as a labeled Blob, with the label $\langle$`gdoc.com`$\rangle$ (3); the iframe unlabels the Blob and raises its label (4) so it can decrypt the document. Finally, the iframe passes the decrypted document (labeled as $\langle$`gdoc.com` $\wedge$ `eff.org`$\rangle$) to the iframe (5) implementing the editor proper.

To save the document, these steps proceed in reverse: the editor sends a decrypted document to the encryptor (5), which encrypts it with the private key. Next, the critical step occurs: the encryptor exercises its privileges to send a labeled blob of the encrypted document which is *only* labeled $\langle$`gdoc.com`$\rangle$ (3). Since the encryptor is the only compartment with the `eff.org` privilege, all documents must pass through it for encryption before being sent elsewhere; conversely, it itself cannot exfiltrate any data, as it is confined by `gdoc.com` in its label.

We have implemented a password manager atop COWL that lets users safely store passwords on third-party web-accessible storage. We elide its detailed design in the interest of brevity, and note only that it operates similarly to the encrypted document editor.

Third-Party Mashup Labeled XHR as composed with CORS is central to COWL's support for third-party mashups. Today's CORS policies are DAC-only, such that a server must either allow

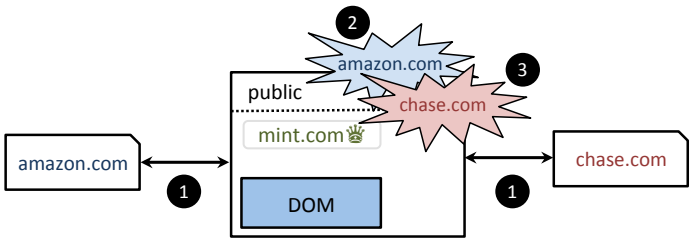

Figure 3.4: Third-party mashup under COWL.

another origin to read its data and fully trust that origin not to disclose the data, or deny the other origin access to the data altogether. Under COWL, however, a server could CORS-whitelist a foreign origin to permit that origin to read its data, and by setting a label on its response, be safe in the knowledge that COWL would appropriately confine the foreign origin's scripts in the browser.

Figure 3.4 depicts an application that reconciles a user's Amazon purchases and bank statement. Here, Chase and Amazon respectively expose authenticated read-only APIs for bank statements and purchase histories that whitelist known applications' origins, such as `mint.com`, but set MAC labels on responses.[11] As discussed in Section 3.7, with MAC in place, COWL allows users to otherwise augment CORS by whitelisting foreign origins on a per-origin basis. The mashup makes requests to both web sites using labeled XHR (1) to receive the bank statement and purchase history as labeled Blobs. Once all of the information is received, the mashup unlabels the responses and raises its context's label accordingly (2–3); doing so restricts communication to the web at large.

Note that in contrast to when solely using CORS, by setting MAC labels on responses, Chase and Amazon need not trust Mint to write bug-free code—COWL confines the Mint code to ensure that it cannot arbitrarily leak sensitive data. As we discuss in Section 3.7, however, a malicious Mint application could potentially leak data through covert channels. We emphasize that COWL nevertheless offers a significant improvement over the status quo, in which, *e.g.,* users give their login credentials to Mint, and thus not only trust Mint to keep their bank statements confidential, but also not to steal their funds!

Untrusted Third-Party Library COWL can confine tightly coupled untrusted third-party libraries like jQuery by delegating privileges to a trusted context and subsequently dropping them from the main page. In doing so, COWL completely confines the main page, and ensures that it can

[11] On authentication: note that when the browser sends any XHR (labeled or not) from a foreign origin to origin `chase.com`, it still includes any cookies cached for `chase.com` in the request.

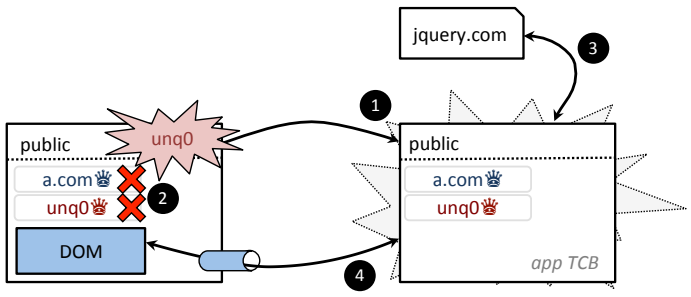

Figure 3.5: Privilege separation and library confinement.

only communicate with the trusted and unconfined context. Here, the main page may start out with sensitive data in context, or alternatively, receive it from the trusted compartment.

Figure 3.5 shows how to use COWL to confine the untrusted jQuery library referenced by a web page. The goal is to establish a separate DOM worker with the `a.com` privilege, while the main browsing context runs jQuery in confined fashion—without privileges or the ability to talk to the network. Initially the main browsing context holds the `a.com` privilege. The page generates a fresh origin `unq0` and spawns a DOM worker (1), delegating it both privileges. The main context then drops its privileges and raises its label to $\langle$`unq0`$\rangle$ (2). Finally, the trusted worker downloads jQuery (3) and injects the script content into the main context's DOM (4). When the library is loaded, the main context becomes untrusted, but also fully confined. As the trusted DOM worker holds both privileges, it can freely modify the DOM of the main context, as well as communicate with the wider web. One may view this DOM worker as a *firewall* between the page proper (with the untrusted library) and the rest of the world.

3.5 Implementation

We implemented COWL in Firefox 31.0a1 and Chromium 31.0.1612.0. Because COWL operates at a context granularity, it admits an implementation as a new DOM-level API for the Gecko and Blink layout engines, without any changes to the browsers' JavaScript engines. Figure 3.6 shows the core parts of this API. We focus on the Firefox implementation and only describe the Chromium one where the two diverge non-trivially.

```
interface Label :
  Label Label(String)
  Label and(String or Label)
  Label or(String or Label)
  bool subsumes(Label [,Privilege])
interface Privilege :
  Privilege FreshPrivilege()
  Privilege combine(Privilege)
  readonly attribute Label asLabel
```

(a) Labels and privileges.

```
interface LabeledBlob :
  readonly attribute Label label
  readonly attribute Blob blob
```

(b) Labeled Blobs.

```
interface COWL :
  static void enable()
  static attribute Label label
  static attribute Label clearance
  static attribute Privilege privilege
interface LWorker :
  LWorker LWorker(String, Label
                  [, Privilege, object])
  postMessage(object)
  attribute EventHandler onmessage
```

(c) Labeled compartments.

Figure 3.6: COWL programming interface in simplified WebIDL.

3.5.1 Labeled Browsing Contexts

Gecko's existing isolation model relies on JavaScript compartments, *i.e.,* disjoint JavaScript heaps, both for efficient garbage collection and security isolation [205]. To achieve isolation, Gecko performs all cross-compartment communication (*e.g.,* `postMessage` between iframes) through *wrappers* that implement the object-capability *membrane* pattern [131, 129]; membranes enable sound reasoning about "border crossing" between compartments. Wrappers ensure that an object in one compartment can never directly reference another object in a different compartment. Wrappers also include a security policy, which enforces all inter-compartment access control checks specified by the SOP. Security decisions are made with respect to a compartment's security principal, which contains the origin and CSP of the compartment.

Channel	Mechanism
`postMessage`	Cross-compartment wrappers
DOM window properties	Cross-compartment wrappers
Content loading	CSP
XHR	CSP + DOM interposition
Browser storage	SOP + CSP (sandbox)
Other (*e.g.,* iframe height)	DOM interposition

Table 3.1: Confining code from exfiltrating data using existing browser mechanisms. Since the Chromium architecture does not have cross-compartment wrappers, we modify the DOM binding code to insert label checks.

Since COWL's security model is very similar to this existing model, we can leverage these wrappers to introduce COWL's new security policies. We associate a label, clearance, and privilege with each compartment alongside the security principal. Wrappers consider all of these properties together when making security decisions.

Intra-Browser Confinement As shown in Table 3.1, we rely on wrappers to confine communication across compartments. Once confinement mode is enabled, we "recompute" all the cross-compartment wrappers to use our MAC wrapper policy and thereby ensure that all subsequent cross-compartment access is mediated not only by the SOP, but also by confinement. For `postMessage`, our policy ensures that the receiver's label subsumes that of the sender (taking the receiver's privileges into consideration); otherwise the message is silently dropped. For a cross-compartment DOM property access, we additionally check that the sender's label subsumes that of the receiver—*i.e.,* that the labels of the compartments are equivalent after considering the sender's privileges (in addition to the same-origin check performed by the SOP).

Blink's execution contexts (the dual to Gecko's compartments) do not rely on wrappers to enforce cross-context access control. Instead, Blink implements the SOP security checks in the DOM binding code for a limited subset of DOM elements that may allow cross-origin access. Since COWL policies are more fine-grained, we modified the binding code to extend the security checks to all DOM objects and also perform label checks when confinement mode is enabled. Unfortunately, without wrappers, shared references cannot efficiently be revoked (*i.e.,* without walking the heap). Hence, before enabling confinement mode, a page can create a same-origin iframe with which it shares references, and the iframe can thereafter leak any data from the parent even if the latter's label is raised. To prevent this eventuality, our current Chromium API allows senders to disallow

unlabeling Blobs if the target created any children before entering confinement mode.

Our implementations of LWorkers, whose API appears in Figure 3.6(c), reuse labeled contexts straightforwardly. In fact, the `LWorker` constructor simply creates a new compartment with a fresh origin that contains a fresh JavaScript global object to which we attach the XHR constructor, COWL API, and primitives for communicating with the parent (*e.g.,* `postMessage`). Since LWorkers may have access to their parents' DOM, however, our wrappers distinguish them from other contexts to bypass SOP checks and only restrict DOM access according to MAC. This implementation is very similar to the content scripts used by Chrome and Firefox extensions [37, 141].

Browser-Server Confinement As shown in Table 3.1, we confine external communication (including XHR, content loading, and navigation) using CSP. While CSP alone is insufficient for providing flexible confinement,[12] it sufficiently addresses our external communication concern by precisely controlling from where a page loads content, performs XHR requests to, *etc.* To this end, we set a custom CSP policy whenever the compartment label changes, *e.g.,* with `COWL.label`. For instance, if the compartment label is `Label("https://amazon.com").and("https://bank.ch")`, all the underlying CSP directives are set to `'none'`, disallowing all network communication. We also disable navigation with the `'sandbox'` directive [212, 215, 214].

Browser Storage Confinement As shown in Table 3.1, we use the `sandbox` directive to restrict access to storage (*e.g.,* cookies and HTML5 local storage [214]), as have other systems [5]. We leave the implementation of labeled storage as future work.

3.6 Evaluation

Performance largely determines acceptance of new browser features in practice. We evaluate the performance of COWL by measuring the cost of our new primitives as well as their impact on legacy web sites that do not use COWL's features. Our experiments consist of micro-benchmarks of API functions and end-to-end benchmarks of our example applications. We conducted all measurements on a 4-core i7-2620M machine with 16GB of RAM running GNU/Linux 3.13. The browser retrieved applications from the Node.js web server over the loopback interface. We note that these

[12] There are two primary reasons. First, JavaScript code cannot (yet) modify a page's CSP. And, second, CSP does not (yet) provide a directive for restricting in-browser communication, *e.g.,* with `postMessage`.

	Firefox			Chromium		
	vanilla	unlabeled	labeled	vanilla	unlabeled	labeled
New iframe	14.4	14.5	14.4	50.6	48.7	51.8
New worker	15.9	15.4	0.9†	18.9	18.9	3.3†
Iframe comm.	0.11	0.11	0.12	0.04	0.04	0.04
XHR comm	3.5	3.6	3.7	7.0	7.4	7.2
Worker comm.	0.20	0.24	0.03‡	0.07	0.07	0.03‡

Table 3.2: Micro-benchmarks, in milliseconds.

measurements are harsh for COWL, in that they omit network latency and the complex intra-context computation and DOM rendering of real-world applications, all of which would mask COWL's overhead further. Our key findings include:

- COWL's latency impact on legacy sites is negligible.
- Confining code with LWorkers is inexpensive, especially when compared to iframes/Workers. Indeed, the performance of our end-to-end confined password checker is only 5 ms slower than that of an inlined `script` version.
- COWL's incurs low overhead when enforcing confinement on mashups. The greatest overhead observed is 16% (for the encrypted document editor). Again, the absolute slowdown of 16 ms is imperceptible by users.

3.6.1 Micro-Benchmarks

Context Creation Table 3.2 shows micro-benchmarks for the stock browsers (vanilla), the COWL browsers with confinement mode turned off (unlabeled), and with confinement mode enabled (labeled). COWL adds negligible latency to compartment creation; indeed, except for LWorkers (†), the differences in creation times are of the order of measurement variability. We omit measurements of labeled "normal" Workers since they do not differ from those of unlabeled Workers. We attribute COWL's iframe-creation speedup in Chromium to measurement variability. We note that the cost of creating LWorkers is considerably less than that for "normal" Workers, which run in separate OS threads (†).

Communication The iframe, worker, and XHR communication measurements evaluate the round-trip latencies across iframes, workers, and the network. For the XHR benchmark, we report the cost of using the labeled XHR constructor averaged over 10,000 requests. Our Chromium implementation uses an LWorker to wrap the unmodified XHR constructor, so the cost of labeled XHR incorporates an additional cross-context call. As with creation, communicating with LWorkers (‡) is considerably faster than with "normal" Workers. This speedup arises because a lightweight LWorker shares an OS thread and event loop with their parent.

Labels We measured the cost of setting/getting the current label and the average cost of a label check in Firefox. For a randomly generated label with a handful of origins, these operations take on the order of one microsecond. The primary cost is recomputing cross-compartment wrappers and the underlying CSP policy, which ends up costing up to 13ms (*e.g.,* when the label is raised from public to a third-party origin). For many real applications, we expect raising the current label to be a rare occurrence. Moreover, there is much room for optimization (*e.g.,* porting COWL to the newest CSP implementation, which sets policies 15× faster [98]).

DOM We also executed the Dromaeo benchmark suite [161], which evaluates the performance of core functionality such as querying, traversing, and manipulating the DOM, in Firefox and Chromium. We found the performance of the vanilla and unlabeled browsers to be on par: the greatest slowdown was under 4%.

3.6.2 End-to-End Benchmarks

To focus on measuring COWL's overhead, we compare our apps against similarly compartmentalized but non-secure apps—*i.e.,* apps that perform no security checks.

Password-Strength Checker We measure the average duration of creating a new LWorker, fetching an 8 KB checker script based on [135], and checking a password sixteen characters in length. The checker takes an average of 18 ms (averaged over ten runs) on Firefox (labeled), 4 ms less than using a Worker on vanilla Firefox. Similarly, the checker running on labeled Chromium is 5 ms faster than the vanilla counterpart (measured at 54 ms). In both cases COWL achieves a speedup because its LWorkers are cheaper than normal Workers. However, these measurements are roughly 5 ms slower than simply loading the checker using an unsafe `script` tag.

Encrypted Document Editor We measure the end-to-end time taken to load the application and encrypt a 4 KB document using the SJCL AES-128 library [179]. The total run time includes the time taken to load the document editor page, which in turn loads the encryption-layer iframe, which further loads the editor proper. On Firefox (labeled) the workload completes in 116 ms; on vanilla Firefox, a simplified and unconfined version completes in 100ms. On Chromium, the performance measurements were comparable; the completion time was within 1ms of 244ms. The most expensive operation in the COWL-enabled Firefox app is raising the current label, since it requires changing the underlying document origin and recomputing the cross-compartment wrappers and CSP.

Third-Party Mashup We implemented a very simple third-party mashup application that makes a labeled XHR request to two unaffiliated origins, each of which produces a response containing a 27-byte JSON object with a numerical property, and sums the responses together. The corresponding vanilla app is identical, but uses the normal XHR object. In both cases we use CORS to permit cross-origin access. The Firefox (labeled) workload completes in 41 ms, which is 6 ms slower than the vanilla version. As in the document editor the slowdown derives from raising the current label, though in this case only for a single iframe. On Chromium (labeled) the workload completes in 55 ms, 2 ms slower than the vanilla one; the main slowdown here derives from our implementing labeled XHR with a wrapping LWorker.

Untrusted Third-Party Library We measured the load time of a banking application that incorporates jQuery and a library that traverses the DOM to replace phone numbers with links. The latter library uses XHR in attempt to leak the page's content. We compartmentalize the main page into a public outer component and a sensitive iframe containing the bank statement. In both compartments, we place the bank's trusted code (which loads the libraries) in a trusted labeled DOM worker with access to the page's DOM. We treat the rest of the code as untrusted. As our current Chromium implementation does not yet support DOM access for LWorkers, we only report measurements for Firefox. The measured latency on Firefox (labeled) is 165 ms, a 5 ms slowdown when compared to the unconfined version running on vanilla Firefox. Again, COWL prevents sensitive content from being exfiltrated and incurs negligible slowdown.

3.7 Discussion and Limitations

We now discuss the implications of certain facets of COWL's design, and limitations of the system.

User-Configured Confinement Recall that in the status-quo web security architecture, to allow cross-origin sharing, a server must grant individual foreign origins access to its data with CORS in an all-or-nothing, DAC fashion. COWL improves this state of affairs by allowing a COWL-aware server to more finely restrict how its shared data is disseminated—*i.e.,* when the server grants a foreign origin access to its data, it can confine the foreign origin's script(s) by setting a label on responses it sends the client.

Unfortunately, absent a permissive CORS header that whitelists the origins of applications that a user wishes to use, the SOP prohibits foreign origins from reading responses from the server, even in a COWL-enabled browser. Since a server's operator may not be aware of all applications its users may wish to use, the result is usually the same status-quo unpalatable choice between functionality and privacy—*e.g.,* give one's bank login credentials to Mint, or one cannot use the Mint application. For this reason, our COWL implementation lets browser users augment CORS by configuring for an origin (*e.g.,* `chase.com`) any foreign origins (*e.g.,* `mint.com`, `benjamins.biz`) they wish to additionally whitelist. In turn, COWL will confine these client-whitelisted origins (*e.g.,* `mint.com`) by labeling every response from the configured origin (`chase.com`). COWL obeys the server-supplied label when available and server whitelisting is *not* provided. Otherwise, COWL conservatively labels the response with a *fresh* origin (as described in Section 3.3.3). The latter ensures that once the response has been inspected, the code cannot communicate with *any* server, including at the *same* origin, since such requests carry the risks of self-exfiltration [39] and cross-site request forgery [202].

Covert Channels In an ideal confinement system, it would always be safe to let untrusted code compute on sensitive data. Unfortunately, real-world systems such as browsers typically exhibit *covert* channels that malicious code may exploit to exfiltrate sensitive data. Since COWL extends existing browsers, we do not protect against covert channel attacks. Indeed, malicious code can leverage covert channels already present in today's browsers to leak sensitive information. For instance, a malicious script within a confined context may be able to modulate sensitive data by varying rendering durations. A less confined context may then in turn exfiltrate the data to a remote host [101]. It is important to note, however, that COWL does not introduce new covert channels—our implementations re-purpose existing (software-based) browser isolation mechanisms (V8 contexts and SpiderMonkey compartments) to enforce MAC policies. Moreover, these MAC policies are generally more restricting than existing browser policies: they prevent unauthorized data exfiltration through *overt* channels and, in effect, force malicious code to resort to using covert channels.

The only fashion in which COWL relaxes status-quo browser policies is by allowing users to override CORS to permit cross-origin (labeled) sharing. Does this functionality introduce new risks? Whitelisting is user controlled (*e.g.,* the user must explicitly allow `mint.com` to read `amazon.com` and `chase.com` data), and code reading cross-origin data is subject to MAC (*e.g.,* `mint.com` cannot arbitrarily exfiltrate the `amazon.com` or `chase.com` data after reading it). In contrast, today's mashups like `mint.com` ask users for their passwords. COWL is strictly an improvement: under COWL, when a user decides to trust a mashup integrator such as `mint.com`, she *only* trusts the app to not leak her data through covert channels. Nevertheless, users can make poor security choices. Whitelisting malicious origins would be no exception; we recognize this as a limitation of COWL that must be communicated to the end-user.

A trustworthy developer can leverage COWL's support for *clearance* when compartmentalizing his application to ensure that only code that actually relies on cross-origin data has access to it. Clearance is a label that serves as an upper bound on a context's current label. Since COWL ensures that the current label is adjusted according to the sensitivity of the data being read, code cannot read (and thus leak) data labeled above the clearance. Thus, Mint can assign a "low" clearance to untrusted third-party libraries, *e.g.,* to keep `chase.com`'s data confidential. These libraries will then not be able to leak such data through covert channels, even if they are malicious.

Expressivity of Label Model COWL uses DC labels [184] to enforce confinement according to an information flow control discipline. Although this approach captures a wide set of confinement policies, it is not expressive enough to handle policies with a circular flow of information [15] or some policies expressible in more powerful logics (*e.g.,* first order logic, as used by Nexus [176]). DC labels are, however, as expressive as other popular label models [136], including Myers and Liskov's Decentralized Label Model [144]. Our experience implementing security policies with them thus far suggests they are expressive enough to support featureful web applications.

We adopted DC labels largely because their fit with web origins pays practical dividends. First, as developers already typically express policies by whitelisting origins, we believe they will find DC labels intuitive to use. Second, because both DC labels and today's web policies are defined in terms of origins, the implementation of COWL can straightforwardly reuse the implementation of existing security mechanisms, such as CSP.

3.8 Related Work on Browser-side Web Application Security

Existing browser confinement systems based on information flow control can be classified either as *fine-grained* or *coarse-grained*. The former associate IFC policies with individual objects, while the latter associate policies with entire browsing contexts. We compare COWL to previously proposed systems in both categories, then contrast the two categories' overall characteristics.

Coarse-grained IFC COWL shares many features with existing coarse-grained systems. For example, BFlow [221] allows web sites to enforce confinement policies stricter than the SOP via *protection zones*—groups of iframes sharing a common label. However, BFlow cannot mediate between mutually distrustful principals—*e.g.,* the encrypted document editor is not directly implementable with BFlow. This is because only asymmetric confinement is supported—a sub-frame cannot impose any restrictions on its parent. For the same reasons, BFlow cannot support applications that require security policies more flexible than the SOP, such as our third-party mashup example. These differences reflect different goals for the two systems. BFlow's authors set out to confine untrusted third-party scripts, while we also seek to support applications that incorporate code from mutually distrusting parties.

More recently, Akhawe *et al.* propose the data-confined sandbox (DCS) system [5], which allows pages to intercept and monitor the network, storage, and cross-origin channels of `data:` URI iframes. The limitation to `data:` URI iframes means DCS cannot confine the common case of a service provided in an iframe [178]. Like BFlow, DCS does not offer symmetric confinement, and does not incorporate functionality to let developers build applications like third-party mashups.

Fine-grained IFC Per-object-granularity IFC makes it easier to confine untrusted libraries that are closely coupled with trusted code on a page (*e.g.,* jQuery) and avoid the problem of *over-tainting,* where a single context accumulates taint as it inspects more data.

JSFlow [78] is one such fine-grained JavaScript IFC system, which enforces policies by executing JavaScript in an interpreter written in JavaScript. This approach incurs a two order of magnitude slowdown. JSFlow's authors suggest that this cost makes JSFlow a better fit for use as a development tool than as an "always-on" privacy system for users' browsers. Additionally, JSFlow does not support applications that rely on policies more flexible than the SOP, such as our third-party mashup example.

The FlowFox fine-grained system [50] enforces policies with secure-multi execution (SME) [55].

SME ensures that no leaks from a sensitive context can leak into a less sensitive context by executing a program multiple times. Unlike JSFlow and COWL, SME is not amenable to scenarios where declassification plays a key role (*e.g.,* the encrypted editor or the password manager). FlowFox's labeling of user interactions and metadata (history, screen size, *etc.*) do allow it to mitigate history sniffing and behavior tracking; COWL does not address these attacks.

While fine-grained IFC systems may be more convenient for developers, they impose new language semantics for developers to learn, require invasive modifications to the JavaScript engine, and incur greater performance overhead. In contrast, because COWL repurposes familiar isolation constructs and does not require JavaScript engine modifications, it is relatively straightforward to add to legacy browsers. It also only adds overhead to cross-compartment operations, rather than to all JavaScript execution. The typically short lifetime of a browsing context helps avoid excessive accumulation of taint. We conjecture that coarse-grained and fine-grained IFC are equally expressive, provided one may use arbitrarily many compartments—a cost in programmer convenience. Finally, coarse- and fine-grained mechanisms are not mutually exclusive. For instance, to confine legacy (non-compartmentalized) JavaScript code, one could deploy JSFlow within a COWL context.

Sandboxing The literature on sandboxing and secure subsets of JavaScript is rich, and includes Caja [75], BrowserShield [160], WebJail [199], TreeHouse [90], JSand [4], SafeScript [196], Defensive JavaScript [21], and Embassies [85]). While our design has been inspired by some of these systems (*e.g.,* TreeHouse), the usual goals of these systems are to mediate security-critical operations, restrict access to the DOM, and restrict communication APIs. In contrast to the mandatory nature of confinement, however, these systems impose most restrictions in discretionary fashion, and are thus not suitable for building some of the applications we consider (in particular, the encrypted editor). Nevertheless, we believe that access control and language subsets are crucial complements to confinement for building robustly secure applications.

3.9 Conclusion

Web applications routinely pull together JavaScript contributed by parties untrusted by the user, as well as by mutually distrusting parties. The lack of confinement for untrusted code in the status-quo browser security architecture puts users' privacy at risk. In this chapter, we have presented COWL, a label-based MAC system for web browsers that preserves users' privacy in the common case

where untrusted code computes over sensitive data. COWL affords developers flexibility in synthesizing web applications out of untrusted code and services while preserving users' privacy. Our positive experience building four web applications atop COWL for which privacy had previously been unattainable in status-quo web browsers suggests that COWL holds promise as a practical platform for preserving privacy in today's pastiche-like web applications. And our measurements of COWL's performance overhead in the Firefox and Chromium browsers suggest that COWL's privacy benefits come at negligible end-to-end cost in performance. COWL is a new W3C specification and on track to be standardized and deployed in major browsers by default [181].

Part II

Practical and Flexible Dynamic Language-Level IFC Foundations

Chapter 4

Disjunction Category Labels[1]

We present disjunction category (DC) labels, a new label format for enforcing information flow in the presence of mutually distrusting parties. DC labels can be ordered to form a lattice, based on propositional logic implication and conjunctive normal form. We introduce and prove soundness of decentralized privileges that are used in declassifying data, in addition to providing a notion of privilege-hierarchy. Our model is simpler than previous decentralized information flow control (DIFC) systems and does not rely on a centralized principal hierarchy. Additionally, DC labels can be used to enforce information flow both statically and dynamically. To demonstrate their use, we describe two Haskell implementations, a library used to perform dynamic label checks, compatible with existing DIFC systems, and a prototype library that enforces information flow statically, by leveraging the Haskell type checker.

4.1 Introduction

Information flow control (IFC) is a general method that allows components of a system to be passed sensitive information and restricts its use in each component. Information flow control can be used to achieve confidentiality, by preventing unwanted information leakage, and integrity, by preventing unreliable information from flowing into critical operations. Modern IFC systems typically label data and track labels, while allowing users exercising appropriate privileges to explicitly downgrade information themselves. While the IFC system cannot guarantee that downgrading preserves the desired information flow properties, it is possible to identify all the downgrading operations and

[1] This chapter is a copy of the NordSec 2011 paper [184].

limit code audit to these portions of the code. Overall, information flow systems make it possible to build applications that enforce end-to-end security policies even in the presence of untrusted code.

We present disjunction category (DC) labels: a new label format for enforcing information flow in systems with mutually distrusting parties. By formulating DC labels using propositional logic, we make it straightforward to verify conventional lattice conditions and other useful properties. We introduce and prove soundness of decentralized privileges that are used in declassifying data, and provide a notion of privilege-hierarchy. Compared to Myers and Liskov's decentralized label model (DLM) [144], for example, our model is simpler and does not rely on a centralized principal hierarchy. Additionally, DC labels can be used to enforce information flow both statically and dynamically, as shown in our Haskell implementations.

A DC label, written $\langle S, I \rangle$, consists of two Boolean formulas over principals, the first specifying secrecy requirements and the second specifying integrity requirements. Information flow is restricted according to implication of these formulas in a way that preserves secrecy and integrity. Specifically, secrecy of information labeled $\langle S, I \rangle$ is preserved by requiring that a receiving channel have a stronger secrecy requirement S' that *implies* S, while integrity requires the receiver to have a weaker integrity requirement I' that is *implied by* I. These two requirements are combined to form a can-flow-to relation, which provides a partial order on the set of DC labels that also has the lattice operations meet and join.

Our decentralized privileges can be delegated in a way that we prove preserves confidentiality and integrity properties, resulting in a privilege hierarchy. Unlike [144], this is accomplished without a notion of "can act for" or a central principal hierarchy. Although our model can be extended to support revocation using approaches associated with public key infrastructures, we present a potentially more appealing selective revocation approach that is similar to those used in capability-based systems.

We illustrate the expressiveness of DC labels by showing how to express several common design patterns. These patterns are based in part on security patterns used in capability-based systems. Confinement is achieved by labeling data so that it cannot be read and exfiltrated to the network by arbitrary principals. A more subtle pattern that relies on the notion of clearance is used to show how a process can be restricted from even accessing overly-sensitive information (e.g., private keys); this pattern is especially useful when covert channels are a concern. We also describe privilege separation and user authentication patterns. As described more fully later in the chapter, privilege separation may be achieved using delegation to subdivide the privileges of a program and compartmentalize a program into components running with fewer privileges. The user authentication pattern

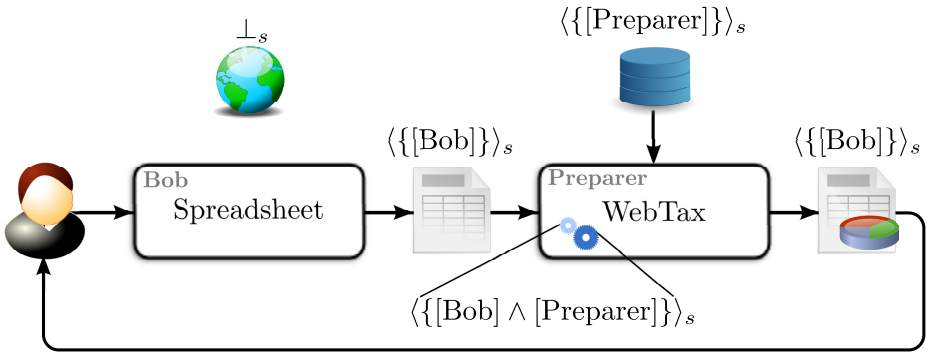

Figure 4.1: A tax preparation system with mutually distrusting parties.

shows how to leverage a login client that users trust with their username and password (since the user supplies them as input), without unnecessarily creating other risks.

We describe two Haskell implementations: a library used to perform dynamic label checks, compatible with existing DIFC systems, and a prototype library that enforces information flow statically by leveraging Haskell's type checker.

The remainder of the chapter is structured as follows. In Section 4.2, we introduce DC labels and present some of their properties. Section 4.3 presents semantics and soundness proofs for our DC label system. Design patterns are presented and explained in Section 4.5, with the implementations presented in Section 4.6. We summarize related work in Section 4.7 and conclude in Section 4.8.

4.2 DC Label Model

In a DIFC system, every piece of data is *labeled*, or "tagged." Labels provide a means for tracking, and, more importantly, controlling the propagation of information according to a security policy, such as *non-interference* [73].

DC labels can be used to express a conjunction of restrictions on information flow that represents the interests of multiple stake-holders. As a result, DC labels are especially suitable for systems in which participating parties do not fully trust each other. Figure 4.1 presents an example, originally given in [144], that illustrates such a system. Here, user Bob firstly inputs his tax information into the *Spreadsheet* program, which he fully trusts. The data is then exported to another program, called *WebTax*, for final analysis. Though conceptually simple, several challenges arise since Bob does not trust WebTax with his data. Without inspecting WebTax, Bob cannot be sure that his privacy policies are respected and his tax information is not exfiltrated to the network. Analogously, the

WebTax author, called Preparer, does not entrust Bob with the source code. Furthermore, the tax preparation program relies on a proprietary database and Preparer wishes to assert that even if the program contains bugs, the proprietary database information cannot be leaked to the public network. It is clear that even for such a simple example the end-to-end guarantees are difficult to satisfy with more-traditional access control mechanisms. Using IFC, however, these security policies can be expressed naturally and with minimal trust. Specifically, the parties only need to trust the system IFC-enforcement mechanism; programs, including WebTax, can be executed with no implicit trust. We now specify DC labels and show their use in enforcing the policies of this example.

As previously mentioned, a DC label consists of two Boolean formulas over principals. We make a few restrictions on the labels' format in order to obtain a unique representation for each label and an efficient and decidable can-flow-to relationship.

Definition 1 (DC Labels)**.** A *DC label*, written $\langle S, I \rangle$, is a pair of Boolean formulas over principals such that:

- Both S and I are minimal formulas in conjunctive normal form (CNF), with terms and clauses sorted to give each formula a unique representation, and
- Neither S nor I contains any negated terms.

In a DC label, S protects secrecy by specifying the principals that are allowed (or whose consent is needed) to observe the data. Dually, I protects integrity by specifying principals who created and may currently modify the data. For example, in the system of Figure 4.1, Bob and Preparer respectively label their data $\langle \{\text{Bob}\}, \{\text{Bob}\} \rangle$ and $\langle \{\text{Preparer}\}, \{\text{Preparer}\} \rangle$, specifying that they created the data and they are the only observers.

Data may flow between differently labeled entities, but only in such a way as to accumulate additional secrecy restrictions or be stripped in integrity ones, not vice versa. Specifically there is a partial order, written $\sqsubseteq$ ("can-flow-to"), that specifies when data can flow between labeled entities. We define $\sqsubseteq$ based on logical implication ($\implies$) as follows:

Definition 2 (can-flow-to relation)**.** Given any two DC labels $L_1 = \langle S_1, I_1 \rangle$ and $L_2 = \langle S_2, I_2 \rangle$, the can-flow-to relation is defined as:

$$\frac{S_2 \implies S_1 \qquad I_1 \implies I_2}{\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle}$$

In other words, data labeled $\langle S_1, I_1 \rangle$ can flow into an entity labeled $\langle S_2, I_2 \rangle$ as long as the secrecy of the data, and integrity of the entity are preserved. Intuitively, the $\sqsubseteq$ relation imposes the restriction that any set of principals who can observe data afterwards must also have been able to observe it earlier. For instance, it is permissible to have $S_2 = \{\text{Bob} \wedge \text{Preparer}\}$ and $S_1 = \text{Bob}$, because $S_2 \implies S_1$, and Bob's consent is still required to observe data with the new label. Dually, integrity of the entity is preserved by requiring that the source label impose more restrictions than that of the destination.

In our model, public entities (e.g., network interface in Figure 4.1) have the default, or *empty* label, $\langle \textbf{True}, \textbf{True} \rangle$, written L_{pub}. Although specified by the label $\langle S, I \rangle$, it is intuitive that data labeled as such can be written to a public network with label L_{pub}, only with the permission of a set of principals satisfying the Boolean formula S. Conversely, data read from the network can be labeled $\langle S, I \rangle$ only with the permission of a set of principals satisfying I.

In an IFC system, label checks using the can-flow-to relation are performed at every point of possible information flow. Thus, if the WebTax program of Figure 4.1 attempts to write Bob or Preparer's data to the network interface, either by error or malfeasance, both label checks $\langle \{\text{Bob}\}, \{\text{Bob}\} \rangle \sqsubseteq L_{\text{pub}}$ and $\langle \{\text{Preparer}\}, \{\text{Preparer}\} \rangle \sqsubseteq L_{\text{pub}}$ will fail. However, the system must also label the intermediate results of a WebTax computation (on Bob and Preparer's joint data) such that they can only be observed and written to the network if both principals consent.

The latter labeling requirement is recurring and directly addressed by a core property of many IFC systems: the label lattice property [51]. Specifically, for any two labels L_1, L_2 the lattice property states that there is a well defined, *least upper bound* (*join*), written $L_1 \sqcup L_2$, and *greatest lower bound* (*meet*), written $L_1 \sqcap L_2$, such that $L_i \sqsubseteq L_1 \sqcup L_2$ and $L_1 \sqcap L_2 \sqsubseteq L_i$ for L_i and $i = 1, 2$. We define the join and meet for DC labels as follows.

Definition 3 (Join and meet for DC labels)**.** The join and meet of any two DC labels $L_1 = \langle S_1, I_1 \rangle$ and $L_2 = \langle S_2, I_2 \rangle$ are respectively defined as:

$$L_1 \sqcup L_2 = \langle S_1 \wedge S_2, I_1 \vee I_2 \rangle$$
$$L_1 \sqcap L_2 = \langle S_1 \vee S_2, I_1 \wedge I_2 \rangle$$

where each component of the resulting labels is reduced to CNF.

Intuitively, the secrecy component of the join protects the secrecy of L_1 and L_2 by specifying that both set of principals, those appearing in S_1 and those in S_2, must consent for data labeled $S_1 \wedge S_2$ to be observed. Conversely, the integrity component of the join, $I_1 \vee I_2$, specifies that either

principals of I_1 or I_2 could have created and modify the data. Dual properties hold for the meet $L_1 \sqcap L_2$, a label computation necessary when labeling an object that is written to multiple entities. We note that although we use $I_1 \vee I_2$ informally, by definition, a DC label component must be in CNF. Reducing logic formulas, such as $I_1 \vee I_2$, to CNF is standard [147], and we do not discuss it further.

Revisiting the example of Figure 4.1, we highlight that the intermediate results generated by the WebTax program from both Bob and Preparer's data are labeled by the join $\langle\{\text{Bob}\},\{\text{Bob}\}\rangle \sqcup \langle\{\text{Preparer}\},\{\text{Preparer}\}\rangle$ which is reduced to $\langle\{\text{Bob} \wedge \text{Preparer}\},\{\text{Bob} \vee \text{Preparer}\}\rangle$. The secrecy component of the label confirms our intuition that the intermediate results are composed of both party's data and thus the consent of both Bob and Preparer is needed to observe it. In parallel, the integrity component agrees with the intuition that the intermediate results could have been created from Bob or Preparer's data.

4.2.1 Declassification and endorsement

We model both declassification and endorsement as principals explicitly deciding to exercise *privileges*. When code exercises privileges, it means code acting on behalf of a combination of principals is requesting an action that might violate the can-flow-to relation. For instance, if the secrecy component of a label is $\{\text{Bob} \wedge \text{Preparer}\}$, then by definition code must act on behalf of both Bob and the Preparer to transmit the data over a public network. However, what if the Preparer unilaterally wishes to change the secrecy label on data from $\{\text{Bob} \wedge \text{Preparer}\}$ to $\{\text{Bob}\}$ (as to release the results to Bob)? Intuitively, such a partial declassification should be allowed, because the data still cannot be transmitted over the network without Bob's consent. Hence, if the data is eventually made public, both Bob and the Preparer will have consented, even if not simultaneously.

We formalize such partial declassification by defining a more permissive pre-order, $\sqsubseteq_P$ ("can-flow-to given privileges P"). $L_1 \sqsubseteq_P L_2$ means that when exercising privileges P, it is permissible for data to flow from an entity labeled L_1 to one labeled L_2. $L_1 \sqsubseteq L_2$ trivially implies $L_1 \sqsubseteq_P L_2$ for any privileges P, but for non-empty P, there exist labels for which $L_1 \sqsubseteq_P L_2$ even though $L_1 \not\sqsubseteq L_2$.

We represent privileges P as a conjunction of principals for whom code is acting. (Actually, P can be a more general Boolean formula like label components, but the most straight-forward use is as a simple conjunction of principals.) We define $\sqsubseteq_P$ as follows:

Definition 4 (can-flow-to given privileges relation)**.** Given a Boolean formula P representing privileges and any two DC labels $L_1 = \langle S_1, I_1\rangle$ and $L_2 = \langle S_2, I_2\rangle$, the can-flow-to given privileges P

relation is defined as:

$$\frac{P \wedge S_2 \implies S_1 \qquad P \wedge I_1 \implies I_2}{\langle S_1, I_1 \rangle \sqsubseteq_P \langle S_2, I_2 \rangle}$$

Recall that without exercising additional privileges, data labeled $\langle S, I \rangle$ can be written to a public network, labeled L_{pub}, only with the permission of a set of principals satisfying the Boolean formula S, while data read from a public network can be labeled $\langle S, I \rangle$ only with the permission of a set of principals satisfying I. Considering additional privileges, it is easy to see that $\langle S, I \rangle \sqsubseteq_P L_{\text{pub}}$ iff $P \implies S$ and, conversely, $L_{\text{pub}} \sqsubseteq_P \langle S, I \rangle$ iff $P \implies I$. In other words, code exercising privileges P can declassify and write data to the public network if P implies the secrecy label of that data, and can similarly incorporate and endorse data from the public network if P implies the integrity label.

In our WebTax example, the Spreadsheet program runs on behalf of Bob and exercises the $\{\text{Bob}\}$ privilege to endorse data sent to WebTax. Conversely, the WebTax program is executed with the $\{\text{Preparer}\}$ privilege which it exercises when declassifying results from $\{\text{Bob} \wedge \text{Preparer}\}$ to $\{\text{Bob}\}$; as expected, to allow Bob to observe the results, this declassification step is crucial.

It is a property of our system that exporting data through multiple exercises of privilege cannot reduce the overall privilege required to export data. For instance, if $\langle S, I \rangle \sqsubseteq_{P_1} \langle S', I' \rangle \sqsubseteq_{P_2} L_{\text{pub}}$, it must be that $P_1 \wedge P_2 \implies S$, since $P_2 \implies S'$ and $P_1 \wedge S' \implies S$. A similar, and dual, property holds for multiple endorsements.

The mechanisms provided by $\sqsubseteq_P$ corresponds to the *who* dimension of declassification [169], i.e., whoever has the privileges P can use the relationship $\sqsubseteq_P$ to release (endorse) information. With minimal encoding, it is also possible to address the *what* and *when* dimension using $\sqsubseteq_P$. Specifically, the what dimension can be addressed by carefully designing the data type in such a way that there is an explicit distinction on what part of the data is allowed to be released. The *when* dimension, on the other hand, consists on designing the trusted modules in such a way that certain privileges can only be exercised when some, well-defined, events occurs.

In our model, privileges can be *delegated*. Specifically, a process may delegate privileges to another process according to the following definition:

Definition 5 (Can-delegate relation)**.** A process with privilege P can delegate any privilege P', such that $P \implies P'$.

In other words, it is possible to delegate a privilege P' that is at most as strong as the parent privilege P. In Section 4.5, we give a concrete example of using delegation to implement a privilege separation.

4.2.2 Ownership and categories

Our definition of DC label components as conjunctions of clauses, each imposing an information flow restriction, is similar to the DStar [229] label format which uses a set of *categories*, each of which is used to impose a flow restriction. Though the name category may be used interchangeably with clause, our categories differ from those of DStar (or even DLM) in that they are disjunctions of principals—hence the name, *disjunction category* labels.

The principals composing a category are said to *own* the category—every owner is trusted to uphold or bypass the restriction imposed by the category. For instance, the category $[\text{Bob} \vee \text{Alice}]$ is owned by both Alice and Bob. We can thus interpret the secrecy component $\{[\text{Bob} \vee \text{Alice}] \wedge \text{Preparer}\}$ to specify that data can be observed by the Preparer in collaboration with *either* Bob or Alice. Though implicit in our definition of a DC label, this joint ownership of a category allows for expressing quite complex policies. For example, to file joint taxes with Alice, Bob can simply labels the tax data $\langle \{[\text{Bob} \vee \text{Alice}]\}, \{\text{Bob}\} \rangle$, and now the WebTax results can be observed by both him and Alice. Expressing such policies in other systems, such as DLM or DStar, can only be done through external means (e.g., by creating a new principal AliceBob and encoding its relationship to Alice and Bob in a centralized principal hierarchy).

In the previous section we represent privileges P as a conjunction of principals for whom code is acting. Analogous to a principal owning a category, we say that a process (or computation) *owns* a principal if it acting or running on its behalf. (More generally, the code is said to own all the categories that compose P.)

4.3 Soundness

In this section, we show that the can-flow-to relation ($\sqsubseteq$) and the relation ($\sqsubseteq_P$) for can-flow-to given privileges P satisfy various properties. We first show that $\sqsubseteq$, given in Definition 2, is partial order.

Lemma 1 (DC labels form a partially ordered set)**.** *The binary relation $\sqsubseteq$ over the set of all DC labels is a partial order.*

Proof. Reflexivity and transitivity follow directly from the Reflexivity and transitivity of ($\implies$). By Definition 1, the components of a label, and thus the label, have a unique representation. Directly, the antisymmetry property holds. □

Recall from Section 4.2 that for any two labels L_1 and L_2 there exists a join $L_1 \sqcup L_2$ and meet $L_1 \sqcap L_2$. The join must be the least upper bound of L_1 and L_2, with $L_1 \sqsubseteq L_1 \sqcup L_2$, and $L_2 \sqsubseteq L_1 \sqcup L_2$;

conversely, the meet must be the greatest lower bound of L_1 and L_2, with $L_1 \sqcap L_2 \sqsubseteq L_1$ and $L_1 \sqcap L_2 \sqsubseteq L_2$. We prove these properties and show that DC labels form a lattice.

Proposition 1 (DC labels form a bounded lattice)**.** *DC labels with the partial order relation $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$ form a bounded lattice with minimum element $\bot = \langle \mathbf{True}, \mathbf{False} \rangle$ and maximum element $\top = \langle \mathbf{False}, \mathbf{True} \rangle$.*

Proof. The lattice property follows from Lemma 1, the definition of DC labels, and the definition of the join and meet as given in Definition 3. □

It is worth noting that the DC label lattice is actually product lattice, i.e., a lattice where components are elements of a secrecy and (a dual) integrity lattice [224].

In Section 4.2.1 we detailed declassification and endorsement of data in terms of exercising privileges. Both actions constitute bypassing restrictions of $\sqsubseteq$ by using a more permissive relation $\sqsubseteq_P$. Here, we show that this privilege-exercising relation, as given in Definition 4, is a pre-order and that privilege delegation respects its restrictions.

Proposition 2 (The $\sqsubseteq_P$ relation is a pre-order)**.** *The binary relation $\sqsubseteq_P$ over the set of all DC labels is a pre-order.*

Proof. Reflexivity and transitivity follow directly from the reflexivity and transitivity of ($\Longrightarrow$). Unlike $\sqsubseteq$, however, $\sqsubseteq_P$ is not necessarily antisymmetric (showing this, for a non-empty P, is trivial). □

Informally, exercising privilege P may allow a principal to ignore the distinction between certain pairs of clauses, hence $\sqsubseteq_P$ is generally not a partial order. Moreover, the intuition that $\sqsubseteq_P$, for any non-empty P, is always more permissive than $\sqsubseteq$ follows as a special case of the following proposition.

Proposition 3 (Privileges substitution)**.** *Given privileges P and P', if $P \Longrightarrow P'$ then P can always be substituted in for P'. Specifically, for all labels L_1 and L_2, if $P \Longrightarrow P'$ and $L_1 \sqsubseteq_{P'} L_2$ then $L_1 \sqsubseteq_P L_2$.*

Proof. First, we note that if $P \Longrightarrow P'$, then for any X, X', such that $X \wedge P' \Longrightarrow X'$, the proposition $X \wedge P \Longrightarrow X \wedge P' \Longrightarrow X'$ holds trivially. By Definition 4, $L_1 \sqsubseteq_{P'} L_2$ is equivalent to: $S_2 \wedge P' \Longrightarrow S_1$ and $I_1 \wedge P' \Longrightarrow I_2$. However, from $P \Longrightarrow P'$, we have $S_2 \wedge P \Longrightarrow S_2 \wedge P' \Longrightarrow S_1$, and $I_1 \wedge P \Longrightarrow I_1 \wedge P' \Longrightarrow I_2$. Correspondingly, we have $L_1 \sqsubseteq_P L_2$. □

Informally, if a piece of code exercises privileges P' to read or endorse a piece of data, it can do so with P as well. In other words, $\sqsubseteq_P$ is at least at as permissive as $\sqsubseteq_{P'}$. Letting $P' = \textbf{True}$, it directly follows that for any non-empty P, i.e., for $P \neq \textbf{True}$, the relation $\sqsubseteq_P$ is more permissive than $\sqsubseteq$. Moreover, negating the statement of the proposition (if $L_1 \not\sqsubseteq_P L_2$ then $L_1 \not\sqsubseteq_{P'} L_2$) establishes that if exercising a privilege P does not allow for the flow of information from L_1 to L_2, then exercising a privilege delegated from P will also fail to allow the flow. This property is especially useful in guaranteeing soundness of privilege separation.

4.4 Model Extensions

The base DC label model, as described in Section 4.2, can be used to implement complex DIFC systems, despite its simplicity. Furthermore, the model can easily be further extended to support features of existing security (IFC and capability) systems, as we detail below.

4.4.1 Principal hierarchy

As previously mentioned, DLM [144] has a notion of a principal hierarchy defined by a reflexive and transitive relation, called *acts for*. Specifically, a principal p can act for another principal p', written $p \succeq p'$, if p is at least as powerful as p': p can read, write, declassify, and endorse all objects that p' can; the principal hierarchy tracks such relationships.

To incorporate this feature, we modify our model by encoding the principal hierarchy as a set of axioms Γ. Specifically, if $p \succeq p'$, then $(p \implies p') \in \Gamma$. Consequently, Γ is used as a hypothesis in every proposition. For example, without the principal hierarchy $\emptyset \vdash p_1 \implies [p_2 \vee p_3]$ does not hold, but if $p_1 \succeq p_2$ then $(p_1 \implies p_2), \Gamma \vdash p_1 \implies [p_2 \vee p_3]$ does hold. We, however, note that our notion of privileges and label component clauses (disjunction categories) can be used to capture such policies, that are expressible in DLM only through the use of the principal hierarchy. Compared to DLM, DC labels can be used to express very flexible policies (e.g., joint ownership) even when $\Gamma = \emptyset$.

4.4.2 Using DC labels in a distributed setting

For scalability, extending a system to a distributed setting is crucial. Addressing this issue, Zeldovich et al. [229], provide a distributed DIFC system, called DStar. DStar is a framework (and protocol) that extends OS-level DIFC systems to a distributed setting. Core to DStar is the notion

of an *exporter* daemon, which, among other things, maps DStar network labels to OS local labels such as DC labels, and conversely. DC labels (and privileges) are a generalization of DStar labels (and privileges)—the core difference being the ability of DC labels to represent joint ownership of a category with disjunctions, a property expressible in DStar only with privileges. Hence, DC labels can directly be used when extending a system to a distributed setting. More interestingly, however, we can extend DStar, while remaining backwards compatible (since every DStar label can be expressed using a DC label), to use disjunction categories and thus, effectively, use DC labels as network labels—this extension is part of our future work.

4.4.3 Delegation and pseudo-principals

As detailed in Section 4.2.1, our decentralized privileges can be delegated and thus create a privilege hierarchy. Specifically, a process with a set of privileges may delegate a category it owns (in the form of a single-category privilege), which can then be further *granted* or delegated to another process.

In scenarios involving delegated privileges, we introduce the notion of a *pseudo-principal*. Pseudo-principals allows one to express providence on data, which is particularly useful in identifying the contributions of different computations to a task. A pseudo-principal is simply a principal (distinguished by the prefix #) that cannot be owned by any piece of code and can only be created when a privilege is delegated. Specifically, a process that owns principal p may delegate a single-category privilege $\{[p \vee \#c]\}$ to a piece of code c. The disjunction is used to indicate that the piece of code c is responsible for performing a task been delegated by the code owing p, which also does not trust c with the privilege p. Observe that the singleton $\{\#c\}$ cannot appear in any privilege, and as a result, if some data is given to p with the integrity restriction $[p \vee \#c]$, then the piece of code c must have been the originator. In a system with multiple components, using pseudo-principals, one can enforce a pipeline of operations, as shown by the implementation of a mail delivery agent in Section 4.6.

We note that pseudo-principals are treated as ordinary principals in label operations. Moreover, in our implementation, the distinction is minimal: principals are strings that cannot contain the character '`#`', while pseudo-principals are strings that always have the prefix '`#`'.

4.4.4 Privilege revocation

In dynamic systems, security policies change throughout the lifetime of the system. It is common for new users to be added and removed, as is for privileges to be granted and revoked [29]. Although

our model can be extended to support revocation similar to that of public key infrastructures [76], we describe a selective revocation approach, common to capability-based systems [159].

To allow for the flexibility of selective revocation, it is necessary to keep track of a delegation chain with every category in a delegated privilege. For example, if processes A and C respectively delegate the single-principal privileges $\{a\}$ and $\{c\}$ to process B, B's privilege will be encoded as $\{(\{A \to B\}, a), (\{C \to B\}, c)\}$. Similarly, if B delegates $\{[a \vee c]\}$ to D, the latter's privilege set will be $\{(\{A \to B \to D, C \to B \to D\}, [a \vee c])\}$. Now, to selectively revoke a category, a process updates a system-wide revocation set Ψ with a pair consisting of the chain prefix and a privilege (it delegated) to be revoked. For example, A can revoke B's ownership of $\{a\}$ by adding $(\{A \to B\}, a)$ to Ψ. Consequently, when B or D perform a label comparison involving privileges, i.e., use $\sqsubseteq_P$, the revocation set Ψ is consulted: since $A \to B$ is a prefix in both cases, and $a \implies a$ and $a \implies [a \vee c]$, neither B nor D can exercise their delegated privileges. More generally, ownership of single-category privilege $\{c\}$ with chain x is revoked if there is a pair $(y, \psi) \in \Psi$ such that the chain y is a prefix of a chain in x and $\psi \implies c$. We finally note that, although this description of revocation relies on a centralized revocation set Ψ, selective revocation, in practice, can be implemented without a centralized set, using patterns such as Redell's "caretaker pattern" [159, 128] with wrapper, or membrane, objects transitively applying the revocation [130, 128].

We note that it is also possible to support revocation in the system extended with a principal hierarchy by allowing for the removal of can act for relationships. Hicks et al. [83], consider such arbitrary principal hierarchy modifications in the context of a security-typed language.

4.5 Security Labeling Patterns

When building practical IFC systems, there are critical design decisions involving: (1) assigning labels to entities (data, channels, etc.), and (2) delegating privileges to executing code. In this section, we present *patterns* that can be used as a basis for these design decisions, illustrated using simplified examples of practical system applications.

4.5.1 Confinement and access control

A very common security policy is *confinement*: a program is allowed to compute on sensitive data but cannot export it [107, 170]. The tax-preparation example of Section 4.2 is a an examples of a system that enforces confinement.

In general, we may wish to confine a computation and guarantee that it does not release (by

declassification) user A's sensitive data to the public network or any other channel. Using the network as an illustrative example, and assuming A's sensitive data is labeled L_A, confinement may be achieved by executing the computation with privileges P chosen such that $L_A \not\sqsubseteq_P L_{\text{pub}}$. A complication is that most existing IFC systems (though not all, see, e.g., [55, 97]) are susceptible to covert channel attacks that circumvent the restrictions based on labels and privileges. For example, a computation with no privileges might read sensitive data and leak information by, e.g., not terminating or affecting timing behavior. To address confinement in the presence of covert channels, we use the notion of *clearance* [53], previously introduced and formalized in [228, 228, 187] in the context of IFC.

Clearance imposes an upper bound on the sensitivity of the data that the computation can read. To prevent a computation from accessing (reading or writing) data labeled L_A, we set the computation's clearance to some L_C such that $L_A \not\sqsubseteq L_C$. With this restriction, the computation may read data labeled L_D only if $L_D \sqsubseteq L_C$. Observe that in a similar manner, clearance can be used to enforce other forms of discretionary access control.

4.5.2 Privilege separation

Using delegation, a computation may be compartmentalized into sub-computations, with the privileges of the computation subdivided so that each sub-computation runs with *least privilege*. Consider, for example, a privilege-separated mail delivery agent (MDA) that performs spam filtering.

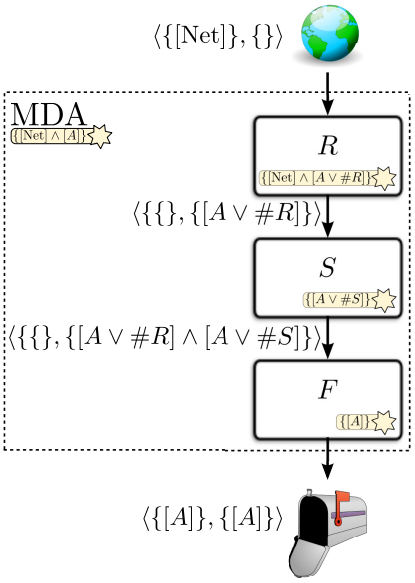

Figure 4.2: Simple MDA.

As with many real systems, the example MDA of Figure 4.2 is composed of different, and possibly untrustworthy, modules. In this example, the components are a network receiver, R, and a spam filter, S. Instead of combining the components into a monolithic MDA, the MDA author can segregate the untrustworthy components and execute then with the principle of least privilege. This avoids information leaks and corruption due to negligence or malfeasance on the component authors' part. Specifically, the receiver R is executed with the delegated privilege $\{[A \vee \#R]\}$, and the spam filter S is executed with the privilege $\{[A \vee \#S]\}$. As a consequence, R and S cannot read A's sensitive information and leak it to the network, corrupt A's mailbox, nor forge data on A's behalf.

Additionally, the MDA can enforce the policy that a mail message always passes through both receiver R and spam filter S. To this end, the MDA includes a small, trusted forwarder F, running with the privilege $\{A\}$, which endorses messages on behalf of A and writes them to the mailbox only after checking that they have been endorsed by both R and S. In a similar manner, this example can be further extended to verify that the provenance of a message is the network interface, or that the message took a specific path (e.g., R then S, but not S then R), among other.

In detail, the receiver R reads messages from the network, declassifies them by removing the [Net] category from their secrecy components, endorses the declassified messages by adding the $A \vee \#R$ to their integrity component, and, finally, forwards the messages to the sanitizer S. Since R is not executed with the full privilege $\{A\}$, even if compromised, it cannot create files with a label that has $[A]$ in the integrity component, and thereby pollute A's mailbox. Furthermore, R cannot pass unfiltered messages to the forwarder F, skipping the sanitizer, since it cannot create or endorse messages such that the resulting-label's integrity category contains category $[A \vee \#S]$, a restriction imposed by F.

Assuming the receiver passes messages to the sanitizer S, the latter filters messages and exercising privilege $\{[A \vee \#S]\}$ to endorse the filtered messages. At this point in the pipeline, a message should be labeled $\langle \{\}, \{[A \vee \#R] \wedge [A \vee \#S]]\} \rangle$. It is important to note that, like R, S cannot pollute A's mailbox by writing messages directly. Moreover, unlike R, if S is compromised, it cannot forge messages and forward them to F, since S does not have the privilege to endorse messages with the category $[A \vee \#R]$. The S-endorsed messages are further passed on to the forwarder. The forwarder F simply checks if receiving messages have been endorsed by both R and S, and, exercising privilege $\{[A]\}$ relabels the messages to $\langle \{[A]\}, \{[A]\} \rangle$. This final message is then delivered to A's mailbox.

4.5.3 User authentication

Another common requirement of security systems is user authentication. We consider password-based login as an example, where a successful authentication corresponds to granting the authenticated user the set of privileges associated with their credentials. Furthermore, we consider authentication in the context of (typed) language-level DIFC systems; an influential OS-level approach has been considered in [228]. Shown in Figure 4.3 is an example system which consists of a login client L, and an authentication service A_U.

To authenticate user U, the login client *invokes* the user authentication service A_U, which runs with the $\{U\}$ privilege. Conceptually, when invoked with U's correct credentials, A_U grants (by delegating) the caller the $\{U\}$ privilege. However, in actuality, the login client and authentication

service are in mutual distrust: L does not trust A_U with U's password, for A_U might be malicious and simply wish to learn the password, while A_U does not trust L to grant it the $\{U\}$ privilege without first verifying credentials. Consequently, the authentication requires several steps.

We note that due to the mutual distrust, the user's stored salt s and password hash $h = H(p\|s)$ is labeled with both, the user and login client's, principals, i.e., h and s have label $\langle\{U \wedge L\}, \{U \wedge L\}\rangle$. Solely, labeling them $\langle\{U\}, \{U\}\rangle$ would allow A_U to carry out an off-line attack to recover p. The authentication procedure is as follows.

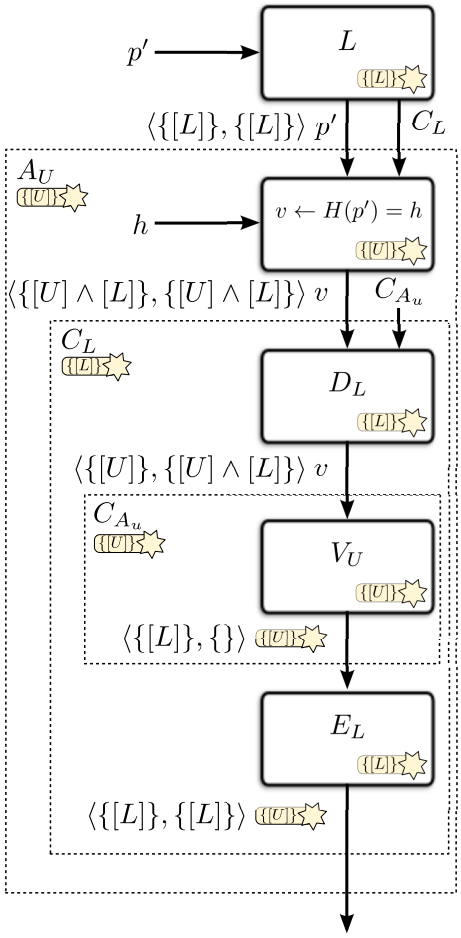

Figure 4.3: User authentication.

1. The user's input password p' to the login client is labeled $\langle\{L\}, \{L\}\rangle$, and along with a closure C_L is passed to the authentication service A_U. As further detailed below, closures are used in this example as a manner to exercise privileges under particular conditions and operations.

2. A_U reads U's stored salt s and password hash h. It then computes the hash $h' = H(p'\|s)$ and compares h' with h. The label of this result is simply the join of h and h': $\langle\{U \wedge L\}, \{L\}\rangle$. Since A_U performed the computation, it endorses the result by adding U to its integrity component; for clarity, we name this result v, as show in Fig 4.3.

   Remark: At this point, neither L nor A_U are able to read and fully declassify the secret password-check result v. Moreover, without eliminating the mutual distrust, neither L nor S can declassify v directly. Consider, for example, if A_U is malicious and had, instead, performed a comparison of $H(p'\|s)$ and $H(p''\|s)$, for some guessed password p''. If L were to declassify the result, A_U would learn that $p = p''$, assuming the user typed in the correct password, i.e., $p = p'$. Hence, we rely on purely functional (and statically-typed) closures to carry out the declassification indirectly.

3. When invoking A_U, L passed a declassification closure C_L, which has the $\{L\}$ privilege locally bound. Now, A_U invokes C_L with v and its own declassification closure C_{A_U}.

4. C_L declassifies v (D_L in Figure 4.3) to $\langle\{U\},\{U \wedge L\}\rangle$, and then invokes C_{A_U} with the new, partially-declassified result.

5. The C_{A_U} closure has the $\{U\}$ privilege bound and upon being invoked, simply verifies the result and its integrity (V_U in Figure 4.3). If the password is correct v is true and then C_{A_U} returns the privilege $\{U\}$ labeled with $\langle\{L\},\mathbf{True}\rangle$; otherwise it returns the empty privilege set. It is important that the integrity of v be verified, for a malicious L could provide a closure that forges password-check results, an attempt to wrongfully gain privileges.

6. The privilege returned from invoking C_{A_U} is endorsed by C_L (E_L in Figure 4.3), only if its secrecy component is L. This asserts that upon returning the privilege from C_L, A_U cannot check if the privilege is empty or not, and thus infer the comparison result.

7. It only remains for A_U to forward the labeled privilege back to L.

We finally note that the authentication service is expected to keep state that tracks the number of attempts made by a login client, as each result leaks a bit of information; to limit the number of unsuccessful attempts requires the use of a (minimal) code that is trusted by both L and A_U, as shown in [228].

4.6 Implementing DC Labels

We present two Haskell implementations of DC labels. The first, `dclabel`, is a library that provides a simple embedded domain specific language for constructing and working with dynamic labels and privileges. Principals in the `dclabel` library are represented by strings, while label components are lists of clauses (categories), which, in turn, are lists of principals. We use lists as sets for simplicity and because Haskell supports list comprehension; this allowed for a very simple translation from the formal definitions of this chapter to (under 180 lines of) Haskell code. We additionally implemented the instances necessary to use DC labels with the LIO. Given the simplicity of the implementation, we believe that porting it to other libraries, such as [115, 94], can be accomplished with minimal effort. Finally, we note that while our implementation was thoroughly tested using the QuickCheck, a future direction for this work is to use the Coq specification of DC Labels from [136] as to extract provably-correct Haskell code.

Although we have primarily focused on dynamic IFC, in cases where covert channels, runtime overhead, or failures are not tolerable, DC labels can also be used to enforce IFC statically. To this end, we implement `dclabel-static`, a prototype IFC system that demonstrates the feasibility of statically enforcing DIFC using secrecy-only DC labels, without modifying the Haskell language or the GHC compiler. Since DC labels are expressed using propositional logic, a programming language that has support for sum, product, and function types can be used, *without modification*, to enforce information flow control according to the Curry-Howard correspondence [84, 70]. According to the correspondence, disjunction, conjunction and implication respectively correspond to sum, product, and function types. Hence, for a secrecy-only DC label, to prove $L_1 \sqsubseteq L_2$, i.e., $L_2 \implies L_1$, we need only construct a function that has type $L_2 \to L_1$: successfully type-checking a program directly corresponds to verifying that the code does not violate IFC.

The library exports various type classes and combinators that facilitates the enforcement of static IFC. For example, we provide type constructors to create labels from principals—a principal in this system is a type for which an instance of the `Principal` type class is defined. To label values, we associate labels with types. Specifically, a labeled type is a wrapper for a product type, whose first component is a label, and whose second component, the value, cannot be projected without declassification. The library further provides a function, `relabel`, which, given a labeled value (e.g., $(L_1, 3)$), a new label L_2, and a proof of $L_1 \sqsubseteq L_2$ (a lambda term of type $L_2 \to L_1$), returns the relabeled value (e.g., $(L_2, 3)$). Since providing such proofs is often tedious, we supply a tool called `dcAutoProve`, that automatically inserts proofs of can-flow-to relations for expressions named `auto`, with an explicit type signature. Our automated theorem prover is based a variant of Gentzen's LJ sequent calculus [61].

4.7 Related Work on Label Models

DC labels closely resemble DLM labels [144] and their use in Jif [145]. Like DC labels, DLM labels express both secrecy and integrity policies. Core to a DLM label are components that specify an owner (who can declassify the data) and a set of readers (or writers). Compared to our disjunction categories, DLM does not allow for joint ownership of a component—they rely on a centralized principal hierarchy to express partial ownership. However, policies (natural to DLM) which allow for multiple readers, but a single owner, in our model, require a labeling pattern that relies on the notion of clearance, as discussed in Section 4.5 and used in existing DIFC systems [228, 229, 187]. Additionally, unlike to DLM labels as formalized in [143], DC labels form a bounded lattice with

a join and meet that respectively correspond to the least upper bound and greatest lower bound; the meet for DLM labels is not always the greatest lower bound.

The language Paralocks [33] uses Horn clauses to encode fine-grained IFC policies following the notion of locks: certain flows are allowed when corresponding locks are open. Constraining our model to the case where a privilege set is solely a conjunction of principals, Paralocks be easily used to encode our model. However, it remain an open problem to determine if disjunctive privileges can be expressed in their notion of *state*.

The Asbestos [63] and HiStar [228] operating systems enforce DIFC using Asbestos labels. Asbestos labels use the notion of categories to specify information flow restrictions in a similar manner to our clauses/categories. Unlike DC labels, however, Asbestos labels do not rely on the notion of principals. We can map a subset of DC labels to Asbestos labels by mapping secrecy and integrity categories to Asbestos levels **3** and **0**, respectively. Similarly ownership of a category maps to level $\star$. This mapping is limited to categories with no disjunction, which are equivalent to DStar labels [229], as discussed in Section 4.4. Mapping disjunction categories can be accomplished by using the system's notion of privileges. Conversely, both Asbestos and DStar labels are subsumed by our model. Moreover, compared to these systems we give precise semantics, prove soundness of the label format, and show its use in enforcing DIFC statically.

Montagu *et al.* present various embeddings between different label models, including DLM, DC Labels, HiStar, DStar, and Asbestos labels. Through these embeddings they formally compare the expressiveness of the different models, taking into consideration sub lattices, privileges, principal hierarchies, etc. We refer the interested reader to their paper and the related worked presented therein.

Capability-based systems such as KeyKOS [28], and E [130] are often used to restrict access to data. Among other purposes, capabilities can be used to enforce discretionary access control (DAC), and though they can enforce MAC using patterns such as membranes, the capability model is complimentary. For instance, our notion of privilege is a capability, while a delegated privilege loosely corresponds to an attenuated capability. This notion of privileges as capabilities is like that of Flume [104]. However, whereas they consider two types of privilege (essentially one for secrecy and another for integrity), our notion of privilege directly corresponds to ownership and conferring the right to exercise it in any way. Moreover, delegated privileges and the notion of disjunction provides an equal abstraction.

4.8 Conclusion

Decentralized information flow control can be used to build applications that enforce end-to-end security policies using untrusted code. DIFC systems rely on labels to track and enforce information flow. We present disjunction category labels, a new label format useful in enforcing information flow control in systems with mutually distrusting parties. In this chapter, we give precise semantics for DC labels and prove various security properties they satisfy. Furthermore, we introduce and prove soundness of decentralized privileges that are used in declassifying and endorsing data. Compared to Myers and Liskov's DLM, our model is simpler and does not rely on a centralized principal hierarchy, our privilege hierarchy is distributed. We highlight the expressiveness of DC labels by providing several common design and labeling patterns. Specifically, we show how to employ DC labels to express confinement, access control, privilege separation, and authentication. Finally, further illustrating flexibility of the model, we describe two Haskell implementations: a library used to perform dynamic label checks, compatible with existing DIFC systems, and a prototype library that enforces information flow statically by leveraging Haskell's module and type system.

Chapter 5

Flexible Dynamic Information Flow Control in the Presence of Exceptions[1]

We describe a language-based, dynamic information flow control (IFC) system called LIO. Our system presents a new design point for IFC, influenced by the challenge of implementing IFC as a Haskell library, as opposed to the more typical approach of modifying the language runtime system. In particular, we take a coarse-grained, floating-label approach, previously used by IFC Operating Systems, and associate a single, mutable label—the *current label*—with all the data in a computation's context. This label is always raised to reflect the reading of sensitive information and it is used to restrict the underlying computation's effects. To preserve the flexibility of fine-grained systems, LIO also provides programmers with a means for associating an explicit label with a piece of data. Interestingly, these labeled values can be used to encapsulate the results of sensitive computations which would otherwise lead to the creeping of the current label. Unlike other language-based systems, LIO also bounds the current label with a *current clearance*, providing a form of discretionary access control that LIO programs can use to deal with covert channels. Moreover, LIO provides programmers with mutable references and exceptions. The latter, exceptions, are used in LIO to encode and recover from monitor failures, all while preserving data confidentiality and integrity—this addresses a longstanding concern that dynamic IFC is inherently prone to information leakage due to monitor failure.

[1] This chapter is a copy of the extended version of the Haskell 2011 paper [186], which, at the time of this writing, is under revision at the Journal of Functional Programming.

5.1 Introduction

Information flow control (IFC) tracks the flow of data through a system and prohibits code from operating on data in violation of a security policy. Significant research, development, and experimental effort has been devoted to static information flow mechanisms. Static analysis has a number of benefits, including reduced runtime overhead, fewer runtime failures, and robustness against implicit flows [52]. However, static analysis in difficult to use in certain scenarios, such as web apps, where for example, users can join (or leave) the system arbitrarily, and where the security policy may depend on data provided by users, at runtime. For such systems, dynamic enforcement techniques are a more natural fit; dynamic IFC systems address many of the shortcomings of static IFC systems while retaining permissiveness.

Dynamic IFC systems fall into roughly two categories: fine-grained and coarse-grained enforcement. Fine-grained approaches, typically employed by language-based systems, e.g., [11, 12, 79, 86], explicitly associate security policies—or *labels*—with every value. Such systems have the benefit of giving programmers the ability to associate a particular security policy with a particular value. Unfortunately, this also places the burden of understanding and specifying labels on values that are not relevant for certain tasks. Moreover, fine-grained IFC systems are typically implemented as new (or changes to) languages or runtimes, imposing a large start-up cost on programmers.

Coarse-grained approaches, typically employed by IFC Operating Systems [201, 228, 104], associate a single label with every value in the context of a computation, usually a process. The advantage of such systems is simplicity: programmers do not need to clutter code with labels and can easily understand the security policy of any value—it is simply the label of the context. However, this is also a downside; programmers cannot associate a particular, and heterogeneous, security policy with a particular value. Moreover, incorporating sensitive data into a context usually amounts to "tainting" the whole context, which can lead to the *label creep* problem. Label creep occurs when the context label is tainted to a point where the computation cannot perform any useful side-effects.

In this chapter, we present LIO, a language-based dynamic IFC system, implemented as a Haskell library, that borrows ideas from both fine- and coarse-grained IFC systems. Like coarse-grained systems, LIO associates a label—the *current label*—with the current context. In particular, we define a monad, *LIO*, that restricts computations to a safe, IFC sublanguage of Haskell.[2] This

[2]Using *SafeHaskell* [197] we ensure that untrusted code executes in this sublanguage.

monad keeps track of the current label, which is, in turn, used to permit restricted access to IO functionality by executing actions in the underlying *IO* monad. Like many Operating Systems (OSes), LIO is a *floating-label* system; the current label is raised to allow reading of sensitive data, thereby "floating above" the labels of all data observed by the current computation. Of course, raising a computation's label comes at the cost of restricting where the computation may subsequently write.

Like fine-grained systems, LIO allows code to associate explicit labels with particular values, thus allowing applications to handle differently-labeled data in the same context. Specifically, LIO provides a *Labeled* type and a value constructor *label* that wraps explicit labels around values. Typically, labels are created at run time and incorporate dynamic information such as usernames or email addresses. LIO safely allows the label of a *Labeled* value to always be inspected. The wrapped value, on the other hand, can be inspected only using *unlabel*, a monadic function that appropriately raises the current label before returning the value.

Explicit unlabeling trivially addresses the problem of implicit flows endemic to fine-grained IFC systems, where control flow constructs are (ab)used to leak sensitive information. In LIO, code cannot branch on a *labeled boolean value* without first calling *unlabel* on the value; this ensures that the code cannot leak information via control flow. However, label creep could still occur if code keeps unlabeling heterogeneously labeled data. To address this problem, we introduce a function called *toLabeled*. This primitive executes a computation (that may raise the current label) and restores the current label upon its termination—i.e., it provides a separate context in which to execute the sensitive computation. Importantly, however, the result of the computation is encapsulated as a *Labeled* value—only when the (outer) computation wishes to inspect the result will the current label be raised.

Our dynamic IFC approach makes LIO more permissive than previous static approaches for functional languages (e.g., [154, 116, 164]), while still providing similar security guarantees [168]. Intuitively, dynamic IFC monitors, such as LIO, are more permissive since they only reject the run of a program if the executed code is about to violate policy. Static IFC analysis, on the other hand, would reject a program, even if a single line of unreachable code is insecure. But, of course static IFC analysis does not incur runtime overheard. More importantly, static approaches also do not usually suffer from covert channel leaks, present in most dynamic language-based IFC systems because of the typical stop-the-world semantics (see [144]). LIO addresses these limitations in several ways.

Unlike other language-based work, LIO limits the ability to leak information via covert channels by bounding the current label of a computation with a *current clearance*. The clearance of a region of code may be set to impose an upper bound on the floating current label within the region. Hence,

clearance can be understood as a discretionary access control mechanism that restricts the data that a subcomputation can access. And, by limiting access to data on a "need to know" basis, it reduces opportunities for code to leak data through covert channels—after all, code that cannot access sensitive data cannot leak it.

LIO furthermore addresses two limitations common to most dynamic fine-grained systems: the lack of exception handling facilities and inability to recover from IFC monitor failures (and thus the reason for stop-the-world semantics). Laminar [163], Breeze [86], and an early, unpublished, version of LIO [188] are the notable exceptions, further discussed in Section 5.7. Our "mostly coarse-grained" dynamic IFC approach makes it easy to reason about leaks due to exceptional control flow. In particular, we need only reason about exception propagation across *toLabeled* blocks since the current label is only restored, or "lowered," at these points; by treating computations executed by *toLabeled* as being of a separate context, the solution becomes clear: exceptions should not propagate outside the *toLabeled* block.

Equipped with exception-handling facilities, LIO encodes all IFC violations as catchable exceptions. This has the important consequence of allowing untrusted code to recover from IFC violations; this is in contrast to most language-level systems, which consider monitor failures fatal and leave the program in a stuck state (which itself may leak a bit). And, in contrast to Laminar, which also supports recovery from monitor failure (albeit in limited form, when compared to LIO), our uniform treatment of exceptions has led to a more flexible and permissive system (see Section 5.7)—as with other exceptions, LIO code can *always* recover from monitor failures.

This chapter extends an earlier conference paper [186] with *dynamic* exception-handling facilities, an implementation of a real-world conference review web application called λChair, and formal proofs mechanized in the Coq theorem prover. Moreover, this chapter corrects the formalism of the sequential LIO calculus to match the Haskell implementation; the formal semantics given in the original conference paper (and our tech report presenting an alternative semantics for dynamic IFC exceptions [188]) did not faithfully capture Haskell's evaluation strategy. The contributions of this chapter are the design, formalization, and implementation of a flexible and practical language-level dynamic IFC system. Our main contributions are as follows.

- We propose a new, mostly coarse-grained, design point for dynamic language-level IFC in which most values in lexical scope are protected by a single, mutable, current label. This design has the simplicity of OS-style IFC systems—e.g., because it alleviates the need for developers to annotate the sensitivity of all objects in scope. Instead, in LIO, programmers only

associate labels with values they care about by encapsulating them using the *Label* constructor. Such *Labeled* values are similar to labeled values in fine-grained programming languages IFC systems, but differ in a crucial way: our encapsulation is explicitly reflected by types in a way that prevents implicit flows. In a similar way, our calculus and Haskell implementation provides labeled mutable references. In contrast to the Laminar IFC system [163], which proposed a similar mostly coarse-grained system, LIO's mutable current label leads to a simpler and more flexible design—since it requires fewer annotations.

- Unlike other language-based work, our IFC model provides a notion of clearance which is used to provide a form of discretionary access control on code, i.e., it provides a way for restricting code to only access data it "needs to know." This is particularly useful in eliminating the opportunity for code to leak sensitive data by exploiting covert channels.

- We present a simple dynamic, yet safe, exception-handling mechanism and encode IFC monitor failures using exceptions. Exceptions are crucial to making LIO a practical IFC system; real-world applications cannot "stop the world" on an IFC violation attempt. This has been longstanding problem with dynamic IFC monitors, as highlighted by Myers and Liskov: "the difficulty with runtime checks is exactly the fact that they can *fail*... failure (or its absence) can serve as a covert channel [144]."

- We prove information flow, access control, and isolation security properties of our design. A large part of our formalization is encoded in Coq. We remark that while our formal description of LIO is Haskell-centric, this is not a fundamental restriction—our formalism can be generalized to other programming languages.

- We describe a Haskell implementation of the IFC calculus in Haskell. LIO can be implemented entirely as a library, demonstrating both the applicability and simplicity of the approach. This has the added benefit of not imposing the burden of learning a new programming language on developers—they simply need to understand a new API. Moreover, developers can use many existing compilers, tools, and libraries (e.g., roughly 12,500 Haskell modules on Hackage are safe to be used in LIO). Our library, applications built on top of it (including λChair), and Coq proofs are available at `http://labeled.io`.

This chapter is organized as follows. Section 5.2 describes the core information flow control LIO calculus. In Sections 5.3–5.5, we extend the core with clearance, mutable references, and exception-handling facilities. The security guarantees of the full calculus are given in Section 5.6. Related work

is described in Section 5.7. We conclude in Section 5.8.

5.2 Core dynamic information flow control LIO calculus

IFC systems track and restrict the propagation of information according to a security policy. The core policy enforced by LIO, and most other IFC systems, is *noninterference*. Noninterference guarantees confidentiality [73], by preventing sensitive information from being leaked to public entities, and integrity [22], by preventing unreliable information from flowing into critical operations.

In this section, we detail the core design of LIO and discuss the design trade-offs of a library-driven, mostly coarse-grained approach using the λChair conference review system as a driving example. In λChair, authenticated users can read any paper and can normally read any review. This reflects the normal practice in conference reviewing where, for example, every member of the program committee can see submissions and their reviews, and can participate in related discussion. In λChair, users can be added dynamically and assigned to review specific papers. Importantly, we use IFC to ensure that only assigned reviewers can write reviews for any given paper and that committee members in conflict with a paper cannot access the related discussions.

We incrementally describe the semantics of LIO using an extended simply-typed λ-calculus. First, we describe a pure base calculus. This calculus is then extended with labels (Section 5.2.2), labeled computations (Section 5.2.3), and labeled values (Section 5.2.4). Further leveraging labeled values we extend the calculus with *toLabeled* blocks to address label creep (Section 5.2.5). Finally, we extend this core with other features such as clearance, references and exceptions (Sections 5.3–5.5).

5.2.1 Base calculus for pure terms

Our semantics build on a pure, base calculus. The formal syntax of this base calculus is given in Figure 5.1. Syntactic categories v, t, and τ represent values, terms, and types, respectively. Values include primitives (Booleans *True*, *False*; and unit $()$) and functions ($\lambda x.t$). Terms constitute values (v), variables (x), function applications ($t_1\ t_2$), the standard fixpoint operator **fix** t, and conditionals (**if** t_1 **then** t_2 **else** t_3). Types consist of *Bool*, unit $()$,

$$\begin{aligned}
&\text{Values } v ::= \mathit{True} \mid \mathit{False} \mid () \mid \lambda x.t \\
&\text{Terms } t ::= v \mid x \mid t_1\ t_2 \mid \textbf{fix}\ t \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \\
&\text{Types } \tau ::= \mathit{Bool} \mid () \mid \tau_1 \to \tau_2
\end{aligned}$$

Figure 5.1: Formal syntax for values, terms, and types.

$$\textsc{appCtx}\ \frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2} \qquad \textsc{app}\ \frac{}{(\lambda x.t_1)\ t_2 \rightsquigarrow \{t_2 \,/\, x\}\ t_1} \qquad \textsc{fixCtx}\ \frac{t \rightsquigarrow t'}{\mathbf{fix}\ t \rightsquigarrow \mathbf{fix}\ t'} \qquad \textsc{fix}\ \frac{}{\mathbf{fix}\ (\lambda x.t) \rightsquigarrow \{\mathbf{fix}\ (\lambda x.t) \,/\, x\}\ t}$$

$$\textsc{ifCtx}\ \frac{t_1 \rightsquigarrow t_1'}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \rightsquigarrow \mathbf{if}\ t_1'\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3} \qquad \textsc{ifTrue}\ \frac{}{\mathbf{if}\ \mathit{True}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \rightsquigarrow t_2}$$

$$\textsc{ifFalse}\ \frac{}{\mathbf{if}\ \mathit{False}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \rightsquigarrow t_3}$$

Figure 5.2: Semantic rules for pure terms in the base LIO calculus.

and functions $\tau_1 \to \tau_2$.

Figure 5.2 shows the reduction rules for these pure terms using structural operational semantics [216]. The relation $t_1 \rightsquigarrow t_2$ represents a single evaluation step of pure term t_1 to term t_2; we say that t_1 reduces to t_2 in one step. We write $\rightsquigarrow^*$ for the reflexive and transitive closure of $\rightsquigarrow$.

Substitution $\{t_2 \,/\, x\}\ t_1$ is defined in the usual way, homomorphic on all operators, renaming bound names to avoid capture. The reduction rules for these terms are self-explanatory and very much the same as those of standard λ-calculus—we do not explain them further. We solely remark that our semantics does not model the sharing in lazy evaluation, as implemented by Haskell; modeling full lazy evaluation is beyond the scope of this chapter and has no impact on our termination- and timing-insensitive security guarantees.

LIO is implemented as a domain specific language embedded in Haskell. Hence, the typing judgements for our calculus are a subset of Haskell's and standard. We do not give any of the type judgements in this chapter. (The interested reader can see our Coq formalization.) Rather, we remark that LIO relies on types only to distinguish terms that can be used to compose safe computations and those that cannot, as further discussed in Section 5.2.3. Indeed, LIO can be generalized to dynamically-typed languages, as shown in Chapter 6.

5.2.2 Security lattice

To enforce security policies, like most modern dynamic IFC systems, LIO associates *labels* with objects. Labels encode confidentiality and integrity data policies which are propagated alongside the information they protect. In turn, the system mandatorily enforces these individual policies when

objects are read or written.

Labels are elements of a set $\mathscr{L}$ that forms a security lattice $(\mathscr{L}, \sqsubseteq, \sqcap, \sqcup)$, with partial order $\sqsubseteq$ (pronounced "can flow to"), binary join $\sqcup$, and binary meet $\sqcap$ [51]. The $\sqsubseteq$ relation is used by IFC systems when governing the allowed flows between differently labeled entities.[3] For example, LIO only allows data labeled $l_d \in \mathscr{L}$ to be written to a channel labeled $l_c \in \mathscr{L}$ if $l_d \sqsubseteq l_c$ holds true. The binary join is used to label computation results that depend on two objects by encoding the restrictions imposed by their labels, i.e., for labels $l_A, l_B \in \mathscr{L}$, the join $l_A \sqcup l_B$ is the smallest element such that $l_A \sqsubseteq l_A \sqcup l_B$ and $l_B \sqsubseteq l_A \sqcup l_B$. Dually, the binary meet $l_A \sqcap l_B$ encodes the intersection of the restrictions imposed by l_A and l_B; the meet is primarily used when labeling objects that we expect to be read by entities labeled l_A or l_B. Figure 5.3 shows how information flows in a simple lattice.

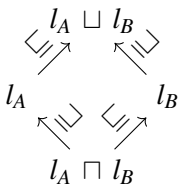

Figure 5.3: Simple example label lattice.

In LIO, labels are typed values. But, unlike most existing IFC systems [228, 145, 104, 86], LIO is polymorphic in the label format. We solely require that the label type provide definitions for lattice relations $\sqsubseteq$, $\sqcup$, and $\sqcap$. In Haskell, this amounts to making the label type an instance of the typeclass *Label*; all LIO library functions are qualified by *Label*:

$$\begin{aligned}
&\textbf{class } \mathit{Eq}\ \mathscr{L} \Rightarrow \mathit{Label}\ \mathscr{L} \textbf{ where} \\
&\quad (\sqsubseteq) :: \mathscr{L} \to \mathscr{L} \to \mathit{Bool} \\
&\quad (\sqcup) :: \mathscr{L} \to \mathscr{L} \to \mathscr{L} \\
&\quad (\sqcap) :: \mathscr{L} \to \mathscr{L} \to \mathscr{L}
\end{aligned}$$

As an example, consider the definition of the typical 2-point lattice $\mathscr{L}_2 = \{\mathit{Public}, \mathit{Secret}\}$, where $\mathit{Public} \sqsubseteq \mathit{Secret}$ and $\mathit{Secret} \not\sqsubseteq \mathit{Public}$:

$$\begin{aligned}
&\textbf{data } \mathscr{L}_2 = \mathit{Public} \mid \mathit{Secret} \textbf{ deriving } (\mathit{Eq}, \mathit{Ord}) \\
&\textbf{instance } \mathit{Label}\ \mathscr{L}_2 \textbf{ where} \\
&\quad x \sqsubseteq y = x \leqslant y \\
&\quad x \sqcup y = \mathit{max}\ x\ y \\
&\quad x \sqcap y = \mathit{min}\ x\ y
\end{aligned}$$

[3] Decentralized IFC (DIFC) extends IFC with the decentralized label model of Myers and Liskov [144], in which computations execute with a set of *privileges*, that can be used to loosen the restrictions imposed by the $\sqsubseteq$ relation. LIO supports privileges and the DIFC model in full. But, since our formalisation is limited to the system without privileges, we omit this from the presentation and refer the interested reader to the library documentation.

$$\textsc{LOpCtxL}\ \frac{t_1 \rightsquigarrow t_1'}{t_1 \otimes t_2 \rightsquigarrow t_1' \otimes t_2} \qquad \textsc{LOpCtxR}\ \frac{t_2 \rightsquigarrow t_2'}{l_1 \otimes t_2 \rightsquigarrow l_1 \otimes t_2'} \qquad \textsc{LOp}\ \frac{v = [\![l_1 \otimes l_2]\!]_{\mathscr{L}}}{l_1 \otimes l_2 \rightsquigarrow v}$$

Figure 5.5: Semantics for pure label operations, with binary operator $\otimes \in \{\sqcup, \sqcap, \sqsubseteq\}$. The precise definition of these operators depends on the underlying label mode $\mathscr{L}$.

Here, we simply use the *Ord* functions ($\leqslant$, *min*, and *max*), as defined by the compiler, to define the lattice operations. Of course, in real-world applications developers can define more complex label formats, such as the DLM [144], HiStar [228], Flume [104], or DC Labels [184]. Since such label definitions are typically provided by trusted code, LIO simply assumes that labels form a lattice, i.e., we do not verify that labels form a partially ordered set with a well-defined least upper bound and greatest lower bound. However, in certain cases, static analysis (e.g., in the form of refinement types [162]) can be used to verify that provided definitions are well-defined.

To model labels, we extend our calculus to make labels first-class. Instead of modeling typeclasses, for simplicity, we assume that our calculus is polymorphic in the label type $\mathscr{L}$. With this in mind, we extend the syntactic categories of Figure 5.1 as shown on the right (Figure 5.4). Here, values are extended with labels—metavariables l and c span over such values; types are extended with the label type $\mathscr{L}$; and, terms are extended with label operations.

$$\begin{aligned} &\text{Values } v ::= \cdots \mid l \mid c \\ &\text{Terms } t ::= \cdots \mid t_1 \sqcup t_2 \mid t_1 \sqcap t_2 \mid t_1 \sqsubseteq t_2 \\ &\text{Types } \tau ::= \cdots \mid \mathscr{L} \end{aligned}$$

Figure 5.4: Formal syntax for labels.

The reduction rules for these label operations are straightforward and given in Figure 5.5. The rules for the label operations $\sqcup$, $\sqcap$, and $\sqsubseteq$ rely on the label-specific implementation of these operators, as used in the premise of rule (LOp); we use the partial function $[\![\cdot]\!]_{\mathscr{L}}$, which maps terms to values, to denote this. For example, instantiating our calculus to $\mathscr{L}_2$, $[\![\mathit{Public} \sqcup \mathit{Secret}]\!]_{\mathscr{L}_2} = \mathit{Secret}$, $[\![\mathit{Secret} \sqsubseteq \mathit{Public}]\!]_{\mathscr{L}_2} = \mathit{True}$, etc. We highlight that our evaluation rules reduce the left operand first. Reducing the right operand first does not affect the semantics—we chose left-to-right evaluation solely because it matches the implementation of the labels used in λChair (see [184]). In the rest of the chapter, we sometimes use a more lax notation to describe label operations, e.g., $l_1 \sqsubseteq l_2$ in place of $l_1 \sqsubseteq l_2 \rightsquigarrow \mathit{True}$.

5.2.3 Restricting Haskell to safe IFC subset with the *LIO* monad

As previously mentioned, every object in an IFC system must be labeled. Importantly, this includes the *current execution context* whose label we call the *current label*.[4] The current label serves a role similar to the program counter (*pc*) in static IFC systems [52]. Namely, it prevents the current computation from performing side-effects which might compromise confidentiality. For instance, if the current label is l_{cur}, LIO prevents the computation from writing to entities labeled l_e unless $l_{\text{cur}} \sqsubseteq l_e$.

To accomplish this, LIO provides a monad called *LIO*. The *LIO* monad encapsulates Haskell's *IO* monad as to allow for *LIO* computations to perform (restricted) I/O. The monad also encapsulates the current label l_{cur}, which is retrieved with the *getLabel* function. The relevant parts of the definition are given below. By convention, we use $\mathscr{L}$ for type variables that are expected to be instantiated by a label. The library is polymorphic over $\mathscr{L}$ for greater flexibility, but in any normal program, every occurrence of $\mathscr{L}$ will be instantiated by the same label type. Hence, it is more intuitive to think of $\mathscr{L}$ as representing a particular (though unspecified) label type. Below we give the interface for this monad. We omit the definitions for simplicity.

$$
\begin{aligned}
&\textbf{data}\ \mathit{LIO}\ \mathscr{L}\ \tau \\
&\textbf{instance}\ \mathit{Monad}\ (\mathit{LIO}\ \mathscr{L}) \\
&\quad \mathit{return} :: \tau \to \mathit{LIO}\ \mathscr{L}\ \tau \\
&\quad (\gg\!=) :: \mathit{LIO}\ \mathscr{L}\ \tau_1 \to (\tau_1 \to \mathit{LIO}\ \mathscr{L}\ \tau_2) \to \mathit{LIO}\ \mathscr{L}\ \tau_2 \\
&\mathit{getLabel} :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{LIO}\ \mathscr{L}\ \mathscr{L}
\end{aligned}
$$

As usual, *return* lifts a value into the $\mathit{LIO}\ \mathscr{L}$ monad, while bind ($\gg\!=$) is used to chain two actions by executing the first and binding the result to be used in the executing second. The definitions for the monadic *return* and bind ($\gg\!=$) are straightforward—a reference to the current label is simply threaded through the computation. This label is exposed via *getLabel*; *getLabel* is a monadic action (in the $\mathit{LIO}\ \mathscr{L}$ monad), which, when executed, returns the current label (of type $\mathscr{L}$).

We remark that since *return* and bind are essentially the standard State monad combinators [117], no security checks are performed internally by these combinators. Instead, LIO library functions (e.g., *readFile*) use the current label to perform security checks (so as to enforce IFC) before executing any underlying *IO* actions. Taking this approach, the LIO library provides a collection of

[4] More generally, every thread in the system is labeled. But, since we are focusing on a single-threaded system, we refer to the main thread context as the current execution context and its label as the current label.

$$\textsc{Return}\quad \frac{}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{return}\ t\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{LIO}^{\text{TCB}}\ t\rangle}$$

$$\textsc{Bind}\quad \frac{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1\rangle \xrightarrow{n}{}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{LIO}^{\text{TCB}}\ t'_1\rangle}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1 \gg\!= t_2\rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t_2\ t'_1\rangle}$$

$$\textsc{LIOPure}\quad \frac{t_1 \rightsquigarrow t'_1}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid t'_1\rangle}$$

$$\textsc{GetLabel}\quad \frac{}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{getLabel}\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{return}\ l_{\text{cur}}\rangle}$$

Figure 5.7: Semantics for the *LIO* monad.

LIO actions that are similar to the *IO* actions available in standard Haskell libraries—and, indeed, usually wrap them—but additionally enforce IFC. Henceforth, we assume that all computations are in the *LIO* monad.

To formally describe the behavior of the *LIO* monad, we extend the syntactic categories of our calculus as shown on the right (Figure 5.6). Our extension simply adds monadic actions ($\mathit{LIO}^{\text{TCB}}\ t$) to values, monadic operations to terms, and a type for *LIO* computations. We note that the $\mathit{LIO}^{\text{TCB}}$ constructor is not part of the surface syntax, i.e., programs that use $\mathit{LIO}^{\text{TCB}}$ are not considered valid.[5]

Values	$v ::= \cdots \mid \mathit{LIO}^{\text{TCB}}\ t$
Terms	$t ::= \cdots \mid \mathit{return}\ t \mid t_1 \gg\!= t_2 \mid \mathit{getLabel}$
Types	$\tau ::= \cdots \mid \mathit{LIO}\ \mathscr{L}\ \tau$
Memories m	
Programs k	$::= \langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle$

Figure 5.6: Formal syntax for core *LIO*.

We explicitly distinguish pure-term evaluation from top-level monadic-term evaluation. Specifically, an LIO program is a *configuration*—spanned over by metavariable k—of the form $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle$, where l_{cur} is the current label, c_{cur} is the current clearance (explained in Section 5.3), m is the memory store (see Section 5.4), and t is the monadic term under evaluation. The reduction $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n}$

[5] For simplicity, we do not use additional syntactic categories to distinguish between values and terms that are part of the surface syntax from those that are not. In Section 5.6, we define a *safe* predicate for making such a distinction.

$\langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t' \rangle$ represents a single evaluation step from term t, with current label l_{cur}, current clearance c_{cur}, and memory m, to term t', with current label l'_{cur}, current clearance c'_{cur}, and memory m'. For the moment, we ignore the clearance and memory in the configuration. Index n in the transition relation counts the number of executed *toLabeled* actions; this is an artifact of the proof technique and not relevant to the semantics. We write $\xrightarrow{n}{}^{*}$ for the reflexive and transitive closure of $\xrightarrow{n}$. The reduction rules for the core LIO operations are given in Figure 5.7. The rules for *return* and $(\gg\!=)$ are trivial and standard—all IFC checks are performed by the non-proper morphism of *LIO*. Similarly, the (LIOPURE) rule specifies that if we have a top level pure term, it should be evaluated to completion, i.e., until it reduces to a monadic term. Rule (GETLABEL) defines the *LIO* library function for retrieving the underlying current label, further discussed below.

Coarse-grained labeling with the current label

To soundly reason about IFC, every value *must* be labeled. However, and in contrast to other language-based systems (e.g., Jif [145], FlowCaml [174], Breeze [86] etc.) in which every value is explicitly labeled, the values in our calculus are not associated with explicit labels (see Figure 5.1–5.6). This is a direct consequence of taking a library-based approach: we cannot explicitly label every Haskell value without modifying the language runtime. Instead, and like several IFC operating systems [63, 228], we take a coarse-grained approach and use the current label to protect all values in scope, i.e., in LIO, the current label l_{cur} is the label on all "unlabeled" values in the current execution context. Since we use the current label to restrict the current computation from performing arbitrary side-effects, this also ensures that the confidentiality (and integrity) of all values in scope is preserved.

In addition to ensuring that every value is labeled, this coarse-grained labeling approach has two other interesting consequences. First, it does not force developers to explicitly label every piece of data. This eliminates the need to clutter code with labels, reason about the security implications of every value, or define a special *default* label (e.g., that would be used to label literals). Instead, developers only explicitly label data they care about, as detailed in Section 5.2.4.

Second, it eliminates the implicit flows problem by construction [167]. As previously mentioned, this problem arises when information can be leaked through the program control flow. An example of an implicit flow is given in Figure 5.8, written in a hypothetical alternative LIO language without explicit labels. Here,

if b_{Secret}
  then $x_{Public} := 1_{Public}$
  else $x_{Public} := 0_{Public}$

Figure 5.8: Implicit flows.

secret bit b is leaked into public reference x according to the program control flow, i.e., what code—which assignment (to public reference x)—is executed depends on the secret b.

To prevent such leaks, language-based approaches rely on the program counter label to reflect the sensitivity of the branch condition within each branch and, in turn, disallow such unsafe assignments. In Haskell, and thus LIO, branch conditions have type *Bool*—they are not explicitly labeled values. Rather, the branch condition is (conceptually) labeled by the current label, which is common across both branches. As a consequence, control flow cannot be used to leak sensitive information: regardless of the branch taken, the current label prevents writes to public entities. Consider implementing the attack in Figure 5.8 with LIO. Since the branch condition b_{Secret} is not explicitly labeled, it is protected by l_{cur}. But since b_{Secret} is secret, we must have $l_{\text{cur}} = \mathit{Secret}$, meaning any subsequent writes (within the branches or after) to public references are disallowed since $l_{\text{cur}} \not\sqsubseteq \mathit{Public}$. In Section 5.4, we give the precise semantics for mutable references in LIO.

A floating current label

The current label protects all data in scope by serving as an upper bound label on all values. To preserve this invariant, when reading sensitive data, we can either disallow reads from entities more sensitive or raise the current label to protect the newly read data. Like other coarse-grained systems we take the latter approach and raise the current label to "float" above the labels of all the entities from which data has been read.

Raising the current label allows computations to flexibly read data, at the cost of being more limited in where they can subsequently write. Concretely, a computation with current label l_{cur} can read data labeled l_d by raising its current label to $l'_{\text{cur}} = l_{\text{cur}} \sqcup l_d$, but can thereafter only write to entities labeled l_e if $l'_{\text{cur}} \sqsubseteq l_e$. For example, LIO allows a public computation to read secret data by raising l_{cur} from *Public* to *Secret*. Importantly, the new current label prevents the computation from subsequently writing to public entities. Some static IFC systems, such as Jif [145], are even more permissive in allowing public writes after reading secret data if no secret data is actually being leaked. In Section 5.2.5, we present a method that can be used to safely restore the current label, making our dynamic IFC system equally permissive.

Ensuring all code executes in the *LIO* monad

To ensure security, all side-effecting computations must be encoded in *LIO*. LIO can only guarantee confidentiality and integrity for computations written using the LIO library; if an attacker can bind

an arbitrary *IO* action within a larger *LIO* computation, IFC can trivially be violated. Hence, the visibility of the *LIO* value constructor, i.e., the constructor used to create values of type *LIO* $\mathscr{L}$, must be limited to the LIO *trusted computing base* (TCB) so as to guarantee that "untrustworthy" (and potentially malicious) code cannot perform arbitrary I/O. In our formal mode, this amounts to not making LIO^{TCB} part of of the surface syntax.

To accomplish this, we use Safe Haskell [197]. Specifically, the module in which the *LIO* data type is defined is marked `Unsafe`, while the modules that expose IFC-enforcing *LIO* actions are marked—by us, the library providers—as `Trustworthy`. In doing so, Safe Haskell ensures that we can safely execute arbitrary, attacker-provided *LIO* actions by simply marking the top-level modules as `Safe`. Safe Haskell prevents `Safe` code from depending on `Unsafe` modules thus ensuring that the computation could only have been composed of `Trustworthy` LIO library functions or the subset of Haskell that is "safe," i.e., the part that does not contain the *LIO* value constructor or other unsafe features such as `unsafePerformIO` [197].

5.2.4 Explicitly labeling values

While LIO ensures that everything in a context is protected by the current label, for many applications it is useful to be able to handle differently-labeled data in a single scope. To motivate this, let's consider an HTTP route (e.g., `/papers/index.html`) in λChair which lists all the papers submitted by the logged-in user.

In λChair, each submitted paper is associated with a label to ensure that the paper can only be read by users that have the appropriate role (e.g., is the author or committee member). When reading a paper from the database system, the label of the *HTTP request handler* for the route, or *controller*, which is an *LIO* action, is raised to reflect the fact that sensitive data is being incorporated into the context. In doing so, LIO can ensure that a response is only sent to the user's browser—which, itself, has a label corresponding to the authenticated user—when the controller label can flow to the browser label.

Suppose that the λChair database contains two papers, as shown in Figure 5.9, submitted by Alice and Bob (neither of whom is part of the committee). When Alice wishes to see the index of all papers she submitted, the controller must read from the database only data whose labels can flow to the browser label l_{Alice}. Otherwise, the controller will reach a state in which the current label is above the browser label (e.g., $l_{\text{Alice}} \sqcup l_{\text{Bob}}$) and it will

Paper	Label
p_{Alice}	l_{Alice}
p_{Bob}	l_{Bob}

Figure 5.9: DB with two papers.

no longer be allowed to respond to the user. In language-based IFC systems [146, 174], this is typically not a concern because values returned from the database can be individually and explicitly labeled. As a result, the controller would be able to compare the label of the value retrieved from the database and the browser label, only using the retrieved value if its label flows to the browser label. In LIO, reading both values into the context would taint the controller with both l_{Alice} and l_{Bob}, preventing the overtainted controller from replying to Alice.

To avoid being overly restrictive, LIO provides *Labeled* values. A labeled value protects an arbitrary term with a strict, explicit label, irrespective of the current label. We define such values as follows.

data *Labeled* $\mathscr{L}$ τ

As before, we restrict the value constructor to the TCB. However, to allow non-TCB code to create and manipulate labeled values, we provide a safe, IFC-abiding, interface. This is particularly important since labeled values are protected by their explicit labels—untrusted code should not be allowed to bypass the label and arbitrarily inspect (or modify) the protected value. This interface for creating and inspecting labeled values is given below.

$$\begin{aligned}
&\mathit{label} &&:: \mathit{Label}\ \mathscr{L} \Rightarrow \mathscr{L} \to \tau \to \mathit{LIO}\ \mathscr{L}\ (\mathit{Labeled}\ \mathscr{L}\ \tau)\\
&\mathit{unlabel} &&:: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{Labeled}\ \mathscr{L}\ \tau \to \mathit{LIO}\ \mathscr{L}\ \tau\\
&\mathit{labelOf} &&:: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{Labeled}\ \mathscr{L}\ \tau \to \mathscr{L}
\end{aligned}$$

To describe the semantics of these functions, we extend the values, terms and types of our calculus as shown in Figure 5.11. (As with $\mathit{LIO}^{\text{TCB}}$, we do not consider the $\mathit{Labeled}^{\text{TCB}}$ constructor part of the surface syntax.) The reduction rules for the new terms are given in Figure 5.10; rule (LABELCTX), (UNLABELCTX), and (LABELOFCTX) reduce terms until they have appropriate structures to trigger rules (LABEL), (UNLABEL), and (LABELOF), respectively. We ignore parts of these rules that involve the current clearance c_{cur} until Section 5.3.

$$\begin{aligned}
\text{Values } v &::= \cdots \mid \mathit{Labeled}^{\text{TCB}}\ v\ t\\
\text{Terms } t &::= \cdots \mid \mathit{label}\ t_1\ t_2 \mid \mathit{unlabel}\ t \mid \mathit{labelOf}\ t\\
\text{Types } \tau &::= \cdots \mid \mathit{Labeled}\ \mathscr{L}\ t
\end{aligned}$$

Figure 5.11: Formal syntax for labeled values in LIO.

The *label* function is used to explicitly label terms. The function takes two arguments, a label and a term, and returns an *LIO* action, which, when executed, produces an explicitly labeled value. Rule (LABEL) gives the precise semantics: the function associates the supplied label l with term t

$$\textsc{LabelCtx}\quad \frac{t_1 \rightsquigarrow t_1'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{label}\ t_1\ t_2\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{label}\ t_1'\ t_2\rangle}$$

$$\textsc{Label}\quad \frac{l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{label}\ l\ t\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{return}\ (\mathit{Labeled}^{\mathsf{TCB}}\ l\ t)\rangle}$$

$$\textsc{UnlabelCtx}\quad \frac{t \rightsquigarrow t'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{unlabel}\ t\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{unlabel}\ t'\rangle}$$

$$\textsc{Unlabel}\quad \frac{l_{\text{cur}} \sqcup l \rightsquigarrow l'_{\text{cur}} \qquad l'_{\text{cur}} \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{unlabel}\ (\mathit{Labeled}^{\mathsf{TCB}}\ l\ t)\rangle \xrightarrow{0} \langle l'_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{return}\ t\rangle}$$

$$\textsc{LabelOfCtx}\quad \frac{t \rightsquigarrow t'}{\mathit{labelOf}\ t \rightsquigarrow \mathit{labelOf}\ t'} \qquad \textsc{LabelOf}\quad \frac{}{\mathit{labelOf}\ (\mathit{Labeled}^{\mathsf{TCB}}\ l\ t) \rightsquigarrow l}$$

Figure 5.10: Semantics for labeled values in LIO.

by wrapping the term with the $\mathit{Labeled}^{\mathsf{TCB}}$ constructor. It first asserts that the new label (l) used to protect t is at least as restricting as the old label (the current label, l_{cur}), i.e., $l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True}$.

We remark that if the premise does not hold the function throws an exception to indicate an IFC violation—our semantics do not employ stop-the-world semantics as a way to encode monitor failures. This is the case for all other rules in LIO in which a premise is not satisfied. Section 5.5 describes this in more detail and defines exception handling facilities that code can use to recover from such IFC violations.

The dual of *label*, *unlabel*, takes an explicitly labeled value and returns an *LIO* action which, when executed, returns the underlying wrapped value. As given by rule (UNLABEL), the function takes a labeled value $\mathit{Labeled}^{\mathsf{TCB}}\ l\ t$ and returns the wrapped term t. However, since the returned term is no longer protected by l and is, instead, protected by the current label, l_{cur} must be at least as restricting as l[6]. To ensure this, the current label is raised from l_{cur} to $l_{\text{cur}} \sqcup l$—this captures the

[6]The effects of *unlabel* are similar to those of *bind* in DCC [1]: subsequent computations must be protected by the label of the recently observed value.

fact that the remaining computation might depend on t. The current label always "floats" above the labels of the values observed by the current computation.

Finally, we provide the *labelOf* function as a way to inspect the label of a labeled value. As detailed by the (LABELOF) reduction rule, *labelOf* takes a labeled value $Labeled^{TCB}\ l\ t$ and simply returns the label l protecting term t. Since the label of a *Labeled* value is strict, *labelOf* does not require an additional context rule for reducing the label. Unlike *unlabel*, *labelOf* also does not raise the current label—*labelOf* is part of the pure calculus. Indeed, this allows code to check the label of a labeled value before deciding to *unlabel* it (and thereby raise the current label). This design decision has an important consequence: regardless of the current label (and clearance) of the configuration, *labelOf* always succeeds. While this may seem like LIO labels are "public," they are in fact protected by a label—the current label—and thus cannot be used as a covert channel. Section 5.2.5 describes an alternative design in which labels are not public and shows how labels can be used to leak information when not properly protected.

Example 1 (Fetching papers for reviewers)**.** Turning to our λChair use case, we now consider some of the core functions that are used by the top-level request handler. In particular, we show how to fetch papers for a given reviewer using a simple underlying database system. The specific label type used by λChair is *DCLabel*. As defined in [184], a *DCLabel* is a pair of formulae over principals (e.g., users) in conjunctive normal form, representing the principals that can read and write data labeled as such. We define a type alias for the *LIO* monad with the label instantiated to *DCLabel*:

$$\textbf{type}\ DC\ \tau = LIO\ DCLabel\ \tau$$

The λChair database system operates on *DCLabel*ed papers, in the *DC* monad. As defined below, a paper is simply a record with several fields, including the (unique) paper id (*paperId*), the paper itself (*pdf*), labeled *reviews*, etc.[7]

$$\textbf{data}\ Paper = Paper\ \{paperId :: Id, pdf :: PDF, reviews :: [LabeledReview], \ldots\}$$
$$\textbf{type}\ LabeledPaper = Labeled\ DCLabel\ Paper$$

Among other operations, the database system provides a *fetchPapers* function which is used to get the list of *all* such papers:

[7] We elide the details of labeled reviews used in the actual λChair implementation and simplify some of the application details (e.g., the generic database system API). The interested reader is referred to the code documentation at `http://labeled.io` for more details.

$$fetchPapers :: DC\ [LabeledPaper]$$

For simplicity, we omit the implementation details of *fetchPapers* and only remark that it relies on TCB code to wrap an underlying *IO*-based database system API and explicitly label the fetched papers.

While simple, the *fetchPapers* function is sufficient for fetching a given reviewer's papers. Note that if the controller simply unlabels the papers returned by *fetchPapers*, the current label may be raised to a point where the computation cannot respond back to the user, i.e., the current label may not flow to the browser label. This situation, for example, happens when the current user is not part of the committee and another author's paper is unlabeled—λChair prevents such data from being sent (leaked) back to the user's browser. Hence, we need to make sure that the controller only reads data that the end-user can see.

To this end, we define *fetchPapersFor*:

```
fetchPapersFor :: User → DC [LabeledPaper]
fetchPapersFor user = do
  -- Get all labeled papers:
  lpapers ← fetchPapers
  -- Filter the papers the user is allowed to read:
  let browserLabel = userToLabel user
    lpapers' = filter (λlpaper.labelOf lpaper ⊑ browserLabel) lpapers
  -- Unlabel and return all the papers this user can read:
  mapM unlabel lpapers'
```

This function fetches the papers, filters the ones the user is allowed to read by comparing the paper's label with the user's browser label—itself computed with function *userToLabel*—and unlabels them. At this point, the controller can compose the HTML page containing the paper information and safely respond to the user.

In addition to providing a simple illustration of how labeled values are used in LIO, this simple example serves to illustrate the importance of labeled values. Specifically, by providing labeled values in the language, we can implement core functionality such as *fetchPapersFor* in the untrusted LIO application code; without labeled values such functionality would otherwise have to be implemented in the trusted database layer or database system itself. Indeed, building on this observation, we can, for example, extend λChair to implement an in-memory database which solely uses the

aforementioned database system as a persistence layer, i.e., it solely relies on the actual database system to keep the papers persistent.

5.2.5 Addressing label creep with *toLabeled*

In conference systems, it is often the case that some reviews are superseded by others, papers change titles, submissions are withdrawn, etc. Hence, the λChair database system provides functions for updating (or deleting) existing papers. For instance, *updatePaper* is used to update the paper with the supplied paper id with the new labeled paper. The type for this function is given below.

$$updatePaper :: Id \to LabeledPaper \to DC\ ()$$

Similar to *fetchPapers*, this function relies on TCB code to communicate with the actual database system; from a security stance, it is only interesting to note that the function always ensures that the current computation can overwrite the existing paper (by performing a $\sqsubseteq$-check with the current label, current clearance (see Section 5.3), and label protecting the existing paper).

Suppose we wish to implement a function that performs a partial update, i.e., an update wherein only part of the paper object is updated. This is useful, for example, when a user only updates the abstract of the paper and leaves other parts such as the underlying PDF intact. Indeed, sending a PDF file, which may be large, to simply perform a "full" update is not practical. An implementation of such a partial update function is given below.

```
partialUpdatePaper :: Id → PartialPaper → DC ()
partialUpdatePaper i new = do
  -- Get the existing paper according to its id:
  lold ← fetchPaperById i
  old ← unlabel lold
  -- Merge the new (partial) paper and existing paper:
  lnew ← label (labelOf lold) (merge new old)
  -- Perform actual update:
  updatePaper i lnew
```

Here, we assume that the type *PartialPaper* encodes a partial paper (e.g., by using a *Maybe* type for each of the fields in *Paper*) and function *merge* simply merges the content of the new partial paper and existing paper. The underlying *fetchPaperById* database function behaves as expected: it

returns the labeled paper corresponding to the id.[8]

Unfortunately, this implementation has the drawback of always raising the current label to the label of the paper being updated. This can result in a scenario where actions that follow a partial update fail (e.g., writes to less sensitive entities), solely because the current label is overly restricting. Raising the current label to a point where the computation can no longer perform certain useful side-effects is known as *label creep* [167]. Label creep does not compromise security, since the current label still protects all data in lexical scope. But, it hinders functionality. In the *partialUpdatePaper* example, label creep is particularly unappealing since *partialUpdatePaper* does not return any information about the existing paper—it simply writes back to the database. Ideally, we should be able to implement the *partialUpdatePaper* computation that operates on sensitive data, but avoid raising the current label and thus label creep.

In general, being able to perform computations on sensitive data without raising the current label is crucial to building practical applications. To this end, LIO provides the *toLabeled* function which can be used to execute an *LIO* action and subsequently restores the current context label. The type signature for this function is:

$$toLabeled :: Label\ \mathscr{L} \Rightarrow \mathscr{L} \rightarrow LIO\ \mathscr{L}\ \tau \rightarrow LIO\ \mathscr{L}\ (Labeled\ \mathscr{L}\ \tau)$$

The function takes a label l (the upper bound, describe below) and the *LIO* term t that computes on sensitive data. Intuitively, if the current label at the point where $toLabeled\ l\ t$ gets executed is l_{cur}, *toLabeled* executes t and restores the current label to l_{cur}, i.e., *toLabeled* provides a separate context in which t is evaluated. Of course, returning the result of t directly would allow for trivial leaks of sensitive data. Hence, *toLabeled* labels the result of t with l. This design decision effectively states that the result of t is protected by label l, as opposed to the current label at the point t completed. Of course, *toLabeled* requires that the result of t not be more sensitive than l.

To formally describe the semantics of *toLabeled*, we extend terms with the *toLabeled* primitive: $t ::= \cdots \mid toLabeled\ t_1\ t_2$ and give two new reduction rules in Figure 5.12. In both rules, the current label and clearance are preserved. Rule (TOLABELEDCTX) simply reduces the label argument. Rule (TOLABELED) specifies the non-trivial case. As noted above, the label l is used to label the result of t. Hence, the rule first ensures that we are not trying to create a labeled value below the current label (or above the current clearance, see Section 5.3), i.e., $l_{\text{cur}} \sqsubseteq l \rightsquigarrow True$. The rule then

[8] Note that this has the implication that id's are effectively public. However, since the number of elements in the database is public (as revealed by the length of the list returned by *fetchPapers*), this is not surprising.

$$\textsc{toLabeledCtx}\quad \frac{t_1 \rightsquigarrow t_1'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{toLabeled}\ t_1\ t_2\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{toLabeled}\ t_1'\ t_2\rangle}$$

$$\textsc{toLabeled}\quad \frac{l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True} \qquad \langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n\,*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{LIO}^{\mathsf{TCB}}\ t'\rangle \qquad l'_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{toLabeled}\ l\ t\rangle \xrightarrow{n+1} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid \mathit{return}\ (\mathit{Labeled}^{\mathsf{TCB}}\ l\ t')\rangle}$$

Figure 5.12: Semantics for the LIO *toLabeled* construct.

completely reduces t to an *LIO* value.[9] If the current label l'_{cur} at the time of completion is below the provided upper bound l, then "transferring protection" of the result t' from l'_{cur} to l is safe and we thus simply return the result, labeled with l. Observe that if $l'_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{False}$, then labeling the result t' with l might result in a leak, e.g., if t' actually contains information above l. In Section 5.5, we consider the cases where these conditions do not hold. We finally remark that the (TOLABELED) increments the index n to indicate that *toLabeled* was executed. This decoration is used to simplify the proof burden and is further explained in Section 5.6.

Example 2 (Partially updating papers)**.** Returning to our partial update λChair example, we can now use *toLabeled* in a straightforward way to implement *partialUpdatePaper*. This new implementation is given below.

```
partialUpdatePaper :: Id → PartialPaper → DC ()
partialUpdatePaper i new = do
  -- Get the existing paper according to its id:
  lold ← fetchPaperById i
  lnew ← toLabeled (labelof lold) (do
    old ← unlabel lold
    -- Merge the new (partial) paper and existing paper:
    return (merge new old))
  -- Perform actual update:
```

[9] By using big-step semantics, we do not need to rely on the use of trusted functions that (save and) restore the current label and clearance.

updatePaper i lnew

This implementation is almost identical to the original one. It only differs in wrapping the part of the code that is computing on sensitive data with *toLabeled*. Specifically, it wraps the part of the code that unlabels the existing paper and performs the merge. (Since *toLabeled* returns a labeled value, we no longer need to explicitly *label* the merged paper—we simply return it.) The current label within the *toLabeled* blocks is raised to the join of the current label and the label of the existing paper (*labelOf lold*) by function *unlabel*. Importantly, however, the current label before and after calling *partialUpdatePaper* remains the same.

An alternative semantics for *toLabeled*

Naturally, one may ask why *toLabeled* demands that we provide the label of the result as an argument, as opposed to simply using the final current label of the executed computation. Indeed, an early version of LIO had such an implementation. The reduction rule for this alternative function

$$toLabeled' :: Label\ \mathscr{L} \Rightarrow LIO\ \mathscr{L}\ \tau \rightarrow LIO\ \mathscr{L}\ (Labeled\ \mathscr{L}\ \tau)$$

is given below.

$$\textsc{toLabeled'}\ \frac{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n} {}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid LIO^{\text{TCB}}\ t'\rangle}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid toLabeled'\ t\rangle \xrightarrow{n+1} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid return\ (Labeled^{\text{TCB}}\ l'_{\text{cur}}\ t')\rangle}$$

But, different from the version of LIO as presented in this chapter, inspecting the label of labeled values with *labelOf* must raise the current label to the join of the current label and label of the value. The semantics for this alternative function

$$labelOf' :: Label\ \mathscr{L} \Rightarrow Labeled\ \mathscr{L}\ \tau \rightarrow LIO\ \mathscr{L}\ \mathscr{L}$$

is given below.

$$\textsc{labelOf'}\ \frac{l_{\text{cur}} \sqcup l \rightsquigarrow l'_{\text{cur}} \qquad l'_{\text{cur}} \sqsubseteq c_{\text{cur}}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid labelOf'\ (Labeled^{\text{TCB}}\ l\ t)\rangle \xrightarrow{0} \langle l'_{\text{cur}}, c_{\text{cur}}, m \mid return\ l\rangle}$$

This difference is particularly important since information can otherwise be leaked by encoding it into the labels themselves [166, 35]. To illustrate this point, consider the 3-point lattice $\mathscr{L}_3 =$

$\{Public, Secret, TopSecret\}$ and the following code that uses *toLabeled′* and *labelOf* to leak the value of a secret Boolean.

```
leakBool :: Labeled L3 Bool → LIO L3 Bool
leakBool secretBool = do
  -- Current label is Public
  secretBool′ ← toLabeled′ (do
                    s ← unlabel secretBool    -- Raise current label to Secret
                    -- Raise label to TopSecret if s is True
                    when s (raiseLabel TopSecret))
  -- Current label is Public
  return (labelOf secretBool′ ≡ TopSecret)
    where raiseLabel l = label l () >>= unlabel
```

The key distinction between the two designs is *what label is used to protect the label of a labeled value* [35]. (Recall that in an IFC system every piece of data must be labeled—this include labels themselves.) In the early version of LIO (that with *toLabeled′* and *labelOf′*) the label on the label of a value was the label itself. Hence, inspecting the label of a value required raising the current label. Importantly, however, *toLabeled′* did not require programmers to supply an upper bound label for the labeled result. In contrast, the current version of LIO considers the current label l_{cur} as the label protecting the labels of labeled values. In this system, inspecting the label of a value does not require raising the current label, and *labelOf* is, in turn, pure. Of course, the trade-off is that the label on the result produced by *toLabeled* must be provided a-priori.

Our experience with building λChair and other larger-scale applications has shown that the ability to inspect labels outweighs the "burden" of specifying an upper bound for *toLabeled*. The interested reader is referred to [72] for a description of an example system built on top of LIO. In fairness, most of the systems and applications we built on top of LIO are web-centric and while we believe this experience to extend to other domains, evaluating this trade-off for other kinds of applications is an interesting direction for future work.

5.3 Addressing covert channels with clearance

IFC systems do not typically restrict what data code can read, rather—and as we have done thus far—they only restrict where the code can write to once it has read the data. Similarly, code can

always write to channels or create objects with arbitrary labels, as long as doing so does not leak information, i.e., code can always write to and allocate entities more sensitive than the current label. But, in many cases it is useful to execute code with *least privilege* by limiting its access to the data/entities it needs to perform its task [170]. This principle not only simplifies security auditing, but, as shown in this section, it also eliminates the opportunity for code to leak sensitive data by exploiting covert channels [107]. LIO introduces the notion of *clearance* to language-based IFC systems [186], later adopted by Breeze [86], as a means for restricting access to certain labeled entities. Clearance in LIO can be seen as a particular discretionary access control mechanism (DAC) integrated into a IFC system, where DAC security checks are performed before their IFC counterparts [190].

5.3.1 Restricting data-access with clearance

The current clearance c_{cur} is a label tracked by the *LIO* monad alongside the current label l_{cur}; in our formalization, the clearance appears as the second component of a program configuration $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t \rangle$. LIO restricts access to certain labeled entities using the clearance in two different ways.

First, the clearance is used to restrict the reading of overly-sensitive data by enforcing that the current clearance always be an upper bound on the current label, i.e., for all valid program configurations $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t \rangle$, it is the case that $l_{\text{cur}} \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}$. This restriction is enforced by the LIO interface. For example, *unlabel* as given in rule (UNLABEL) of Figure 5.10 only unlabels the labeled value if raising the current label l_{cur} will not result in a current label l'_{cur} that is above the current clearance, i.e., $l'_{\text{cur}} \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}$. In a similar way, before reading from a file or reference (see Section 5.4), we ensure that raising the current label will not violate this guarantee.

The use of clearance to restrict code from reading certain entities is a form of discretionary access control; we can prevent malicious code from exploiting covert channels to leak overly-sensitive information by ensuring that it cannot read such data. As an example, suppose that the partial update function in λChair is implemented by a third-party developer (e.g., to implement a better merging function). If the developer is malicious, they can use the partial update function to leak the contents of a competing author's paper through covert channels. Indeed, this is simple since the developer can create an account on the λChair platform and take on the role of an author to ensure that their malicious code is executed. A malicious version of *partialUpdatePaper* is given below.

$$\mathit{leakyPartialUpdatePaper} :: \mathit{Id} \to \mathit{PartialPaper} \to \mathit{DC}\ ()$$

```
leakyPartialUpdatePaper i new = do
  -- Get all existing papers:
  papers ← fetchPaperById i
  -- Leak information about some of the papers
  mapM maybeLeak papers
  -- Execute the normal partial update:
  partialUpdatePaper i new
    where maybeLeak lpaper = toLabeled (labelOf lpaper) (do
            paper ← unlabel lpaper
            -- If the paper has a specific author, leak it:
            when (paperAuthors paper ≡ ...) (leakToCovertChannel paper))
```

Here, we use function *leakToCovertChannel* to leak information about papers written by certain authors; otherwise the function behaves in the same way as the normal *partialUpdatePaper* code. The function *leakToCovertChannel* leaks (part of) the sensitive paper content through a covert channel. For instance, the code can leak information by diverging (or not) according to the paper content, i.e., one bit at a time through the termination covert channel [7]; alternatively, it can leverage the external timing covert channel [3] to leak the information by delaying the response according to the content, etc. Using clearance, we can prevent such leaks by setting the clearance to the label of the browser—in this case, the *leakyPartialUpdate* will fail to *unlabel* papers which the requesting user, i.e., the attacker, is not allowed to read. Since the code running on behalf of one user does not have access to another user's data, it cannot leak it—the code can only leak data it can already read.

The second role of clearance is to restrict code from writing to and allocating entities labeled above the clearance. For example, *label* as given in rule (LABEL) of Figure 5.10 only creates a *Labeled* value if the label of the value is bounded by the clearance. Similarly, *toLabeled* as given in rule (TOLABELED) of Figure 5.12 requires the upper bound of the result to be below the clearance. In a similar way, before creating or writing to a file or reference (see Section 5.4), we ensure that their label is below the current clearance. As in [228], this addresses attacks in which malicious code duplicates sensitive data, e.g., by copying a file, only to read it later, when the system policy changes (e.g., in λChair, promoting a member to a co-chair and granting them the corresponding privileges). While, within a single run, LIO programs can use robust declassification as in [224, 208] to reason about policy changes, without clearance, reasoning about the consequence of a system policy change across multiple program runs is more difficult. We refer the intersted reader to [228] for a more detailed consideration of this use case.

5.3.2 Making clearance first-class

To leverage clearance for isolation, as described above, we execute a term in a configuration that has initially set the desired clearance. Of course, in many applications it is useful to be able to "drop" privileges and continue executing with least privilege [170]. For example, in λChair when authenticating user requests, the clearance must be high enough to read credentials, but once the authentication is complete, having access to such information is unnecessary and dangerous: a simple bug in the code that generates an HTML list of the user's papers could potentially leak the credentials. Hence, we provide a means for inspecting and manipulating the clearance. Specifically, we provide:

$$\begin{aligned} &\mathit{getClearance} &&:: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{LIO}\ \mathscr{L}\ \mathscr{L} \\ &\mathit{lowerClearance} &&:: \mathit{Label}\ \mathscr{L} \Rightarrow \mathscr{L} \to \mathit{LIO}\ \mathscr{L}\ () \end{aligned}$$

The *getClearance* and *lowerClearance* functions are used to get and set the current clearance, respectively.

$$\textsc{GetClearance}\quad \frac{}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{getClearance}\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{return}\ c_{\text{cur}}\rangle}$$

$$\textsc{LowerClearanceCtx}\quad \frac{t \rightsquigarrow t'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{lowerClearance}\ t\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{lowerClearance}\ t'\rangle}$$

$$\textsc{LowerClearance}\quad \frac{l_{\text{cur}} \sqsubseteq c'_{\text{cur}} \rightsquigarrow \mathit{True} \qquad c'_{\text{cur}} \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{lowerClearance}\ c'_{\text{cur}}\rangle \xrightarrow{0} \langle l_{\text{cur}}, c'_{\text{cur}}, m \mid \mathit{return}\ ()\rangle}$$

Figure 5.13: Semantics for clearance related terms in LIO.

We add the primitives *getClearence* and *lowerClearance* to the syntactic category of terms $t ::= \cdots \mid \mathit{getClearance} \mid \mathit{lowerClearance}\ t$ and formally describe its semantics in Figure 5.13. The rules are mostly self-explanatory. We solely highlight that the premise in rule (LOWERCLEARANCE) requires the new current clearance c'_{cur} to be below the current clearance c_{cur} and above the current label. By lowering the clearance, code can effectively run with least privilege. Of course, allowing code to arbitrarily raise the clearance would trivially prevent us from confining untrusted code—hence code can only decide to access fewer entities.

However, recall from rule (TOLABELED) that *toLabeled* restores the current label and clearance. Hence, combined with *toLabeled*, we can use *lowerClearance* to execute a term t, at a lower clearance, without lowering the current clearance:

$$\begin{aligned}&\mathit{withClearance} :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathscr{L} \to \mathit{LIO}\ \mathscr{L}\ \tau \to \mathit{LIO}\ \mathscr{L}\ (\mathit{Labeled}\ \mathscr{L}\ \tau)\\&\mathit{withClearance}\ c'_{\text{cur}}\ t = \mathit{toLabeled}\ c'_{\text{cur}}\ (\mathit{lowerClearnce}\ c'_{\text{cur}} \gg t)\end{aligned}$$

This use of *toLabeled* addresses the dual to the label creep described in Section 5.2.5: by lowering the current clearance a program can reach a state where $l_{\text{cur}} = c_{\text{cur}}$, at which point it cannot read or write to entities more-sensitive than l_{cur}. More interestingly, this enables powerful security patterns. For instance, it allows arbitrary untrusted code to treat code it depends on as untrustworthy. Indeed, this primitive can be used to address the poison pill attacks described in [86], wherein untrusted libraries carry out denial of service attacks via label creep. Additionally, *withClearance* can be used to structure programs in such a way that different components execute with least privilege and are isolated from one another. For example, in λChair, we can wrap request handlers with *withClearance* to isolate requests based on the user (browser) label. This is similarly done in the Hails web framework, when serving HTTP requests and accessing database tables, which themselves have a notion of clearance for the labels on stored data [72].

5.4 Mutable labeled references

Many practical applications rely on imperative data-structures, often implemented using mutable reference. In the context of λChair mutable references can, for example, be used to implement an efficient in-memory database. Indeed, by modeling each paper as a labeled reference, instead of a labeled immutable value, updating a paper becomes very cheap; it simply amounts to writing to a reference, as opposed to creating a large immutable data structure (that contains the rest of the papers).

Unsurprisingly, LIO provides labeled alternatives to Haskell's *IORef*s [151]. The LIO reference API is given below.

$$\begin{aligned}&\textbf{data}\ \mathit{LIORef}\ \mathscr{L}\ \tau\\&\mathit{newLIORef}\ \ :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathscr{L} \to \tau \to \mathit{LIO}\ \mathscr{L}\ (\mathit{LIORef}\ \mathscr{L}\ \tau)\\&\mathit{readLIORef}\ \ :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{LIORef}\ \mathscr{L}\ \tau \to \mathit{LIO}\ \mathscr{L}\ \tau\\&\mathit{writeLIORef} :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{LIORef}\ \mathscr{L}\ \tau \to \tau \to \mathit{LIO}\ \mathscr{L}\ ()\end{aligned}$$

While the implementation of secure references can vary, we simply wrap Haskell's *IORef*s. Intentionally, this API resembles the standard Haskell API for mutable references. The key difference is that the function for creating references takes an additional argument: the label of the reference.

To formally describe this API, we extend our calculus with references as shown in Figure 5.14. Like $Labeled^{\mathsf{TCB}}$, the $LIORef^{\mathsf{TCB}}$ constructor is restricted to the TCB and is strict in its first argument. References are created with *newLIORef*, read with function *readLIORef*, and modified with *writeLIORef*. We overload the *labelOf* function to allow code to inspect the label of a reference.[10]

$$
\begin{array}{lll}
\text{Values } v & ::= & \cdots \mid LIORef^{\mathsf{TCB}}\ v\ a \\
\text{Terms } t & ::= & \cdots \mid newLIORef\ t_1\ t_2 \mid readLIORef\ t \\
 & & \mid writeLIORef\ t_1\ t_2 \\
\text{Types } \tau & ::= & \cdots \mid LIORef\ \mathscr{L}\ \tau
\end{array}
$$

Figure 5.14: Formal syntax for references in LIO.

The reference store—spanned over by metavariable m—is a map from addresses—spanned over by metavariable a—to labeled values.[11] Since we do not provide any mechanisms for explicit deallocation or address inspection/comparison in the LIO API we can model the store as an infinitely-large map, while allowing the implementation to safely garbage collect unused references as necessary.[12] In our formalization, this memory store appears as the third component of a program configuration $\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t \rangle$.

The reduction rules for references are given in Figure5.15. When creating a reference, as given by rule (NEWLIOREF), *newLIORef l t* creates a labeled value that guards t with label l and stores it in the memory store at a new, fresh, address a. Subsequently, the function returns an *LIORef* value that contains the reference label and the address where the term is stored. (Like $Labeled^{\mathsf{TCB}}$, the constructor $LIORef^{\mathsf{TCB}}$ is not part of the surface syntax and thus cannot be abused by untrusted code.) Rule (READLIOREF) specifies the semantics for reading a labeled reference; reading the term stored at address a simply amounts to unlabeling the value $m(a)$ stored at the underlying

[10] In our implementation, we use a typeclass *LabelOf* to define the *labelOf* function. Both *LIORef* and *Labeled* are instances of this class.

[11] Since the label of a reference accompanies the address (both wrapped by the $LIORef^{\mathsf{TCB}}$ constructor), an alternative memory store that simply maps addresses to terms is sufficient—we chose the labeled-store approach to simplify the proof burden (see Section 5.6).

[12] Non-opaque pointers could potentially be used to leak information (e.g., by freeing a reference in a secret context only to allocate a reference and inspect its address in a public context). Adapting LIO to deal with non-opaque pointers can be done as in [80].

$$\textsc{newLIORefCtx}\ \frac{t_1 \rightsquigarrow t_1'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{newLIORef}\ t_1\ t_2\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{newLIORef}\ t_1'\ t_2\rangle}$$

$$\textsc{newLIORef}\ \frac{l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True} \qquad \text{fresh}(a) \qquad m' = m[a \mapsto \mathit{Labeled}^{\text{TCB}}\ l\ t]}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{newLIORef}\ l\ t\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid \mathit{return}\ (\mathit{LIORef}^{\text{TCB}}\ l\ a)\rangle}$$

$$\textsc{readLIORefCtx}\ \frac{t \rightsquigarrow t'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{readLIORef}\ t\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{readLIORef}\ t'\rangle}$$

$$\textsc{readLIORef}\ \frac{v = m(a)}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{readLIORef}\ (\mathit{LIORef}^{\text{TCB}}\ l\ a)\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{unlabel}\ v\rangle}$$

$$\textsc{writeLIORefCtx}\ \frac{t_1 \rightsquigarrow t_1'}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{writeLIORef}\ t_1\ t_2\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{writeLIORef}\ t_1'\ t_2\rangle}$$

$$\textsc{writeLIORef}\ \frac{l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True} \qquad m' = m[a \mapsto \mathit{Labeled}^{\text{TCB}}\ l\ t]}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{writeLIORef}\ (\mathit{LIORef}^{\text{TCB}}\ l\ a)\ t\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid \mathit{return}\ ()\rangle}$$

$$\textsc{labelOfLIORef}\ \frac{}{\mathit{labelOf}\ (\mathit{LIORef}^{\text{TCB}}\ l\ a) \rightsquigarrow l}$$

Figure 5.15: Semantics for monadic LIO terms related to references.

address. Function *writeLIORef*, specified by rule (WRITELIOREF), updates the memory store with a new labeled term t for the reference at location a, leaving the label intact. Note that in the latter three rules, we impose the restriction that the label of the reference l must be bound by the current label and clearance, i.e., $l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True}$ and $l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}$. This ensures that we both preserve the confidentiality of data in scope and avoid reading/modifying entities above the clearance. It is worth remarking that when one considers the current label l_{cur} as the dynamic version of the *pc*, our restriction that the label of the reference be above the current label ($l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True}$) when writing to the reference is similar to the one imposed by other IFC λ-calculi [227, 11]. The rule *labelOf*, given by (LABELOFLIOREF), is self-explanatory and we do not discuss it further.

5.5 Exception handling

Like references, exceptional control flow is common in real-world applications. As already noted, LIO provides support for throwing and catching exceptions. Code can throw an exception using the *throwLIO* function and catch exceptions using *catchLIO*:

$$\mathit{throwLIO} :: (\mathit{Exception}\ e, \mathit{Label}\ \mathscr{L}) \Rightarrow e \rightarrow \mathit{LIO}\ \mathscr{L}\ \tau$$
$$\mathit{catchLIO} :: (\mathit{Exception}\ e, \mathit{Label}\ \mathscr{L}) \Rightarrow \mathit{LIO}\ \mathscr{L}\ \tau \rightarrow (e \rightarrow \mathit{LIO}\ \mathscr{L}\ \tau) \rightarrow \mathit{LIO}\ \mathscr{L}\ \tau$$

This API is identical to that of standard Haskell, except that it operates in the *LIO* monad. Moreover, the semantics for these functions are standard.[13] Nevertheless, we must consider the implication on security when they are used in concert with other LIO library functions—in particular, *toLabeled*.

In Figure 5.16, we formally extend values with exceptions ξ and a new *LIO* constructor ($\mathit{LIO}_X^{\mathsf{TCB}}$), terms with the exception handling functions (*throwLIO* and *catchLIO*), and types with *Exception*s. For simplicity, we only consider a single exception type.

$$\begin{aligned}
\text{Values } v &::= \cdots \mid \xi \mid \mathit{LIO}_X^{\mathsf{TCB}}\ t \\
\text{Terms } t &::= \cdots \mid \mathit{throwLIO}\ t \mid \mathit{catchLIO}\ t_1\ t_2 \\
\text{Types } \tau &::= \cdots \mid \mathit{Exception}
\end{aligned}$$

Figure 5.16: Formal syntax for exceptions in LIO.

Figure 5.17 gives the exception-related reduction rules. Function *throwLIO*, as given by rule (THROWLIO), raises an exception by simply lifting the exception term t into the *LIO* monad with

[13] This is in contrast with the original semantics of exceptions as presented in [188], where an explicit label was associated with every thrown exception. In comparison to the treatment of exceptions in [188] and [86], the approach described in this chapter is considerably simpler.

$$\textsc{throwLIO}\quad \frac{}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{throwLIO}\ t\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{LIO}_X^{\text{TCB}}\ t\rangle}$$

$$\textsc{bindEx}\quad \frac{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1\rangle \xrightarrow{n}{}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{LIO}_X^{\text{TCB}}\ t'_1\rangle}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1 \gg\!= t_2\rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{throwLIO}\ t'_1\rangle}$$

$$\textsc{catch}\quad \frac{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1\rangle \xrightarrow{n}{}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{LIO}^{\text{TCB}}\ t'_1\rangle}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{catchLIO}\ t_1\ t_2\rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{return}\ t'_1\rangle}$$

$$\textsc{catchEx}\quad \frac{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t_1\rangle \xrightarrow{n}{}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{LIO}_X^{\text{TCB}}\ t'_1\rangle}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{catchLIO}\ t_1\ t_2\rangle \xrightarrow{n} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid t_2\ t'_1\rangle}$$

Figure 5.17: Semantics for exceptions without *toLabeled*. The remaining changes are given in Figure 5.18.

constructor $\mathit{LIO}_X^{\text{TCB}}$. Indeed, the role of the $\mathit{LIO}_X^{\text{TCB}}$ constructor is to distinguish between exceptional and non-exceptional monadic control flow. Building on this, we add a new reduction rule for bind ($\gg\!=$) that propagates exceptions; as shown by the (BINDEX) rule, bind re-throws the exception if the term under evaluation reduced to an exceptional monadic term ($\mathit{LIO}_X^{\text{TCB}}\ t$). (We explicitly define the (BINDEX) in terms of *throwLIO* to more closely match our Haskell implementation.) Otherwise, it behaves as before, according to rule (BIND).

The semantics for *catchLIO* is also straightforward. Since throwing an exception depends on the information present in the lexical scope, *catchLIO* must retain the current label to reflect this fact; observe that all the *catchLIO* reduction rules in Figure 5.17 leave the context intact. Rule (CATCH) specifies the case where the term does not raise an exception and reduces to a "normal" *LIO* value. Here, the value is simply returned. Rule (CATCHEX) specifies the case where the term raises an exception. In this case, the exception handler t_2 is applied to the exception t_1. We note that our semantics are lazy in the exception value, much in the same way as Haskell; neither *throwLIO* nor *catchLIO* force the evaluation of the exception.

The reduction rules of Figure 5.17 take the standard approach of *propagating exceptions up the call stack until the nearest enclosing catchLIO*. Though necessary, this is not sufficient; without modifying the semantics of *toLabeled*, exceptions can be used to leak information. Consider the

following function:

```
condThrow :: Labeled L2 Bool → LIO L2 ()
condThrow secretBool = do
  s ← unlabel secretBool
  when s (throwLIO ξ)
```

Suppose that *condThrow* is invoked with the current label *Public* and *secretBool* has label *Secret*. Then, *throwLIO* raises exception ξ if the secret is *True*; if the secret is *False condThrow* simply returns $()$. This function alone cannot be used to leak the secret, since the current label at the end of *condThrow* is *Secret*. But, by wrapping *condThrow* with *toLabeled*, we can avoid raising the current label when the secret is *False* and thus leak the value into a public reference:

```
leakSecret :: Labeled L2 Bool → LIO L2 Bool
leakSecret secretBool = do
  -- Create public reference:
  publicRef ← newLIORef Public True
  toLabeled Secret (catchLIO (do
    toLabeled Secret (condThrow secretBool)
    writeLIORef publicRef False    -- Write only if no exception is thrown
    )(λ_ → return ()))
  -- Read direct leak of secret:
  readLIORef publicRef
```

Assume that this function is invoked with a *Public* current label. First, the function creates a public reference *publicRef* initialized to *True*. Then, if the secret is *True*, the exception thrown by *condThrow* escapes the innermost *toLabeled* block up to the *catchLIO*, which invokes the handler. At this point the current label is *Secret*, since *condThrow* raised the label to read the secret. However, the outer *toLabeled* restores the current label to *Public*. This allows us to read the *publicRef*, which is still *True*. By contrast, if the secret is *False*, *condThrow* simply returns $()$; the enclosing *toLabeled* ensures that the current label remains *Public*. At this point, we write *False* into the public reference. Finally, we again read and return the reference contents. In both cases the returned value corresponds to the secret boolean.

This code illustrates that the standard propagation of exceptions up the call stack until reaching the nearest enclosing *catchLIO* is not sufficient. LIO must only propagate exceptions up to the

$$\textsc{toLabeledEx}\quad \frac{l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True} \qquad \langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid \mathit{LIO}_X^{\mathsf{TCB}}\ t'\rangle \qquad l'_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{toLabeled}\ l\ t\rangle \xrightarrow{n+1} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid \mathit{return}\ (\mathit{Labeled}_X^{\mathsf{TCB}}\ l\ t')\rangle}$$

$$\textsc{unlabelEx}\quad \frac{l_{\text{cur}} \sqcup l \rightsquigarrow l'_{\text{cur}} \qquad l'_{\text{cur}} \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{unlabel}\ (\mathit{Labeled}_X^{\mathsf{TCB}}\ l\ t)\rangle \xrightarrow{0} \langle l'_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{throwLIO}\ t\rangle}$$

$$\textsc{labelOf2}\quad \frac{}{\mathit{labelOf}\ (\mathit{Labeled}_X^{\mathsf{TCB}}\ l\ t) \rightsquigarrow l}$$

Figure 5.18: Semantics for terms affected by exceptions in LIO.

nearest *catchLIO or toLabeled*. Intuitively, the correct semantics for *toLabeled* are as before with the added requirement that all exceptions be caught by it: regardless of how the computation enclosed by *toLabeled* terminates—with an exception or value—a *Labeled* value must always be returned. In other words, we adapt the semantics of some LIO actions (including *toLabeled*) to secure the exception handling mechanism provided by *throwLIO* and *catchLIO*.

Formally, we extend values with another *Labeled* constructor $v ::= \cdots \mid \mathit{Labeled}_X^{\mathsf{TCB}}\ v\ t$, that encodes the fact that t is an exception. The additional rule for *toLabeled* is given by (TOLABELEDEX) in Figure 5.18: if term t raises an exception (that is not caught) $\mathit{LIO}_X^{\mathsf{TCB}}\ t'$, we wrap the exception by the new *Labeled* constructor. When unlabeling such a labeled value, as given by (UNLABELEX), LIO simply propagates the exception. Of course, *unlabel* raises the current label, ensuring that information from the point of the throw cannot be leaked. Finally, (LABELOF2) gives the additional rule for *labelOf*, which allows programs to inspect the label of *Labeled* values wrapping exceptions. Note that we do not allow code to distinguish between $\mathit{Labeled}^{\mathsf{TCB}}$ and $\mathit{Labeled}_X^{\mathsf{TCB}}$; doing so would allow for trivial leaks.

With these modifications in place, we highlight that the actions in *leakSecret* following the *toLabeled* block will always be executed, even if an exception is raised inside *condThrow*. Intuitively, we close the leak due to exception propagation by simply assuring that the execution of (possibly public) actions following a *toLabeled* block does not depend on the abnormal termination of a computation wrapped by *toLabeled*. In a similar manner, but using concurrent threads, we can address leaks due to the timing and non-termination behavior of the enclosed computation [183].

We remark that closing leaks due to exception propagation, as such, is not without cost. In particular, "delaying" exceptions raised within *toLabeled* blocks raises two challenges. First, developers need to handle exceptions at the point of *unlabel*ing data, even though the exception was potentially raised in a different part of the program. This imposes a somewhat nonstandard, asynchronous programming model which closely resembles promises [69, 130]. We have found that, in general, debugging IFC programs is non-trivial for average developers [72].

To address this, our LIO implementation associates a stack-trace like data-structure with exceptions. Internally, LIO defines an annotation function which is used in the rest of the library:

$$\mathit{withContext} :: \mathit{String} \to \mathit{LIO}\ \mathscr{L}\ \tau \to \mathit{LIO}\ \mathscr{L}\ \tau$$

This function takes a string message (typically the name of the function) and the action to execute, and returns an action that wraps the original action with *catchLIO*. The catch is used to interpose any thrown monitor failure exceptions as to add the annotation message before rethrowing it. Consider the following program:

$$\mathit{withClearance}\ \mathit{lAliceOrBob}\ (\mathit{label}\ \mathit{lAlice}\ 42)$$

Here, the program starts with an initial current label and clearance set to *lPublic*, where $\mathit{lPublic} \sqsubseteq \mathit{lAliceOfBob} \sqsubseteq \mathit{lAlice}$, but neither relations flow hold in the reverse direction. This program throws an exception because it attempts to create a labeled value above the current clearance (within the *withClearance* block). In particular, it produces the following error message:

$$\begin{aligned}
&\mathit{LabelError}\ \{\\
&\quad \mathit{lerrContext} = [\texttt{"withClearance"}, \texttt{"label"}],\\
&\quad \mathit{lerrFailure} = \texttt{"guardAllocP"},\\
&\quad \mathit{lerrCurLabel} = \mathit{lPublic},\\
&\quad \mathit{lerrCurClearance} = \mathit{lAliceOrBob},\\
&\quad \mathit{lerrPrivs} = [\,],\\
&\quad \mathit{lerrLabels} = [\mathit{lAlice}]\\
&\}
\end{aligned}$$

Note that the error message contains a lot of useful information:

- A stack-trace like context of the functions called before the program terminated.
- The actual point of failure; in this case an internal function within *label* called *guardAllocP*, which performs the actual $\sqsubseteq$-check before creating a labeled value.

- The current label and clearance when the exception was thrown.
- The privileges supplied to the action that threw the exception.
- The labels supplied to the action that thew the exception.

While an actual stack trace would be more useful, this information has proved very useful in practice when building our Hails web framework and applications on top of it;[14] particularly because developers can use *withContext* to annotate their own constructs. We remark that in an imperative language, debugging could be simplified even further.

The second issue with delaying exceptions is that it may lead to scenarios in which exceptions go unnoticed. Consider, for example, executing a sensitive computation with the sole interest of performing a side-effect (e.g., a write to the database). Since, the result of the computation is of no interest, we are likely to never *unlabel* the result and, as a result, overlook a failure—*toLabeled* catches all exceptions.

Concretely, suppose we attempt to update a paper stored in the database with a value of type *LabeledPaper*, which was produced as a result of a *toLabeled* computation. (Our *partialUpdatePaper* is an example of one such computation.) Further suppose that the *toLabeled* computation read data more sensitive than its bound, which should be the paper label. In such a case we would write an exceptional value to the database, which will only be observed by the user on a follow-up read. While this is not an issue from a security stance, it is likely not the desired or expected behavior; the computation should not delay the exception and instead reply to the user with an error.

While, in practice, users can also use *label* to create labeled values that contain pure exceptions (e.g., using Haskell's *throw*), an alternative strict label type (e.g., *StrictLabeled*) can ensure that such labeled values never contain exceptions. Given this, an alternative *toLabeled* definition could simply return a labeled variant (see Section 5.5.1), i.e., a value of type $\mathit{StrictLabeled}\ \mathscr{L}\ (\mathit{Either}\ \mathit{Exception}\ \tau)$. While this alternative API would not prevent code from ignoring the result (and thus, the errors), it would prevent developers from overlooking exceptions raised in a *toLabeled* blocks when they try to reuse the resultant values (e.g., to insert them into the database).

In practice, we found that using clearance to restrict what a computation can read and write within a *toLabeled* block and having to provide an upper bound label to *toLabeled* (and the fact that one can freely inspect labels) help with reasoning about and preventing IFC monitor failures

[14] In debugging mode, it is possible to get more accurate information by rewriting the Haskell source to wrap at every bind, and also add file and line number annotations. We do not do this in production because of performance.

a-priori. But, of course, other failures (e.g., network connection failures) are less predictable and in such cases we cannot avoid inspecting the return values to catch any delayed exceptions. In such cases, LIO's support for declassification, though not discussed in this chapter, was used to "safely leak" the success/failure of a sensitive computation. In general, we did not find delayed exceptions to be a hindrance. However, our experience comes from building Hails [72] and applications on top of Hails, which build on the concurrent version of LIO that uses threads in place of *toLabeled*; in these applications, we mostly relied on *toLabeled*-like construct to execute code in which failure was easy to predict (e.g., transformers from strings to abstract data types). Lastly, we refer the reader to the work of [86] for a more exhaustive discussion on the various design points of delayed exceptions.

5.5.1 Recovering from monitor failures

Our reduction rules given thus far in Figure 5.2–5.18 do not consider cases where label checks fail. Like for other dynamic IFC systems (e.g., [9, 168, 11, 12, 56]), this would imply aborting the program execution when a monitor failure occurs. For practical systems, this approach is not appropriate: we cannot halt the system when a λChair request handler is about to violate IFC. Moreover, it is not safe—this introduces a covert channel [144].

As we previously mentioned, LIO and Breeze [86] differ from most other dynamic IFC systems in using exceptions to encode monitor failures. For example, when the security conditions in rule (UNLABEL) are not met, we throw an exception:

UNLABELFAIL

$$\frac{l_{\text{cur}} \sqcup l \rightsquigarrow l'_{\text{cur}} \qquad l'_{\text{cur}} \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{False}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{unlabel}\ (\mathit{Labeled}^{\mathsf{TCB}}\ l\ t)\rangle \xrightarrow{0} \langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{throwLIO}\ \xi_{IFC}\rangle}$$

Here, ξ_{IFC} is simply an exception containing information about the failure. In the same way, we provide reduction rules dual to those of Figure 5.10–5.18 that simply throw exceptions when a security condition is not met. We do not discuss these rules further since they are straightforward. The only interesting case is a particular failure of *toLabeled*, given below.

TOLABELEDFAIL

$$\frac{\begin{array}{c} l_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{True} \qquad l \sqsubseteq c_{\text{cur}} \rightsquigarrow \mathit{True} \\ \langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle \xrightarrow{n\,*} \langle l'_{\text{cur}}, c'_{\text{cur}}, m' \mid v\rangle \qquad l'_{\text{cur}} \sqsubseteq l \rightsquigarrow \mathit{False} \end{array}}{\langle l_{\text{cur}}, c_{\text{cur}}, m \mid \mathit{toLabeled}\ l\ t\rangle \xrightarrow{n+1} \langle l_{\text{cur}}, c_{\text{cur}}, m' \mid \mathit{return}\ (\mathit{Labeled}^{\mathsf{TCB}}_{X}\ l\ \xi_{IFC})\rangle}$$

Here, the enclosed term t raises the current label l'_{cur} above the upper bound l. By simply throwing

an exception we would potentially be leaking information about data more sensitive than l_{cur}. (Malicious code can "throw" an exception by raising the current label above the upper bound imposed by *toLabeled*, reintroducing the attack from the previous section.) As mentioned before, *toLabeled* must return a labeled value. Therefore, we return a labeled value that contains an exception that encodes the monitor failure; at the point of *unlabel*, this "delayed" exception is raised.

By encoding monitor failures with exceptions, as opposed to stopping the program, LIO allows untrusted code to catch exceptions and safely recover from attempted IFC violations. Consider, for instance, the following function that unlabels a *Labeled* value and returns an *Either* value to indicate the success or failure of the operation:

$$
\begin{aligned}
&\mathit{safeUnlabel} :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathit{Labeled}\ \mathscr{L}\ \tau \rightarrow \mathit{LIO}\ \mathscr{L}\ (\mathit{Either}\ \mathit{Exception}\ \tau)\\
&\mathit{safeUnlabel}\ \mathit{lv} = \mathit{catchLIO}\ (\mathbf{do}\ v \leftarrow \mathit{unlabel}\ \mathit{lv}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{return}\ (\mathit{Right}\ v)\\
&\qquad\qquad\qquad\qquad)\ (\lambda e \rightarrow \mathit{return}\ (\mathit{Left}\ e))
\end{aligned}
$$

If the label of *lv* is above the current clearance or if the value is a labeled exception, the LIO *unlabel* throws an exception (raising the label in the latter case), which is handled by simply returning the exception wrapped with the *Left* constructor. If no exception is raised, the current label is raised and the unlabeled result is returned, wrapped by *Right*. As discussed in [86], this is generally a very useful feature since it treats code in an egalitarian fashion, and allows one to integrate untrusted code in an application without having to worry that the code will halt the system by causing a monitor failure.

We remark that, unlike our original treatment of exceptions [188], the (TOLABELEDFAIL) rule treats normal and exceptional results of a failed *toLabeled* block the same. This means that if a computation within a *toLabeled* block raised its current label above the bound and terminated with an exception, the exception will be hidden. (Though, a non-exceptional value would be hidden too.) As for Breeze's $\lambda^{\langle\rangle}_{\mathit{throw}+\mathbf{D}}$ calculus, this means that delayed exceptions are isomorphic to labeled tagged variants, i.e., values of type $\mathit{Labeled}\ \mathscr{L}\ (\mathit{Either}\ \mathit{Exception}\ \tau)$. The trade-off between these semantics and our original ones are explored in detail in [86]. The downside of our current approach is clear: error message are hidden, thus making it more difficult to debug LIO programs.[15] However, this trade-off comes with a benefit: *all* exceptions, including delayed exceptions, can be caught. (After all, when unlabeled, delayed exceptions, are isomorphic to tagged variants.)

[15] We remark that this can be improved by keeping track of the precise point within the *toLabeled* block that the current label was raised above the bound and adding this to the exception stack-trace discussed above.

This is not necessarily true of our original calculus. To understand the difference, suppose an exception is raised in a *toLabeled* block with an upper bound set to l; further suppose that the current label when exception is raised is l', where $l' \not\sqsubseteq l$. Since exceptions are not hidden (in our original calculus), when *unlabel*ing such delayed exceptions, the *unlabel* primitive re-threw the exception, raising the current label l_{cur} to $l_{\text{cur}} \sqcup l \sqcup l'$. Unfortunately, wrapping *unlabel* with a *catchLIO* does not guaranteed that the exception will be caught—in particular, if the l' is not below clearance, *catchLIO* would simply propagate the exception. At a high level, this effectively means that code cannot unlabel values from an untrusted computation without risking a *poison pill attack* [86], i.e., attacks wherein untrusted code running in a *toLabeled* block render outer computations useless by raising the current label above the *expected* label of the labeled value. Of course, code can always use *withClearance* to avoid such attacks, but this approach is less usable.

5.6 Security guarantees

In this section, we show that programs written in LIO satisfy noninterference and a form of discretionary access control. Informally, noninterference states that secret values cannot be leaked by LIO programs, while DAC ensures that computations cannot bypass the restrictions imposed by clearance to access or create arbitrary data. Before delving into the details of these security guarantees, we first highlight some notational difference with the previous sections and describe the extent of our mechanization in Coq.

Notation To allow for incremental introduction of concepts, in the previous section we used LIO^{TCB} and LIO_X^{TCB} constructors to respectively denote non-exceptional and exceptional monadic *LIO* terms that have been executed to the point of containing no more side effects. In this section, we use a single constructor that additionally takes a boolean argument to indicate whether the value is an exception or not: term $LIO_b^{\mathsf{TCB}}\ t$ corresponds to $LIO^{\mathsf{TCB}}\ t$ if $b = \mathbf{true}$ and $LIO_X^{\mathsf{TCB}}\ t$ if $b = \mathbf{false}$. Similarly, we use $Labeled_b^{\mathsf{TCB}}$, with $b \in \{\mathbf{true}, \mathbf{false}\}$, instead of the $Labeled^{\mathsf{TCB}}$ and $Labeled_X^{\mathsf{TCB}}$ constructors.

Mechanized proofs We formalized a large subset of the calculus, described in Section 5.2, using the Coq theorem prover. The mechanized subset omits references and the reduction rules corresponding to monitor failures described in Section 5.5.1. Moreover, the Coq implementation uses a concrete four-point lattice similar to that shown in Figure 5.3. For this subset, we mechanized the

propositions, lemmas, theorems, and proofs given below; we distinguish the non-mechanized parts of the proofs with the symbol ✎. We leave the extension to the full calculus with an abstract lattice to future work.

5.6.1 Noninterference

In this section, we prove that LIO satisfies noninterference using the *term erasure* technique from [116, 164]. Intuitively, the term erasure technique allows us to show that a program satisfies noninterference by showing that the behavior of the program with all the sensitive data (classified above l) "erased" cannot be distinguished by an attacker (at observation level l) from the behavior of the original program.

To model such programs, we extend our calculus and reduction rules with erased terms, denoted by a new terminal $\bullet$, as follows:

$$t ::= \cdots \mid \bullet \qquad \vdash \bullet : \tau \qquad \text{HOLE}\ \bullet \rightsquigarrow \bullet \qquad \text{HOLELIO}\ \langle \bullet, \bullet, \bullet \mid \bullet \rangle \xrightarrow{n} \langle \bullet, \bullet, \bullet \mid \bullet \rangle$$

Intuitively, an erased term can have any type. Moreover, an erased term or configuration, the latter represented by $\langle \bullet, \bullet, \bullet \mid \bullet \rangle$, always reduces to itself. We use a meta-level *erasure function* $\varepsilon_l(\cdot)$ to replace all terms more sensitive than the attacker's observation level l with $\bullet$. To an attacker, terms and configurations above their observation level appear as $\bullet$; the new reduction rules also ensure that no information can be learned from the reduction of such terms (by effectively diverging).

Figure 5.19 gives the definition of the erasure function for values, terms, memories, and configurations. For most values, the erasure function is simply the identity function, since most values are not heterogeneously labeled. Similarly, for most terms, the function is simply applied homomorphically (e.g., $\varepsilon_l($ **if** *True* **then** t_2 **else** $t_3) =$ **if** *True* **then** $\varepsilon_l(t_2)$ **else** $\varepsilon_l(t_3)$). There are only four interesting cases. First, when erasing a $\mathit{Labeled}_b^{\text{TCB}}\ l_1\ t_2$ value, we erase the term t_2 protected by label l_1 to $\bullet$ when the label does not flow to l; otherwise we simply apply the function homomorphically. Second, we aggressively erase values that are about to be labeled with *label*. While the erasure function only erases values when the first argument to *label* is a value (and not a term), we define a new reduction relation that applies the erasure function at every step and thus ensure that values are erased as soon as possible. We note that such aggressive erasure would not be correct for *toLabeled*, which also returns a labeled value, since *toLabeled* takes a monadic *LIO* action that may produce side-effects observable to the attacker. Third, we erase a whole configuration to $\langle \bullet, \bullet, \bullet \mid \bullet \rangle$

when the current label is not below l; this ensures that the attacker cannot observe anything about sensitive configurations. Fourth, we erase all reference more sensitive than the attacker observation label, even those created in public contexts. This ensures the attacker cannot observe anything about the sensitive parts of the memory store.

$$\varepsilon_l(\mathit{True}) = \mathit{True} \qquad \varepsilon_l(\mathit{False}) = \mathit{False} \qquad \varepsilon_l(()) = () \qquad \varepsilon_l(l_1) = l_1$$

$$\varepsilon_l(\mathit{Labeled}_b^{\mathsf{TCB}}\ l_1\ t) = \begin{cases} \mathit{Labeled}_b^{\mathsf{TCB}}\ l_1\ \varepsilon_l(t) & l_1 \sqsubseteq l \\ \mathit{Labeled}_b^{\mathsf{TCB}}\ l_1\ \bullet & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\mathit{label}\ l_1\ t_2) = \begin{cases} \mathit{label}\ l_1\ \varepsilon_l(t_2) & l_1 \sqsubseteq l \\ \mathit{label}\ l_1\ \bullet & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\langle l_{\text{cur}}, c_{\text{cur}}, m \mid t\rangle) = \begin{cases} \langle l_{\text{cur}}, c_{\text{cur}}, \varepsilon_l(m) \mid \varepsilon_l(t)\rangle & l_{\text{cur}} \sqsubseteq l \\ \langle \bullet, \bullet, \bullet \mid \bullet\rangle & \text{otherwise} \end{cases}$$

$$\varepsilon_l(m) = \{(a, \varepsilon_l(m(a))) : a \in \mathrm{dom}(m) \text{ and } \mathit{labelOf}\ m(a) \sqsubseteq l\} \qquad \varepsilon_l(\bullet) = \bullet$$

Figure 5.19: Erasure function for values, terms, configurations, and memory store. For all other terms, the erasure function is simply applied homomorphically..

The addition of $\bullet$ and corresponding reduction rules completes our calculus and semantics definition. We now prove several general properties for this calculus, followed by two key properties needed for the noninterference theorem: simulation and determinacy of our monadic reduction relation and a new relation that erases sensitive terms.

Our first lemma states that values are in normal form, i.e., values do not reduce.

Lemma 2 (Values do not reduce)**.**

- *For any value v, there is no term t such that $v \rightsquigarrow t$.*
- *For any $l_{cur}, c_{cur}, m, v, n$, there is no program configuration k such that $\langle l_{cur}, c_{cur}, m \mid v\rangle \xrightarrow{n} k$.*

Proof. The first case follows by induction on the pure term reduction relation. The second case follows by induction on the structure of v. □

Though straightforward, this lemma is helpful when distinguishing terms that terminate since, as in most sequential IFC calculi, our noninterference guarantee is termination insensitive, i.e., it only holds for terminating terms. And, recall that our calculus allows non-terminating terms with **fix**.

The next proposition show that the erasure function is homomorphic over substitution and idempotent over terms, memories and configurations.

Proposition 4 (Idempotence and distribution properties of the erasure function)**.** *The erasure function is:*

1. *Idempotent over terms:* $\varepsilon_l(t) = \varepsilon_l(\varepsilon_l(t))$
2. *Idempotent over memory*✎*:* $\varepsilon_l(m) = \varepsilon_l(\varepsilon_l(m))$
3. *Idempotent over configurations:* $\varepsilon_l(k) = \varepsilon_l(\varepsilon_l(k))$
4. *Homomorphic over substitution:* $\varepsilon_l(\{t_1 \,/\, x\}\, t_2) = \{\varepsilon_l(t_1) \,/\, x\}\, \varepsilon_l(t_2)$

Intuitively, the first three properties respectively state that multiple application of the erasure function does not affect the term, memory, or configuration once it has been erased. In other words, the erasure function should completely erase sensitive data encoded in a term.

The erasure function additionally distributes over the pure reduction relation.

Proposition 5 (Erasure function distributes over the pure-term reduction relation)**.** *For any label l, if* $t \rightsquigarrow t'$ *then* $\varepsilon_l(t) \rightsquigarrow \varepsilon_l(t')$.

Proof. Straightforward induction on t, using Lemma 2, and Proposition 4. □

In other words, taking a step in the pure reduction and erasing the end term is the same as first erasing the term and taking a step. Intuitively this is stating that sensitive data does not affect the reduction of a pure term.

We now extend this intuition to simulation with a new reduction relation under which sensitive terms and configurations are erased. This new monadic-term reduction relation with erasure is defined as follows:

Definition 6 (Reduction of pure and monadic terms with erasure)**.**

$$\frac{k \xrightarrow{n} k'}{k \xrightarrow{n}_l \varepsilon_l(k')}$$

Configurations under this relation are evaluated in the same way as before, with the exception that, after one evaluation step, the erasure function is applied to the resulting configuration. In this

manner, the relation guarantees that confidential data, i.e., data above level l, is erased as soon as it is created.

To illustrate the need for this relation, consider two labels l_1 and l_2, such that $l_1 \sqsubseteq l_2$, and the following program $p = \langle l_1, l_2, \emptyset \mid (\lambda l.\mathit{label}\ l\ 42)\ l_2 \rangle$. Assuming an attacker at observation level l_1, program p contains the secret 42, which is placed inside a *label* expression when β-reducing. Observe that $\varepsilon_{l_1}(p)$ is not enough to capture what an attacker should see, since it still contains the secret, i.e., $\varepsilon_{l_1}(p) = \langle l_1, l_2, \emptyset \mid (\lambda l.\mathit{label}\ l\ 42)\ l_2 \rangle$. However, observe that $p \xrightarrow{n}_{l_1} \langle l_1, l_2, \emptyset \mid \mathit{label}\ l_2\ \bullet \rangle$ erases the secret (42) as soon as it is β-reduced—capturing the attacker observational power at every reduction step of the program.

Figure 5.20 highlights the intuition behind our simulation result: erasing all sensitive data, i.e., data whose label is not below l, and then taking a step in $\xrightarrow{n}_l$ is the same as taking a step in $\xrightarrow{n}$ and then erasing all the secret values in the resulting configuration. Observe that if configuration k leaks data labeled above l (such that it is observable at l), then erasing all sensitive data and taking a step in $\xrightarrow{n}_l$ might not be the same as taking steps in $\xrightarrow{n}$ and then erasing all the secret values in the resulting configuration—the data might have already been leaked. We remark that, while this simulation result and several statements below involve configurations that are initially erased, we rely on the more general reduction relation for determinacy and prove the more general statement where appropriate.

$$\begin{array}{ccc} k & \xrightarrow{\ n\ } & k' \\ \vdots\,{\scriptstyle \varepsilon_l} & & \vdots\,{\scriptstyle \varepsilon_l} \\ \varepsilon_l(k) & \xrightarrow{\ n\ }_l & \varepsilon_l(k') \end{array}$$

Figure 5.20: Simulation between the normal and erased relation.

First, we show that the current label after taking a step is always at least as restricting as the current label before taking the step.

Proposition 6 (Monotonicity of the current label)**.** *If* $\langle l_{cur}, c_{cur}, m \mid t \rangle \xrightarrow{n} \langle l'_{cur}, c'_{cur}, m' \mid t' \rangle$ *then* $l_{cur} \sqsubseteq l'_{cur}$.

Proof. Straightforward induction on t, using the lattice-properties of labels (namely, reflexivity of $\sqsubseteq$ and definition of $\sqcup$). □

This proposition not only reduces the number of cases we need to consider, but also reinforces our intuition that none of the LIO terms can lower the current label once sensitive data is incorporated in the context (and thereby allow for such data to be leaked). We note that since *toLabeled* is defined using big-step semantics it does not actually restore the current label of the context; rather it executes a term in a separate context in a single step.

We now prove simulation of the monadic-term reduction relation. The proof follows by induction on the number of executed *toLabeled* blocks, i.e., index n on the $\xrightarrow{n}$ relation. These cases are further broken down into several simpler cases, according to the observational level of the attacker and current labels (before and after taking a step). To simplify presentation, these supporting statements are given in Appendix A.

Lemma 3 (Single-step simulation without *toLabeled*)**.** *If* $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{0} \langle l'_{cur}, c'_{cur}, m' \mid t'\rangle$ *then* $\varepsilon_l(\langle l_{cur}, c_{cur}, m \mid t\rangle) \xrightarrow{0}_l \varepsilon_l(\langle l'_{cur}, c'_{cur}, m' \mid t'\rangle)$.

Proof. Straightforward case analysis on $l_{\text{cur}} \sqsubseteq l$ and $l'_{\text{cur}} \sqsubseteq l$. All cases follow directly from supporting Propositions 13, 14, and 15 given in Appendix A. □

This base-case simulation corresponds to the scenario where no *toLabeled* blocks are executed. The single-step simulation lemma for arbitrary terms follows by induction, using this lemma for the base case.

Lemma 4 (Single-step simulation)**.** *If* $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n} \langle l'_{cur}, c'_{cur}, m' \mid t'\rangle$ *then it must be that* $\varepsilon_l(\langle l_{cur}, c_{cur}, m \mid t\rangle) \xrightarrow{n}_l \varepsilon_l(\langle l'_{cur}, c'_{cur}, m' \mid t'\rangle)$.

Proof. Straightforward case analysis on $l_{\text{cur}} \sqsubseteq l$ and $l'_{\text{cur}} \sqsubseteq l$ using Lemma 3 for the base case. The cases follow directly from the supporting propositions—Propositions 14, 15, and 17—given in Appendix A. □

This lemma shows a simulation between a term taking a step in the normal reduction relation and that same term, with all sensitive information erased, taking a step in the reduction relation with erasure. This is highlighted by Figure 5.20. Unfortunately, the statement is overly restricting—it imposes the number of *toLabeled* blocks to be the n. (Indeed, we are only able to prove this lemma because the reduction rule (HOLELIO) is defined for any index.)

A more general statement would allow for the number of *toLabeled* blocks to differ. In particular, when considering erasure the number of *toLabeled* blocks executed is at most n, since the erasure collapses all sensitive paths (an erased configuration reduces to itself) and thus the number *toLabeled* blocks executed in a sensitive context need not be counted. This statement is given below:

Corollary 1 (Single-step collapsed simulation)**.** *If* $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n} \langle l'_{cur}, c'_{cur}, m' \mid t'\rangle$ *then for some* $n' \leqslant n$, $\varepsilon_l(\langle l_{cur}, c_{cur}, m \mid t\rangle) \xrightarrow{n'}_l \varepsilon_l(\langle l'_{cur}, c'_{cur}, m' \mid t'\rangle)$.

Proof. Directly from Lemma 4 using n as a witness. □

We remark that while we directly use Lemma 4, this is not necessary. Indeed, one can prove a more precise bound by showing that n' corresponds to the number of *toLabeled* blocks executed in attacker-observable contexts, i.e., contexts that have a current label below the attacker observation level.

Having established the simulation between the standard reduction relation and the relation with erasure, we now solely need to show that the latter relation is deterministic to prove noninterference.

First, we show that the pure-term reduction relation is deterministic.

Proposition 7 (Determinacy of pure-term reduction)**.** *If $t \rightsquigarrow t'$ and $t \rightsquigarrow t''$ then $t' = t''$.*

Proof. By induction on the pure-term reduction relation, using Lemma 2. □

Since several reduction rules for the monadic-term reduction relation are given in using big-step semantics, we show that the big-step relation, i.e., relation wherein the end-terms are values, is deterministic:

Proposition 8 (Determinacy of big-step monadic-term reduction)**.** *If both $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{cur}, c'_{cur}, m' \mid LIO^{TCB}_{b'}\ t'\rangle$ and $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n'}{}^{*} \langle l''_{cur}, c''_{cur}, m'' \mid LIO^{TCB}_{b''}\ t''\rangle$, then $l'_{cur} = l''_{cur}$, $c'_{cur} = c''_{cur}$, $m' = m''$, $n = n'$, $t' = t''$, and $b' = b''$.*

Proof. By induction on t. Most cases follow by inversion of the first multi-step monadic-term reduction hypothesis. The *LIO*, *return*, and *throwLIO* cases further require the inversion of the second hypothesis. □

This proposition is crucial to the noninterference theorem. Indeed, it can serve as a first sanity-check when extending the library with new primitives: adding *LIO* actions that are non-deterministic, such as *getTimeOfDay* would trivially break this statement. And, extending the system to consider a non-deterministic reduction relation is non-trivial. Indeed, it may require changing even the security condition [225, 167].

We now use these two propositions to show that the single-step monadic-term reduction relation is deterministic.

Proposition 9 (Determinacy of monadic-term reduction)**.** *If $k \xrightarrow{n} k'$ and $k \xrightarrow{n'} k''$ then $k' = k''$ and $n = n'$.*

Proof. By induction on the monadic-term reduction relation, using Proposition 7 and Lemma 2. We use Proposition 8 for the (BIND), (BINDEX), (TOLABELED), (TOLABELEDEX), (CATCH-LIO), and (CATCHLIOEX) cases. □

From this, the determinacy of the relation with erasure follows in a straightforward way:

Lemma 5 (Determinacy of monadic-term reduction with erasure)**.** *For any label l, configurations k, k', and k'', and index numbers n and n', if $k \xrightarrow{n}_l k'$ and $k \xrightarrow{n'}_l k''$ then $k' = k''$ and $n = n'$.*

Proof. By inversion of the hypotheses, using Proposition 9. □

Before stating the noninterference theorem, we first define a *safe* function ς to distinguish terms that are only composed of surface syntax. Figure 5.21 gives the definition of this function for values, memories and configurations. For terms, we define ς as the conjunction of its application to all the term components. Since the definition of ς is straightforward, we only remark that our definition for memories is permissive in treating a non-empty memory m as safe when m only contains safe terms.

$$\varsigma\,(\mathit{True}) = \mathbf{true} \qquad \varsigma\,(\mathit{False}) = \mathbf{true} \qquad \varsigma\,(()) = \mathbf{true} \qquad \varsigma\,(l_1) = \mathbf{true} \qquad \varsigma\,(\lambda x.t) = \varsigma\,(t)$$

$$\varsigma\,(\mathit{LIO}_b^{\mathsf{TCB}}\ t) = \mathbf{false} \qquad \varsigma\,(\mathit{Labeled}_b^{\mathsf{TCB}}\ l\ t) = \mathbf{false} \qquad \varsigma\,(\mathit{LIORef}^{\mathsf{TCB}}\ l\ t) = \mathbf{false} \qquad \varsigma\,(\xi) = \mathbf{true}$$

$$\varsigma\,(\bullet) = \mathbf{false} \qquad \varsigma\,(x) = \mathbf{true} \qquad \varsigma\,(m) = \bigwedge_{(a,\mathit{Labeled}_b^{\mathsf{TCB}}\ t)\in m} \varsigma\,(t)$$

$$\varsigma\,(\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid t\rangle) = \varsigma\,(l_{\mathrm{cur}}) \wedge \varsigma\,(c_{\mathrm{cur}}) \wedge \varsigma\,(m) \wedge \varsigma\,(t)$$

Figure 5.21: Safe function for values, memories, and configurations. The safe function for terms is defined homomorphically over the structure of the term.

As in previous works on noninterference, we state noninterference as the preservation of l-equivalence, defined according to a syntactic equivalence relation $\approx_l$.[16] We define this l-equivalence relation as the equivalence kernel of the erasure function $\varepsilon_l(\cdot)$ for configurations. That is, $k \approx_l k'$ iff $\varepsilon_l(k) = \varepsilon_l(k')$. Note that this equivalence relation precisely captures the power of an attacker: to an attacker at observation level l, two terms that are l-equivalent cannot be distinguished.

Theorem 1 (Noninterference)**.** *For any label l, index n_1, and two configuration k_1 and k_2, such that $\varsigma\,(k_1)$ and $\varsigma\,(k_2)$, there exists an index n_2, such that if $k_1 \approx_l k_2$, $k_1 \xrightarrow{n_1} k_1'$ and $k_2 \xrightarrow{n_2} k_2'$ then $k_1' \approx_l k_2'$.*

[16] While considering syntactic l-equivalence is standard, a treatment of semantic l-equivalence would be an interesting research direction.

Proof. Applying Corollary 1 to the two hypotheses, we have: $\varepsilon_l(k_1) \xrightarrow{n_1'}_l \varepsilon_l(k_1')$, for $n_1' \leqslant n_1$ and $\varepsilon_l(k_2) \xrightarrow{n_2'}_l \varepsilon_l(k_2')$, for $n_2' \leqslant n_2$. From $k_1 \approx_l k_2$ and the definition of $\approx_l$ we have $\varepsilon_l(k_1) = \varepsilon_l(k_2)$. Then, by Lemma 5, we have $\varepsilon_l(k_1') = \varepsilon_l(k_2')$ and $n_1' = n_2'$. From the definition of l-equivalence, this is the same as $k_1' \approx_l k_2'$. Our Coq proof uses types to eliminate degenerate cases, but this is not fundamental to the proof and we thus elide this detail. □

The theorem states that if two configurations with possibly secret information, but indistinguishable to an attacker at level l, take a step, then the resulting configurations are also indistinguishable to the attacker. In other words, the attacker does not learn any sensitive information by observing configurations at lower sensitivity levels. Note, however, that the number of *toLabeled* actions executed in each step may differ according to data the attacker cannot observe—we assume that the attacker cannot observe the index counts.

This noninterference statement is stronger than that considered in the original conference paper [186], which is stated in terms of a big-step. Specifically, this statement says that no information is leaked at any intermediate step, as opposed to solely stating that the result of two l-equivalent programs do not leak information. However, as in the conference version, this is a termination-insensitive result, i.e., we only make claims about the case where the configurations can each take a step and thus leaks due to non-termination are not captured. In [183], we modify LIO to ensure that no information about the termination of sensitive subcomputation is visible to public contexts. For that, we force the execution of each *toLabeled* block to occur in a separate thread. As described in Chapter 6, the concurrent version of LIO satisfies a much stronger property—termination-sensitive noninterference—and this is the library we use to implement both Hails and λChair.

5.6.2 Discretionary access control and isolation

In this section, we show that LIO programs cannot write or allocate entities below the current label or read, write or allocate entities above their current clearance.[17] Building on this, we then show how LIO can be used to isolate untrusted computations to ensure they can only access a particular part of memory and any faults are contained, i.e., faults in the untrusted code do not percolate into the outer context.

[17] When considering privileges, in the style of the decentralized label model of Myers and Liskov [144], these access restrictions give the code containing the privilege the discretion to access certain entities below the current label and above the current clearance.

Discretionary access control

In the previous section, we showed that the current label after taking a step is always at least as restricting as the current label before taking the step. The dual holds for clearance; the current clearance after taking a step is always at most as restricting as the current clearance before taking the step.

Proposition 10 (Monotonicity of the current clearance)**.** *If* $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n} \langle l'_{cur}, c'_{cur}, m' \mid t'\rangle$ *then* $c'_{cur} \sqsubseteq c_{cur}$.

Proof. By induction on t, using the lattice-properties of labels (namely, reflexivity of $\sqsubseteq$) and the fact that only (LOWERCLEARANCE) modifies the clearance (for which the statement holds trivially). □

This proposition states that the current clearance monotonically decreases within a context. In other words, the context can give up access to certain entities as it progresses, but not conversely. This statement is the clearance equivalent of Proposition 6, which states that once a computation reads confidential data, it cannot lower its current label to write to entities less sensitive.

Before delving into our access control guarantees, we first define two store modifiers:

$$l \preceq m = \{(a, Labeled_b^{\mathsf{TCB}}\ l'\ t) : (a, Labeled_b^{\mathsf{TCB}}\ l'\ t) \in m \text{ and } l \sqsubseteq l'\}$$

$$m \preceq l = \{(a, Labeled_b^{\mathsf{TCB}}\ l'\ t) : (a, Labeled_b^{\mathsf{TCB}}\ l'\ t) \in m \text{ and } l' \sqsubseteq l\}$$

$$l_1 \preceq m \preceq l_2 = l_1 \preceq m \cap m \preceq l_2$$

Symbol $l \preceq m$ denotes the subset of m containing all the references whose labels are above or equal to l. Similarly, $m \preceq l$ contains the references whose label is below or equal to l. Operator $l_1 \preceq m \preceq l_2$ encompasses the subset of m containing all the reference whose labels are between the labels l_1 and l_2. Finally, we introduce the complement of the described subsets as $\overline{l \preceq m}$, $\overline{m \preceq l}$, and $\overline{l_1 \preceq m \preceq l_2}$, respectively.

Lemma 6 (No write-access below current label✎)**.** *Given a term* t *and memory* m*, such that* $\varsigma\ (t)$ *and* $\varsigma\ (m \preceq c_{cur})$*, if the term reduces to a value according to* $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{cur}, c'_{cur}, m' \mid t'\rangle$*, then* $\overline{l_{cur} \preceq m} = \overline{l_{cur} \preceq m'}$.

Intuitively, this lemma states that the partitions, of the initial and final memory stores, that (may) contain references with labels below l_{cur} are identical, i.e., the computation could not have modified or created references below l_{cur}. Note, however, that the lemma does not state that term t cannot read

from a reference below the current label. A corollary of this lemma states that any labeled values created by t are labeled above l_{cur}.

A similar, though slightly stronger, access control statement holds for clearance.

Lemma 7 (No access above current clearance✎)**.** *Given term t and memory m, such that* $\varsigma\ (t)$ *and* $\varsigma\ (m \preceq c_{cur})$*, if the term reduces to a value according to* $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{cur}, c'_{cur}, m' \mid t'\rangle$*, then* $\overline{m \preceq c_{cur}} = \overline{m' \preceq c_{cur}}$.

In other words, the partition of memory above the initial current clearance remains inaccessible throughout the program execution, i.e., the computation could not have modified or created references above c_{cur}. A corollary of this lemma states that any labeled values created by t are labeled below c_{cur}. As shown in Appendix A, computations also cannot read data above the clearance; this allows us to execute a term t with an alternative memory—one where references above the clearance are arbitrarily modified—without affecting its behavior.

From these two lemmas, we can further state that the current computation is restricted to modifying references whose labels are between the current label and clearance:

Proposition 11 (Memory writes bounded by current label and clearance✎)**.** *Given term t and memory m, such that* $\varsigma\ (t)$ *and* $\varsigma\ (m \preceq c_{cur})$*, if the term reduces to a value according to* $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{cur}, c'_{cur}, m' \mid t\rangle$*, then* $\overline{l_{cur} \preceq m \preceq c_{cur}} = \overline{l_{cur} \preceq m' \preceq c_{cur}}$.

Proof. Directly from Lemma 6 and Lemma 7. □

Isolation

Using the above access control properties of LIO, we now show how terms can be executed in isolation. To this end, we first define an *isolate* function, similar to the *withClearance* of Section 5.3:

$$
\begin{aligned}
&\mathit{isolate} :: \mathit{Label}\ \mathscr{L} \Rightarrow \mathscr{L} \to \mathscr{L} \to \mathit{LIO}\ \mathscr{L}\ () \to \mathit{LIO}\ \mathscr{L}\ ()\\
&\mathit{isolate}\ l\ c\ t = \mathit{toLabeled}\ c\ (\mathit{lowerClearance}\ c \gg \mathit{raiseLabel}\ l \gg t) \gg \mathit{return}\ ()\\
&\quad \textbf{where}\ \mathit{raiseLabel}\ l = \mathit{label}\ l\ () \gg\!= \mathit{unlabel}
\end{aligned}
$$

This function executes a term t in a context where the initial current label and clearance are l and c, respectively. While simple, this *isolation* function can be used to ensure that the untrusted term t can only modify a specific portion of memory and indeed, behave, as if it executes in a separate context:

Lemma 8 (Single term isolation✎)**.** *If* $\langle l_{cur}, c_{cur}, m \mid \mathit{isolate}\ l\ c\ t\rangle \xrightarrow{n+1}{}^{*} \langle l_{cur}, c_{cur}, m' \mid \mathit{LIO}^{\mathit{TCB}}_{\mathbf{true}}\ ()\rangle$, *then* $\overline{l \preceq m \preceq c} = \overline{l \preceq m' \preceq c}$, $m' = (l \preceq m' \preceq c) \cup (\overline{l \preceq m \preceq c})$, *and* $\langle l, c, m \mid t\rangle \xrightarrow{n}{}^{*} \langle l', c', m' \mid \mathit{LIO}^{\mathit{TCB}}_{\mathbf{true}}\ ()\rangle$.

Here, the memory equations simply state that term t could only have modified the part of the memory store m that is between l and c. Regardless of whether t terminates by raising the current label, lowering the current clearance, and/or throwing an exception, the *isolate* function ensures that this "fault" is not propagated to the outer computation. Indeed, this can directly be used to address the poison pill attacks described in [86]. Unfortunately, like the noninterference theorem, this lemma assumes that term t terminates.

By wrapping different terms with *isolate* and using disjoint labels for their corresponding current labels and clearances, we can guarantee that the terms will execute in isolation, on disjoint parts of the memory. Such a term isolation theorem, for two terms, is given below.

Theorem 2 (Term isolation✎)**.** *Assume* $\mathit{fresh}(\cdot)$ *deterministically creates objects that are globally unique. Given safe terms* t_1 *and* t_2*, memory* m*, and labels* l_1, c_1, l_2*, and* c_2*, bounded by* l_{cur} *and* c_{cur}*, such that* $l_1 \sqsubseteq c_1$, $l_2 \sqsubseteq c_2$, $l_1 \not\sqsubseteq l_2$, $l_2 \not\sqsubseteq l_1$, $c_1 \not\sqsubseteq c_2$, *and* $c_2 \not\sqsubseteq c_1$, *if* $\langle l_{cur}, c_{cur}, m \mid \mathit{isolate}\ l_1\ c_1\ t_1 \gg \mathit{isolate}\ l_2\ c_2\ t_2\rangle \xrightarrow{n}{}^{*} \langle l_{cur}, c_{cur}, m' \mid \mathit{LIO}^{\mathit{TCB}}\ ()\rangle$ *then* $\langle l_1, c_1, m \mid t_1\rangle \xrightarrow{n_1}{}^{*} \langle l_1', c_1', m_1 \mid \mathit{LIO}^{\mathit{TCB}}\ ()\rangle$, *and* $\langle l_2, c_2, m \mid t_2\rangle \xrightarrow{n_2}{}^{*} \langle l_2', c_2', m_2 \mid \mathit{LIO}^{\mathit{TCB}}\ ()\rangle$, $n = (n_1 + 1) + (n_2 + 1)$, *and* $\overline{l_1 \preceq m \preceq c_1} = \overline{l_1 \preceq m_1 \preceq c_1}$, $\overline{l_2 \preceq m \preceq c_2} = \overline{l_2 \preceq m_2 \preceq c_2}$, $l_1 \preceq m' \preceq c_1 = l_1 \preceq m_1 \preceq c_1$, *and* $l_2 \preceq m' \preceq c_2 = l_2 \preceq m_2 \preceq c_2$.

Intuitively, the theorem states that the behavior of terms t_1 and t_2 (under the supplied context labels) is not affected by *isolate* function. Importantly, it also states that the two terms operate on disjoint parts of the memory—indeed the behavior of t_2 is the same as executing it with initial memory m, as opposed to m_1, the memory after term t_1 was executed. In the context of λChair, this isolation property is especially important since it allows us to ensure that requests running on behalf of different users run in isolation.

5.7 Related Work on Language-level Dynamic IFC

Heintze and Riecke [81] consider security for lambda-calculus where lambda-terms are explicitly annotated with security labels, for a type-system that guarantees noninterference. One of the key ideas behind their work is to provide an operator that raises the security label of a term. Similarly, Zdancewic's PhD thesis [227] introduces a security λ-calculus which raises the *pc* associated to a term when sensitive information gets obtained by reading references. Austin and Flanagan [11] design a λ-calculus which might temporary raise the *pc* when reducing function application. These

features are similar to raising the current label when manipulating labeled values whose labels are above the current label. The notion of a floating current label dates back to the High-Water-Mark security model [108] of the ADEPT-50 in the late 1960s, which was later adopted by Asbestos [63], HiStar [228], and Flume [104] IFC Operating Systems.

Abadi et al. [1] develop the dependency core calculus (DCC) based on a hierarchy of monads to guarantee noninterference. In their calculus, they define a monadic type that protects the confidentiality of pure values at different security levels. Our *LIO* and *Labeled* types serve a similar role. However, since LIO has the guarantee that code cannot create labeled values below the current label and or above the current clearance, the *Labeled* type is not a monad—we must inspect the current label and clearance before a new labeled value can be created (e.g., by applying a function to the protected value). Nevertheless, we can use *unlabel* and *toLabeled* in the *LIO* monad to achieve the dynamic equivalent functionality of DCC's (non-standard) typing rules for the bind operator. Tse and Zdancewic [198] translate DCC to System F and show that noninterference can be stated using parametricity. Unfortunately, like DCC, they rely on a non-standard typing rule for bind—they provide several definitions for this operator and rely on GHC's `UndecidableInstance` extension (which lifts type conditions of [192]) to resolve the correct bind. Crary et al. [48] present a monadic calculus for noninterference for programs with mutable state. While inspired by these works, we do not take a domain-specific approach to extend the Haskell type system or modify the Haskell runtime; rather, we take a dynamic, label-polymorphic, and library approach to IFC. Importantly, our implementation does not rely on any non-standard constructs—this reduces the task of understanding IFC enforcement to understanding the LIO API.

Harrison and Hook show how to monadically encode abstract operating systems called *separation kernels* [77]. The idea behind this work is to first partition a program into multiple processes, each associated with a separate domain (label), running in isolation. Inter-process communication is allowed through a kernel that mediates the message exchange according to a security policy (e.g., noninterference). To formally reason about separation kernels, the authors use a monad-layering approach, modeling state with the *State* monad, concurrency with the *Resumption* monad, etc. This approach is orthogonal to our approach; we use monads in a trivial fashion and primarily as a way to implement the calculus semantics as a library. In [183] and Chapter 6, we describe the concurrent version of LIO, which unlike [77], considers termination-sensitive noninterference.

The seminal work by Li and Zdancewic [114] presents an implementation of information-flow security for Haskell. Instead of modifying the language runtime, they take a library-based approach by encoding IFC-constrained computations using arrows [87] (a generalization of monads). This

work was extended by Tsai et al. [46] to consider concurrency and side-effecting computations. Russo et al. [164] show an alternative library-based approach that eliminates the need for arrows; they, instead, describe a monadic library that encodes static IFC. This library relies on monadic types to track information-flow in pure and side-effecting computations. Morgenstern and Licata [138] extend this idea to implement an authorization- and IFC-aware programming language in Agda. However, and as is the case with many static systems [168], their library is less permissive. Nevertheless, this library is a closely related work. In particular, we note that the SecIO library [164] has functions that serve the static counterpart of some of the core LIO functions (e.g., like *unlabel*, they provide a function that maps pure labeled values into monadic computations; like *toLabeled*, they provide a function that allows safely writing to public entities after reading secret data).

Another closely related work is that of [56]; this work uses monad transformers and parametrized monads [10] to enforce noninterference, both dynamically and statically. Different from our work, they focus on modularity (separating IFC enforcement from underlying user API), using typeclass-level tricks. Unfortunately, like the work on separation kernels, this requires programmers to first partition their code to fit the new programming model, whereas the usage of *LIO* strives to be very close to Haskell's existing *IO* libraries.

Laminar [163] is a closely related system that combines OS- and PL-techniques to jointly provide application and OS end-to-end guarantees. Although our work does not extend to the OS, Laminar's OS-confinement could be unified with LIO, much as they unify the mechanism with their Java language-level system. More interestingly, at the language level, Laminar enforces IFC within certain code regions named *security regions*, where labeled data can be accessed. Security regions have a (secrecy and integrity) label associated with them and are superficially similar to our *toLabeled* blocks.[18] Unlike in LIO, however, security regions cannot change their current label; if code wishes to read data more sensitive than the region's label, it must create another region with the supplied label. Moreover, if code within a region violates a security check (e.g., attempts to write to less sensitive file), the Laminar runtime raises an exception. Each security region has a required catch block, which is executed when such an exception is raised. (Though, code within a region can terminate the process by exiting.) Catch blocks run with the same label as the security region and provides developers with a way for recovering from monitor failures. Importantly, the runtime suppresses exceptions raised within the security region's catch block and any exceptions

[18] Laminar also associates a set of capabilities as a means for declassification and endorsement, much like LIO's privileges. However, we do not discuss them further since we do not address such topics in this work.

not explicitly caught. Similar to our approach, this decision is done to avoid an exception raised in a sensitive context to suppress less sensitive subsequent actions. Despite this similarity, there are several differences between LIO and Laminar. First, LIO provides a single, flexible mechanism for handling exceptions; we do not treat monitor exceptions differently from other exceptions—thus reducing the abstractions developers must understand. Second, LIO does not suppress exceptions. Our *toLabeled* block delays exceptions which may be suppressed, but do not have to be—we can inspect the result of a *toLabeled* result, whether it is a failure or not. A result of these two points is that Laminar's secure regions can be implemented in LIO using *toLabeled* and *withClearance*. More importantly, we remark that LIO code does not have to be wrapped in *toLabeled* blocks—this is unlike Laminar, where code that handles labeled data must always be wrapped by a secure region.

The secure treatment of exception-handling has been studied by the mainstream IFC compilers Jif [145] and FlowCaml [174]. These compilers' type-systems enforce the following rule for exceptions: if an exception *might* be raised in a sensitive context, no public side effects must follow either in the subsequent code in a try block or in the catch handler. On the other hand, LIO enforces that *once exceptions are thrown* in a sensitive context, no subsequent public side effects can be executed either inside the *toLabeled* block where the exception is raised (if any) or in the catch handler. In [8], the authors provide a more permissive static exception-handling mechanism by introducing exceptions that *cannot be caught*. This idea could be easily incorporated in LIO and we state it as an interesting direction for future work.

Hedin and Sabelfeld present a dynamic information-flow monitor for a core JavaScript with exceptions [79]. In their calculus, they associate a security level with every exception. This is similar to our initial approach, described below, in associating the current label with exceptions thrown by *throwLIO*. Their semantics diverge from standard JavaScript in disallowing public exceptions from being thrown in secret contexts and, to address this permissiveness issue, they provide a non-standard construct that can be used to upgrade the label of an exception. Unfortunately, IFC violations (which may arise when an upgrade is not performed) are fatal.

Our initial treatment of exceptions was presented in the unpublished manuscript [188]. While the semantics are mostly the same as those presented in this chapter, there are some subtle differences. In particular, in the original work, exceptions had an associated explicit label—the current label at the time of a *throwLIO*. And, at the time of a *catchLIO*, the current label was raised to the join of the exception label and current label. Unfortunately, these semantics are unnecessarily complex due to the implementation. Specifically, the *LIO* monad was implemented as a *State* monad with *IO* as the base monad and the current label and clearance as the monad state. Since the monad state

may change according to the computation control flow, it was necessary that exceptions carry the additional state information to ensure that the current label is not arbitrarily lowered. By removing this implementation consideration, we were able to simplify the semantics to those presented in this chapter and also simplify the implementation—the key insight is that the current label and clearance are global to the computation and thus the *State* monad needs to only contain a reference to these labels. Indeed, this simplification reduced the complexity of exception handling to the interaction of exceptions and *toLabeled*.

In parallel with our initial work on exceptions, Hriţcu et al. presented the Breeze IFC language [86]. Breeze explored the design space of IFC and exceptions. Not only do they consider various calculi with exceptions, but, like our work, also address the issue of treating IFC monitor failures as recoverable failures. We refer the interested reader to the Breeze paper for a very comprehensive comparison of Breeze and LIO, and a detailed analysis of different design trade-offs that arise due to exceptions. Here, we only remark that, like Breeze, we delay the propagation of exceptions raised in *toLabeled* blocks (in Breeze, these are called brackets). Indeed, our semantics for exceptions are very similar to their calculus $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$. Both of these calculi differ from our original presentation [188] in hiding exceptions raised in a *toLabeled* block where the current label is above the supplied upper bound, see rule (TOLABELEDFAIL).

Different from most language-based IFC systems, LIO relies on the notion of clearance to restrict information leakage due to covert channels. Bell and La Padula [20] formalized clearance as a bound on the current label of particular users' processes. In the 1980s, clearance became a requirement for high-assurance secure systems purchased by the US Department of Defense [53]. HiStar [228] re-cast clearance as a bound on the label of any resource created by the process (where raising a process's label is but one means of creating a something with a higher label). We adopt HiStar's more stringent notion of clearance, which prevents software from copying data it cannot read and facilitates bounding the time during which possibly untrustworthy software can exploit covert channels.

5.8 Summary

This chapter presented LIO, an IFC system that explores a new design point in language-based information flow security. LIO takes a mostly coarse-grained labeling approach, inspired by both IFC OSes and IFC programming languages. In particular, LIO only associates a single, mutable, label—the *current label* with all the values in context (lexical scope) and dictates how information flows

to/from the context. Compared to typical language-based IFC systems, where labels are explicitly associated with values, this design approach is amenable to a fast, library implementation. But, to allow programmers to handle differently labeled data, LIO provides an abstract data type, *Labeled*, that encapsulates a term and its explicit label. (In a similar way we provide mutable labeled references.) *Labeled* values serve the dual purpose of addressing label creep—the raising of the current label as increasingly sensitive data is incorporated into the context—by encapsulating the result of sensitive sub-computation, as executed by *toLabeled*. Unlike other language-based work, our IFC system also implements *clearance* as a means for restricting the kinds of data a computation can read/write to; LIO relies on this form of discretionary access control to address covert channels: code cannot leak data it cannot read. Finally, LIO provides exception handling constructs which serve the dual purpose of encoding monitor failures, from which untrusted code can recover. This addresses a long standing problem with dynamic IFC enforcement—that monitor failures leak information.

We proved several security theorems for LIO. First, we showed that LIO programs, which may perform complex side-effects (e.g., mutate variables and throw exceptions), satisfy noninterference, i.e., LIO programs satisfy data confidentiality and integrity. Second, we showed that clearance is a form discretionary access control. And, finally, we showed that LIO can be used to execute terms in isolation, operating on disjoint parts of memory.

LIO has been implemented as a Haskell library, using Safe Haskell to ensure that untrusted code executes in the *LIO* monad, i.e., our IFC sub-language. To illustrate the expressiveness of LIO, we described the core of a conference review system, λChair, that uses IFC to enforce high-level security policies. In addition to λChair, we (and others) have used LIO to implement several other web applications, some of which are in production use. We found the library-based approach to be very effective, both in terms of deployment (at the time of this writing, the library has thousands of downloads) and design (the interface matured as a result of several iterations).

Chapter 6

IFC-Inside: Retroffitting Languages with Dynamic Information Flow Control[1]

Many important security problems in JavaScript, such as browser extension security, untrusted JavaScript libraries and safe integration of mutually distrustful websites (mash-ups), may be effectively addressed using an efficient implementation of information flow control (IFC). Unfortunately existing fine-grained approaches to JavaScript IFC require modifications to the language semantics and its engine, a non-goal for browser applications. In this chapter, we take the ideas of coarse-grained dynamic IFC and provide the theoretical foundation for a language-based approach that can be applied to any programming language for which external effects can be controlled. We then apply this formalism to server- and client-side JavaScript, show how it generalizes to the C programming language, and connect it to our Haskell LIO system. Our methodology offers design principles for the construction of information flow control systems when isolation can easily be achieved, as well as compositional proofs for optimized concrete implementations of these systems, by relating them to their isolated variants.

[1] This chapter is a copy of the extended version of the POST 2015 paper [82].

6.1 Introduction

Modern web content is rendered using a potentially large number of different components with differing provenance. Disparate and untrusting components may arise from browser extensions (whose JavaScript code runs alongside website code), web applications (with possibly untrusted third-party libraries), and mashups (which combine code and data from websites that may not even be aware of each other's existence.) While just-in-time combination of untrusting components offers great flexibility, it also poses complex security challenges. In particular, maintaining data privacy in the face of malicious extensions, libraries, and mashup components has been difficult.

Information flow control (IFC) is a promising technique that provides security by tracking the flow of sensitive data through a system. Untrusted code is confined so that it cannot exfiltrate data, except as per an information flow policy. Significant research has been devoted to adding various forms of IFC to different kinds of programming languages and systems. In the context of the web, however, there is a strong motivation to preserve JavaScript's semantics and avoid JavaScript-engine modifications, while retrofitting it with dynamic information flow control.

The Operating Systems community has tackled this challenge (e.g., in [228]) by taking a *coarse-grained* approach to IFC: dividing an application into coarse computational units, each with a single label dictating its security policy, and only monitoring communication between them. This coarse-grained approach provides a number of advantages when compared to the fine-grained approaches typically employed by language-based systems. First, adding IFC does not require intrusive changes to an existing programming language, thereby also allowing the reuse of existing programs. Second, it has a small runtime overhead because checks need only be performed at isolation boundaries instead of (almost) every program instruction (e.g., [78]). Finally, associating a single security label with the entire computational unit simplifies understanding and reasoning about the security guarantees of the system, without reasoning about most of the technical details of the semantics of the underlying programming language.

In this chapter, we present a framework which brings coarse-grained IFC ideas into a language-based setting: an information flow control system should be thought of as multiple instances of completely isolated language runtimes or *tasks*, with information flow control applied to inter-task communication. We describe a formal system in which an IFC system can be designed once and then applied to any programming language which has control over external effects (e.g., JavaScript or C with access to hardware privilege separation). We formalize this system using an approach by Matthews and Findler [125] for combining operational semantics and prove non-interference

guarantees that are independent of the choice of a specific target language.

There are a number of points that distinguish this setting from previous coarse-grained IFC systems. First, even though the underlying semantic model involves communicating tasks, these tasks can be coordinated together in ways that simulate features of traditional languages. In fact, simulating features in this way is a useful *design tool* for discovering what variants of the features are permissible and which are not. Second, although completely separate tasks are semantically easy to reason about, real-world implementations often blur the lines between tasks in the name of efficiency. Characterizing what optimizations are permissible is subtle, since removing transitions from the operational semantics of a language can break non-interference. We partially address this issue by characterizing isomorphisms between the operational semantics of our abstract language and a concrete implementation, showing that if this relationship holds, then non-interference in the abstract specification carries over to the concrete implementation.

Our contributions can be summarized as follows:

- We give formal semantics for a core coarse-grained dynamic information flow control language free of non-IFC constructs. We then show how a large class of target languages can be combined with this IFC language and prove that the result provides non-interference. (Sections 6.2 and 6.3)

- We provide a proof technique to show the non-interference of a concrete semantics for a potentially optimized IFC language by means of an isomorphism and show a class of restrictions on the IFC language that preserves non-interference. (Section 6.4)

- We have implemented an IFC system based on these semantics for Node.js, and we connect our formalism to another implementation based on this work for client-side JavaScript [189]. Furthermore, we outline an implementation for the C programming language and describe improvements to our Haskell LIO system that resulted from this framework. (Section 6.5)

6.2 Retrofitting Languages with IFC

Before moving on to the formal treatment of our system, we give a brief primer of information flow control and describe some example programs in our system, emphasizing the parallel between their implementation in a multi-task setting, and the traditional, “monolithic” programming language feature they simulate.

Information flow control systems operate by associating data with *labels*, and specifying whether or not data tagged with one label l_1 can flow to another label l_2 (written as $l_1 \sqsubseteq l_2$). These labels encode the desired security policy (for example, confidential information should not flow to a public channel), while the work of specifying the semantics of an information flow language involves demonstrating that impermissible flows cannot happen, a property called *non-interference* [73]. In our coarse-grained floating-label approach, labels are associated with tasks. The task label—we refer to the label of the currently executing task as the *current label*—serves to protect everything in the task's scope; all data in a task shares this common label.

As an example, here is a program which spawns a new isolated task, and then sends it a mutable reference:

$$\begin{aligned}&\textbf{let } i = {}_{\text{TI}}\lfloor \textbf{sandbox } (\textbf{blockingRecv } x, \_ \textbf{ in } {}^{\text{IT}}\lceil\, !\, {}_{\text{TI}}\lfloor x \rfloor \rceil)\rfloor\\ &\textbf{in } {}_{\text{TI}}\lfloor \textbf{send } {}^{\text{IT}}\lceil i \rceil\ l\ {}^{\text{IT}}\lceil \textbf{ref true} \rceil\rfloor\end{aligned}$$

For now, ignore the tags ${}_{\text{TI}}\lfloor \cdot \rfloor$ and ${}^{\text{IT}}\lceil \cdot \rceil$: roughly, this code creates a new **sandbox**ed task with identifier i which waits (**blockingRecv**, binding x with the received message) for a message, and then **send**s the task a mutable reference (**ref true**) which it labels l. If this operation actually shared the mutable cell between the two tasks, it could be used to violate information flow control if the tasks had differing labels. At this point, the designer of an IFC system might add label checks to mutable references, to check the labels of the reader and writer. While this solves the leak, for languages like JavaScript, where references are prevalently used, this also dooms the performance of the system.

Our design principles suggest a different resolution: when these constructs are treated as isolated tasks, each of which have their own heaps, it is obviously the case that there is no sharing; in fact, the sandboxed task receives a dangling pointer. Even if there is only one heap, if we enforce that references not be shared, the two systems are morally equivalent. (We elaborate on this formally in Section 6.4.) Finally, this semantics strongly suggests that one should restrict the types of data which may be passed between tasks (for example, in JavaScript, one might only allow JSON objects to be passed between tasks, rather than general object structures).

Existing language-based, coarse-grained IFC systems [86, 188] allow a sub-computation to temporarily raise the floating-label; after the sub-computation is done, the floating-label is restored to its original label. When this occurs, the enforcement mechanism must ensure that information does not

leak to the (less confidential) program continuation. The presence of exceptions adds yet more intricacies. For instance, exceptions should not automatically propagate from a sub-computation directly into the program continuation, and, if such exceptions are allowed to be inspected, the floating-label at the point of the exception-raise must be tracked alongside the exception value [86, 188, 79]. In contrast, our system provides the same flexibility and guarantees with no extra checks: tasks are used to execute sub-computations, but the mere definition of isolated tasks guarantees that (a) tasks only transfer data to the program continuation by using inter-task communication means, and (b) exceptions do cross tasks boundaries automatically.

6.2.1 Preliminaries

Our goal now is to describe how to take a **target language** with a formal operational semantics and combine it with an *information flow control language*. For example, taking ECMAScript as the target language and combining it with our IFC language should produce the formal semantics for the core part of COWL [189] (see Chapter 3). In this presentation, we use a simple, untyped lambda calculus with mutable references and fixpoint in place of ECMAScript to demonstrate some the key properties of the system (and, because the embedding does not care about the target language features); we discuss the proper embedding in more detail in Section 6.5.

Notation We have typeset nonterminals of the target language using **bold font** while the nonterminals of the IFC language have been typeset with *italic font*. Readers are encouraged to view a color copy of this chapter, where target language nonterminals are colored **red** and IFC language nonterminals are colored *blue*.

6.2.2 Target Language: Mini-ES

In Figure 6.1, we give a simple, untyped lambda calculus with mutable references and fixpoint, prepared for combination with an information flow control language. The presentation is mostly standard, and utilizes Felleisen-Hieb reduction semantics [65] to define the operational semantics of the system. One peculiarity is that our language defines an evaluation context $\boldsymbol{E}$, but, the evaluation rules have been expressed in terms of a different evaluation context $\mathscr{E}_{\boldsymbol{\Sigma}}$; Here, we follow the approach of Matthews and Findler [125] in order to simplify combining semantics of multiple languages. To derive the usual operational semantics for this language, the evaluation context merely needs to be defined as $\mathscr{E}_{\boldsymbol{\Sigma}}[\boldsymbol{e}] \triangleq \boldsymbol{\Sigma}, \boldsymbol{E}[\boldsymbol{e}]$. However, when we combine this language with an IFC language, we reinterpret the meaning of this evaluation context.

$$
\begin{array}{lll}
\boldsymbol{v} & ::= & \lambda \boldsymbol{x}.\boldsymbol{e} \mid \textbf{true} \mid \textbf{false} \mid \boldsymbol{a} \\
\boldsymbol{e} & ::= & \boldsymbol{v} \mid \boldsymbol{x} \mid \boldsymbol{e}\ \boldsymbol{e} \mid \textbf{if } \boldsymbol{e} \textbf{ then } \boldsymbol{e} \textbf{ else } \boldsymbol{e} \mid \textbf{ref } \boldsymbol{e} \mid !\boldsymbol{e} \mid \boldsymbol{e} := \boldsymbol{e} \mid \textbf{fix } \boldsymbol{e} \\
\boldsymbol{E} & ::= & [\cdot]_{\boldsymbol{T}} \mid \boldsymbol{E}\ \boldsymbol{e} \mid \boldsymbol{v}\ \boldsymbol{E} \mid \textbf{if } \boldsymbol{E} \textbf{ then } \boldsymbol{e} \textbf{ else } \boldsymbol{e} \mid \textbf{ref } \boldsymbol{E} \mid !\boldsymbol{E} \mid \boldsymbol{E} := \boldsymbol{e} \mid \boldsymbol{v} := \boldsymbol{E} \mid \textbf{fix } \boldsymbol{E} \\
\boldsymbol{e}_1; \boldsymbol{e}_2 & \triangleq & (\lambda \boldsymbol{x}.\boldsymbol{e}_2)\ \boldsymbol{e}_1 \textbf{ where } \boldsymbol{x} \notin \mathscr{FV}(\boldsymbol{e}_2) \\
\textbf{let } \boldsymbol{x} = \boldsymbol{e}_1 \textbf{ in } \boldsymbol{e}_2 & \triangleq & (\lambda \boldsymbol{x}.\boldsymbol{e}_2)\ \boldsymbol{e}_1
\end{array}
$$

$$\text{T-APP}\quad \frac{}{\mathscr{E}_{\boldsymbol{\Sigma}}[(\lambda x.\boldsymbol{e})\ \boldsymbol{v}] \rightarrow \mathscr{E}_{\boldsymbol{\Sigma}}[\{\boldsymbol{v}/x\}\ \boldsymbol{e}]} \qquad \text{T-IFTRUE}\quad \frac{}{\mathscr{E}_{\boldsymbol{\Sigma}}[\textbf{ if true then } \boldsymbol{e}_1 \textbf{ else } \boldsymbol{e}_2] \rightarrow \mathscr{E}_{\boldsymbol{\Sigma}}[\boldsymbol{e}_1]}$$

Figure 6.1: λ_{ES}: simple untyped lambda calculus extended with booleans, mutable references and general recursion. For space reasons we only show two representative reduction rules; full rules can be found in Appendix B.1.

In general, we require that a target language be expressed in terms of some global machine state $\boldsymbol{\Sigma}$, some evaluation context $\boldsymbol{E}$, some expressions $\boldsymbol{e}$, some set of values $\boldsymbol{v}$ and a *deterministic* reduction relation on full configurations $\boldsymbol{\Sigma} \times \boldsymbol{E} \times \boldsymbol{e}$.

6.2.3 IFC Language

As mentioned previously, most modern, dynamic information flow control languages encode policy by associating a label with data. Our embedding is agnostic to the choice of labeling scheme; we only require the labels to form a lattice [51] with the partial order $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$. In this chapter, we simply represent labels with the metavariable l, but do not discuss them in more detail. To enforce labels, the IFC monitor inspects the current label before performing a read or a write to decide whether the operation is permitted. A task can only write to entities that are at least as sensitive. Similarly, it can only read from entities that are less sensitive. However, as in other floating-label systems, this current label can be raised to allow the task to read from more sensitive entities at the cost of giving up the ability to write to others.

In Figure 6.2, we give the syntax and *single-task* evaluation rules for a minimal information flow control language. Ordinarily, information flow control languages are defined by directly stating a base language plus information flow control operators. In contrast, our language is purposely minimal: it does not have sequencing operations, control flow, or other constructs. However, it contains support for the following core information flow control features:

- First-class labels, with label values l as well as operations for computing on labels ($\sqsubseteq$, $\sqcup$ and $\sqcap$).

- Operations for inspecting (**getLabel**) and modifying (**setLabel**) the current label of the task (a task can only increase its label).
- Operations for non-blocking inter-task communication (**send** and **recv**), which interact with the global store of per-task message queues Σ.
- A sandboxing operation used to spawn new isolated tasks. In concurrent settings **sandbox** corresponds to a fork-like primitive, whereas in a sequential setting, it more closely resembles computations which might temporarely raise the current floating-label [186, 86].

These operations are all defined with respect to an evaluation context $\mathscr{E}_{\Sigma}^{i,l}$ that represents the context of the current task. The evaluation context has three important pieces of state: the global message queues Σ, the current label l and the task ID i.

We note that first-class labels, tasks (albeit named differently), and operations for inspecting the current label are essentially universal to all floating-label systems. However, our choice of communication primitives is motivated by those present in browsers, namely `postMessage` [210]. Of course, other choices, such as blocking communication or labeled channels, are possible.

These asynchronous communication primitives are worth further discussion. When a task is sending a message using **send**, it also labels that message with a label l' (which must be at or above the task's current label l). Messages can only be received by a task if its current label is at least as high as the label of the message. Specifically, receiving a message using **recv** x_1, x_2 **in** e_1 **else** e_2 binds the message and the sender's task identifier to local variables x_1 and x_2, respectively, and then executes e_1. Otherwise, if there are no messages, that task continues its execution with e_2. We denote the filtering of the message queue by $\Theta \preceq l$, which is defined as follows. If Θ is the empty list **nil**, the function is simply the identity function, i.e., $\mathbf{nil} \preceq l = \mathbf{nil}$, and otherwise:

$$((l', i, e), \Theta) \preceq l = \begin{cases} (l', i, e), (\Theta \preceq l) & \text{if } l' \sqsubseteq l \\ \Theta \preceq l & \text{otherwise} \end{cases}$$

This ensures that tasks cannot receive messages that are more sensitive than their current label would allow.

6.2.4 The Embedding

Figure 6.3 provides all of the rules responsible for actually carrying out the embedding of the IFC language within the target language. The most important feature of this embedding is that every task

$$
\begin{array}{l}
v ::= i \mid l \mid \textbf{true} \mid \textbf{false} \mid \langle\rangle \qquad\qquad \otimes ::= \sqsubseteq \mid \sqcup \mid \sqcap \\
e ::= v \mid x \mid e \otimes e \mid \textbf{getLabel} \mid \textbf{setLabel}\ e \mid \textbf{taskId} \mid \textbf{sandbox}\ e \\
\quad \mid \textbf{send}\ e\ e\ e \mid \textbf{recv}\ x,x\ \textbf{in}\ e\ \textbf{else}\ e \\
E ::= [\cdot]_l \mid E \otimes e \mid v \otimes E \mid \textbf{setLabel}\ E \mid \textbf{send}\ E\ e\ e \mid \textbf{send}\ v\ E\ e \mid \textbf{send}\ v\ v\ E \\
\theta ::= (l, i\ e) \qquad\qquad \Theta ::= \textbf{nil} \mid \theta, \Theta \qquad\qquad \Sigma ::= \emptyset \mid \Sigma\,[i \mapsto \Theta]
\end{array}
$$

$$\text{I-GETTASKID}\ \frac{}{\mathscr{E}^{i,l}_{\Sigma}\,[\textbf{taskId}] \rightarrow \mathscr{E}^{i,l}_{\Sigma}\,[i]} \qquad \text{I-GETLABEL}\ \frac{}{\mathscr{E}^{i,l}_{\Sigma}\,[\textbf{getLabel}] \rightarrow \mathscr{E}^{i,l}_{\Sigma}\,[l]} \qquad \text{I-LABELOP}\ \frac{[\![l_1 \otimes l_2]\!] = v}{\mathscr{E}^{i,l}_{\Sigma}\,[l_1 \otimes l_2] \rightarrow \mathscr{E}^{i,l}_{\Sigma}\,[v]}$$

$$\text{I-SEND}\ \frac{l \sqsubseteq l' \qquad \Sigma(i') = \Theta \qquad \Sigma' = \Sigma\left[i' \mapsto (l', i, v), \Theta\right]}{\mathscr{E}^{i,l}_{\Sigma}\left[\textbf{send}\ i'\ l'\ v\right] \rightarrow \mathscr{E}^{i,l}_{\Sigma'}\,[\langle\rangle]}$$

$$\text{I-RECV}\ \frac{(\Sigma(i) \preceq l) = \theta_1, ..., \theta_k, (l', i', v) \qquad \Sigma' = \Sigma\,[i \mapsto (\theta_1, ..., \theta_k)]}{\mathscr{E}^{i,l}_{\Sigma}\,[\textbf{recv}\ x_1, x_2\ \textbf{in}\ e_1\ \textbf{else}\ e_2] \rightarrow \mathscr{E}^{i,l}_{\Sigma'}\left[\{\,v\,/\,x_1, i'\,/\,x_2\,\}\,e_1\right]}$$

$$\text{I-NORECV}\ \frac{\Sigma(i) \preceq l = \textbf{nil} \qquad \Sigma' = \Sigma\,[i \mapsto \textbf{nil}]}{\mathscr{E}^{i,l}_{\Sigma}\,[\textbf{recv}\ x_1, x_2\ \textbf{in}\ e_1\ \textbf{else}\ e_2] \rightarrow \mathscr{E}^{i,l}_{\Sigma'}\,[e_2]} \qquad \text{I-SETLABEL}\ \frac{l \sqsubseteq l'}{\mathscr{E}^{i,l}_{\Sigma}\left[\textbf{setLabel}\ l'\right] \rightarrow \mathscr{E}^{i,l'}_{\Sigma}\,[\langle\rangle]}$$

Figure 6.2: IFC language with all single-task operations.

$$v ::= \cdots \mid {}^{\mathrm{IT}}\lceil \boldsymbol{v} \rceil \qquad \boldsymbol{v} ::= \cdots \mid {}_{\mathrm{TI}}\lfloor v \rfloor \qquad \mathscr{E}_{\boldsymbol{\Sigma}}[e] \triangleq \Sigma; \langle \boldsymbol{\Sigma}, E[\boldsymbol{e}]_{\boldsymbol{T}} \rangle^i_l, \ldots$$
$$e ::= \cdots \mid {}^{\mathrm{IT}}\lceil \boldsymbol{e} \rceil \qquad \boldsymbol{e} ::= \cdots \mid {}_{\mathrm{TI}}\lfloor e \rfloor \qquad \mathscr{E}^{i,l}_{\Sigma}[e] \triangleq \Sigma; \langle \boldsymbol{\Sigma}, E[e]_I \rangle^i_l, \ldots$$
$$E ::= \cdots \mid {}^{\mathrm{IT}}\lceil \boldsymbol{E} \rceil \qquad \boldsymbol{E} ::= \cdots \mid {}_{\mathrm{TI}}\lfloor E \rfloor \qquad \mathscr{E}[e] \to \Sigma; t, \ldots \triangleq \mathscr{E}[e] \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\mathrm{step}}(t, \ldots)$$

I-SANDBOX
$$\frac{\boldsymbol{\Sigma}' = \Sigma\left[i' \mapsto \mathbf{nil}\right] \qquad \boldsymbol{\Sigma}' = \kappa(\boldsymbol{\Sigma}) \qquad t_1 = \langle \boldsymbol{\Sigma}, E[i'] \rangle^i_l \qquad t_{\mathrm{new}} = \langle \boldsymbol{\Sigma}', e \rangle^{i'}_l \qquad \mathrm{fresh}(i')}{\Sigma; \langle \boldsymbol{\Sigma}, E[\mathbf{sandbox}\ e]_I \rangle^i_l, \ldots \overset{\alpha}{\hookrightarrow} \Sigma'; \alpha_{\mathrm{sandbox}}(t_1, \ldots, t_{\mathrm{new}})}$$

I-DONE
$$\frac{}{\Sigma; \langle \boldsymbol{\Sigma}, v \rangle^i_l, \ldots \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\mathrm{done}}(\langle \boldsymbol{\Sigma}, v \rangle^i_l, \ldots)}$$

I-NOSTEP
$$\frac{\Sigma; t, \ldots \overset{\alpha}{\not\hookrightarrow}}{\Sigma; t, \ldots \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\mathrm{noStep}}(t, \ldots)}$$

I-BORDER
$$\frac{}{\mathscr{E}^{i,l}_{\Sigma}\left[{}^{\mathrm{IT}}\lceil {}_{\mathrm{TI}}\lfloor e \rfloor \rceil\right] \to \mathscr{E}^{i,l}_{\Sigma}[e]}$$

T-BORDER
$$\frac{}{\mathscr{E}_{\boldsymbol{\Sigma}}\left[{}_{\mathrm{TI}}\lfloor {}^{\mathrm{IT}}\lceil \boldsymbol{e} \rceil \rfloor\right] \to \mathscr{E}_{\boldsymbol{\Sigma}}[e]}$$

Figure 6.3: The embedding $L_{\mathrm{IFC}}(\alpha, \lambda)$, where $\lambda = (\boldsymbol{\Sigma}, \boldsymbol{E}, \boldsymbol{e}, \boldsymbol{v}, \to)$

maintains its own copy of the target language global state and evaluation context, thus enforcing isolation between various tasks. In more detail:

- We extend the values, expressions and evaluation contexts of both languages to allow for terms in one language to be embedded in the other, as in [125]. In the target language, an IFC expression appears as ${}_{\mathrm{TI}}\lfloor e \rfloor$ ("Target-outside, IFC-inside"); in the IFC language, a target language expression appears as ${}^{\mathrm{IT}}\lceil \boldsymbol{e} \rceil$ ("IFC-outside, target-inside").
- We reinterpret $\mathscr{E}$ to be evaluation contexts on task lists, providing definitions for $\mathscr{E}_{\boldsymbol{\Sigma}}$ and $\mathscr{E}^{i,l}_{\Sigma}$. These rules only operate on the first task in the task list, which by convention is the only task executing.
- We reinterpret $\to$, an operation on a single task, in terms of $\hookrightarrow$, operation on task lists. The correspondence is simple: a task executes a step and then is rescheduled in the task list according to schedule policy α. Figure 6.4 defines two concrete schedulers.
- Finally, we define some rules for scheduling, handling sandboxing tasks (which interact with the state of the target language), and intermediating between the borders of the two languages.

$$
\begin{aligned}
\mathrm{RR}_{\mathrm{step}}(t_1, t_2, \ldots) &= t_2, \ldots, t_1 \\
\mathrm{RR}_{\mathrm{done}}(t_1, t_2, \ldots) &= t_2, \ldots \\
\mathrm{RR}_{\mathrm{noStep}}(t_1, t_2, \ldots) &= t_2, \ldots \\
\mathrm{RR}_{\mathrm{sandbox}}(t_1, t_2, \ldots) &= t_2, \ldots, t_1
\end{aligned}
\qquad
\begin{aligned}
\mathrm{SEQ}_{\mathrm{step}}(t_1, t_2, \ldots) &= t_1, t_2, \ldots \\
\mathrm{SEQ}_{\mathrm{noStep}}(t_1, t_2, \ldots) &= t_1, t_2, \ldots \\
\mathrm{SEQ}_{\mathrm{done}}(t) &= t \\
\mathrm{SEQ}_{\mathrm{done}}(t_1, t_2, \ldots) &= t_2, \ldots \\
\mathrm{SEQ}_{\mathrm{sandbox}}(t_1, t_2, \ldots, t_n) &= t_n, t_1, t_2, \ldots
\end{aligned}
$$

Figure 6.4: Scheduling policies (concurrent round robin on the left, sequential on the right).

The I-SANDBOX rule is used to create a new isolated task that executes separately from the existing tasks (and can be communicated with via **send** and **recv**). When the new task is created, there is the question of what the target language state of the new task should be. Our rule is stated generically in terms of a function κ. Conservatively, κ may be simply thought of as the identity function, in which case the semantics of **sandbox** are such that the state of the target language is *cloned* when sandboxing occurs. However, this is not necessary: it is also valid for κ to remove entries from the state. In Section 6.4, we give a more detailed discussion of the implications of the choice of κ, but all our security claims will hold regardless of the choice of κ.

The rule I-NOSTEP says something about configurations for which it is not possible to take a transition. The notation $c \overset{\alpha}{\not\longrightarrow}$ in the premise is meant to be understood as follows: If the configuration c cannot take a step by any rule other than I-NOSTEP, then I-NOSTEP applies and the stuck task gets removed.

Rules I-DONE and I-NOSTEP define the behavior of the system when the current thread has reduced to a value, or gotten stuck, respectively. While these definitions simply rely on the underlying scheduling policy α to modify the task list, as we describe in Sections 6.3 and 6.6, these rules (notably, I-NOSTEP) are crucial to proving our security guarantees. For instance, it is unsafe for the whole system to get stuck if a particular task gets stuck, since a sensitive thread may then leverage this to leak information through the termination channel. Instead, as our example round-robin (RR) scheduler shows, such tasks should simply be removed from the task list. Many language runtime or Operating System schedulers implement such schedulers. Moreover, techniques such as instruction-based scheduling [182, 34] can be further applied close the gap between specified semantics and implementation.

As in [125], rules T-BORDER and I-BORDER define the syntactic boundaries between the IFC and target languages. Intuitively, the boundaries respectively correspond to an upcall into and downcall from the IFC runtime. As an example, taking λ_{ES} as the target language, we can now define a blocking receive (inefficiently) in terms of the asynchronous **recv** as series of cross-language calls:

$$\textbf{blockingRecv}\ x_1, x_2\ \textbf{in}\ e \triangleq {}^{\mathrm{IT}}\lceil \textbf{fix}\ (\lambda k.{}_{\mathrm{TI}}\lfloor \textbf{recv}\ x_1, x_2\ \textbf{in}\ e\ \textbf{else}\ {}^{\mathrm{IT}}\lceil k \rceil \rfloor)\rceil$$

For any target language λ and scheduling policy α, this embedding defines an IFC language, which we will refer to as $L_{\mathrm{IFC}}(\alpha, \lambda)$.

6.3 Security Guarantees

We are interested in proving non-interference about many programming languages. This requires an appropriate definition of this notion that is language agnostic, so in this section, we present a few general definitions for what an information flow control language is and what non-interference properties it may have. In particular, we show that $L_{\mathrm{IFC}}(\alpha, \lambda)$, with an appropriate scheduler α, satisfies non-interference [73], without making any reference to properties of λ. We state the appropriate theorems here, and provide the formal proofs in Appendix B.4.

6.3.1 Erasure Function

When defining the security guarantees of an information flow control, we must characterize what the *secret inputs* of a program are. Like other work [116, 164, 186, 183], we specify and prove non-interference using *term erasure*. Intuitively, term erasure allows us to show that an attacker does not learn any sensitive information from a program if the program behaves identically (from the attackers point of view) to a program with all sensitive data "erased". To interpret a language under information flow control, we define a function ε_l that performs erasures by mapping configurations to erased configurations, usually by rewriting (parts of) configurations that are more sensitive than l to a new syntactic construct $\bullet$. We define an information flow control language as follows:

Definition 7 (Information flow control language)**.** An information flow control language $\mathbf{L}$ is a tuple $(\Delta, \hookrightarrow, \varepsilon_l)$, where Δ is the type of machine configurations (members of which are usually denoted by the metavariable c), $\hookrightarrow$ is a reduction relation between machine configurations and $\varepsilon_l : \Delta \to \varepsilon(\Delta)$ is an erasure function parametrized on labels from machine configurations to *erased* machine configurations $\varepsilon(\Delta)$. Sometimes, we use V to refer to set of terminal configurations in Δ, i.e., configurations where no further transitions are possible.

Our language $L_{\mathrm{IFC}}(\alpha, \lambda)$ fulfills this definition as $(\Delta, \overset{\alpha}{\hookrightarrow}, \varepsilon_l)$, where $\Delta = \Sigma \times \mathrm{List}(t)$. The set of terminal conditions V is $\Sigma \times t_V$, where $t_V \subset t$ is the type for tasks whose expressions have been

$$\varepsilon_l(\Sigma; ts) = \varepsilon_l(\Sigma); \text{filter } (\lambda t.t = \bullet) \text{ (map } \varepsilon_l \; ts)$$

$$\varepsilon_l(\langle \Sigma, e \rangle^i_{l'}) = \begin{cases} \bullet & l' \not\sqsubseteq l \\ \langle \varepsilon_l(\Sigma), \varepsilon_l(e) \rangle^i_{l'} & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\Sigma\,[i \mapsto \Theta]) = \begin{cases} \varepsilon_l(\Sigma) & l' \not\sqsubseteq l, \text{ where } l' \text{ is the label of thread } i \\ \varepsilon_l(\Sigma)\,[i \mapsto \varepsilon_l(\Theta)] & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\Theta) = \Theta \preceq l \qquad \varepsilon_l(\emptyset) = \emptyset$$

Figure 6.5: Erasure function for tasks, queue maps, message queues, and configurations. In all other cases, including target-language constructs, ε_l is applied homomorphically. Note that $\varepsilon_l(e)$ is always equal to e (and similar for Σ) in this simple setting. However, when the IFC language is extended with more constructs as shown in Section 6.6, then this will no longer be the case.

reduced to values.[2] The erased configuration $\varepsilon(\Delta)$ extends Δ with configurations containing $\bullet$, and Figure 6.5 gives the precise definition for our erasure function ε_l. Essentially, a task and its corresponding message queue is completely erased from the task list if its label does not flow to the attacker observation level l. Otherwise, we apply the erasure function homomorphically and remove any messages from the task's message queue that are more sensitive than l.

The definition of an erasure function is quite important: it captures the attacker model, stating what can and cannot be observed by the attacker. In our case, we assume that the attacker cannot observe sensitive tasks or messages, or even the number of such entities. While such assumptions are standard [31, 183], our definitions allow for stronger attackers that may be able to inspect resource usage.[3]

6.3.2 Non-Interference

Given an information flow control language, we can now define non-interference. Intuitively, we want to make statements about the attacker's observational power at some security level l. This is done by defining an equivalence relation called l-equivalence on configurations: an attacker should

[2] Here, we abuse notation by describing types for configuration parts using the same metavariables as the "instance" of the type, e.g., t for the type of task.

[3] We believe that we can extend $L_{\text{IFC}}(\alpha, \lambda)$ to such models using the resource limits techniques of [218]. We leave this extension to future work.

not be able to distinguish two configurations that are l-equivalent. Since our erasure function captures what an attacker can or cannot observe, we simply define this equivalence as the syntactic-equivalence of erased configurations [183].

Definition 8 (l-equivalence)**.** In a language $(\Delta, \hookrightarrow, \varepsilon_l)$, two machine configurations $c, c' \in \Delta$ are considered l-equivalent, written as $c \approx_l c'$, if $\varepsilon_l(c) = \varepsilon_l(c')$.

We can now state that a language satisfies non-interference if an attacker at level l cannot distinguish the runs of any two l-equivalent configurations. This particular property is called termination sensitive non-interference (TSNI). Besides the obvious requirement to not leak secret information to public channels, this definition also requires the termination of public tasks to be independent of secret tasks. Formally, we define TSNI as follows:

Definition 9 (Termination Sensitive Non-Interference (TSNI))**.** A language $(\Delta, \hookrightarrow, \varepsilon_l)$ satisfies termination sensitive non-interference if for any label l, and configurations $c_1, c_1', c_2 \in \Delta$, if

$$c_1 \approx_l c_2 \qquad \text{and} \qquad c_1 \hookrightarrow^* c_1' \tag{6.1}$$

then there exists a configuration $c_2' \in \Delta$ such that

$$c_1' \approx_l c_2' \qquad \text{and} \qquad c_2 \hookrightarrow^* c_2' \,. \tag{6.2}$$

In other words, if we take two l-equivalent configurations, then for every intermediate step taken by the first configuration, there is a corresponding number of steps that the second configuration can take to result in a configuration that is l-equivalent to the first resultant configuration. By symmetry, this applies to all intermediate steps from the second configuration as well. We remark that this notion of non-interfernce is similar to *progress sensitive non-interference (PSNI)*, which accounts for leakage via progress (or termination) channels, as used for static systems [137].

Our language satisfies TSNI (and thus PSNI) under the round-robin scheduler RR of Figure 6.4.

Theorem 3 (Concurrent IFC language is TSNI)**.** *For any target language λ, $L_{IFC}(\mathrm{RR}, \lambda)$ satisfies TSNI.*

In general, however, non-interference will not hold for an arbitrary scheduler α. For example, $L_{\mathrm{IFC}}(\alpha, \lambda)$ with a scheduler that inspects a sensitive task's current state when deciding which task to schedule next will in general break non-interference [165, 18].

However, even non-adversarial schedulers are not always safe. Consider, for example, the sequential scheduling policy SEQ given in Figure 6.4. It is easy to show that $L_{\text{IFC}}(\text{SEQ}, \lambda)$ does not satisfy TSNI: consider a target language similar to λ_{ES} with an additional expression terminal $\Uparrow$ that denotes a divergent computation, i.e., $\Uparrow$ always reduces to $\Uparrow$ and a simple label lattice $\{\mathsf{pub}, \mathsf{sec}\}$ such that $\mathsf{pub} \sqsubseteq \mathsf{sec}$, but $\mathsf{sec} \not\sqsubseteq \mathsf{pub}$. Consider the following two configurations in this language:

$$c_1 = \Sigma; \langle \Sigma_1, {}^{\text{IT}}\lceil \textbf{ if false then } \Uparrow \textbf{ else true} \rceil \rangle^1_{\mathsf{sec}}, \langle \Sigma_2, e \rangle^2_{\mathsf{pub}}$$
$$c_2 = \Sigma; \langle \Sigma_1, {}^{\text{IT}}\lceil \textbf{ if true then } \Uparrow \textbf{ else true} \rceil \rangle^1_{\mathsf{sec}}, \langle \Sigma_2, e \rangle^2_{\mathsf{pub}}$$

These two configurations are pub-equivalent, but c_1 will reduce (in two steps) to configuration $c_1' = \Sigma; \langle \Sigma_1, {}^{\text{IT}}\lceil \textbf{true} \rceil \rangle^2_{\mathsf{pub}}$, whereas c_2 will not make any progress. Suppose that e is a computation that writes to a pub channel,[4] then the sec task's decision to diverge or not is directly leaked to a public entity.

To accommodate for sequential languages, or cases where a weaker guarantee is sufficient, we consider an alternative non-interference property called termination insensitive non-interference (TINI). This property can also be upheld by sequential languages at the cost of leaking through (non)-termination [7].

Definition 10 (Termination insensitive non-interference (TINI))**.** A language $(\Delta, V, \hookrightarrow, \varepsilon_l)$ is termination insensitive non-interfering if for any label l, and configurations $c_1, c_2 \in \Delta$ and $c_1', c_2' \in V$, it holds that

$$(c_1 \approx_l c_2 \ \wedge \ c_1 \hookrightarrow^* c_1' \ \wedge \ c_2 \hookrightarrow^* c_2') \implies c_1' \approx_l c_2'$$

TINI states that if we take two l-equivalent configurations, and both configurations reduce to final configurations (i.e., configurations for which there are no possible further transitions), then the end configurations are also l-equivalent. We highlight that this statement is much weaker than TSNI: it only states that terminating programs do not leak sensitive data, but makes no statement about non-terminating programs.

As shown by compilers [146, 174], interpreters [78], and libraries [164, 186], TINI is useful for sequential settings. In our case, we show that our IFC language with the sequential scheduling policy SEQ satisfies TINI.

Theorem 4 (Sequential IFC language is TINI)**.** *For any target language λ, $L_{IFC}(\text{SEQ}, \lambda)$ satisfies TINI.*

[4] Though we do not model labeled channels, extending the calculus with such a feature is straightforward, see Section 6.6.

6.4 Isomorphisms and Restrictions

The operational semantics we have defined in the previous section satisfy non-interference by design. We achieve this general statement that works for a large class of languages by having different tasks executing completely isolated from each other, such that every task has its own state. In some cases, this strong separation is desirable, or even necessary. Languages like C provide direct access to memory locations without mechanisms in the language to achieve a separation of the heap. On the other hand, for other languages, this strong isolation of tasks can be undesirable, e.g., for performance reasons. For instance, for the language λ_{ES}, our presentation so far requires a separate heap per task, which is not very practical. Instead, we would like to more tightly couple the integration of the target and IFC languages by reusing existing infrastructure. In the running example, a concrete implementation might use a single global heap. More precisely, instead of using a configuration of the form $\Sigma; \langle \Sigma_1, e_1 \rangle^{i_1}_{l_1}, \langle \Sigma_2, e_2 \rangle^{i_2}_{l_2} \ldots$ we would like a single global heap as in $\Sigma; \Sigma; \langle e_1 \rangle^{i_1}_{l_1}, \langle e_2 \rangle^{i_2}_{l_2}, \ldots$

If the operational rules are adapted naïvely to this new setting, then non-interference can be violated: as we mentioned earlier, shared mutable cells could be used to leak sensitive information. What we would like is a way of characterizing safe modifications to the semantics which preserve non-interference. The intention of our single heap implementation is to permit efficient execution while *conceptually maintaining isolation between tasks* (by not allowing sharing of references between them). This intuition of having a different (potentially more efficient) concrete semantics that behaves like the abstract semantics can be formalized by the following definition:

Definition 11 (Isomorphism of information flow control languages)**.** A language $(\Delta, \hookrightarrow, \varepsilon_l)$ is *isomorphic* to a language $(\Delta', \hookrightarrow', \varepsilon'_l)$ if there exist total functions $f : \Delta \to \Delta'$ and $f^{-1} : \Delta' \to \Delta$ such that $f \circ f^{-1} = id_\Delta$ and $f^{-1} \circ f = id_{\Delta'}$. Furthermore, f and f^{-1} are functorial (e.g., if $x'\ R'\ y'$ then $f(x')\ R\ f(y')$) over both l-equivalences and $\hookrightarrow$.

If we weaken this restriction such that f^{-1} does not have to be functorial over $\hookrightarrow$, we call the language $(\Delta, \hookrightarrow, \varepsilon_l)$ *weakly isomorphic* to $(\Delta', \hookrightarrow', \varepsilon'_l)$.

Providing an isomorphism between the two languages allows us to preserve (termination sensitive or insensitive) non-interference as the following two theorems state.

Theorem 5 (Isomorphism preserves TSNI)**.** *If L is isomorphic to L' and L' satisfies TSNI, then L satisfies TSNI.*

Proof. Shown by transporting configurations and reduction derivations from L to L', applying TSNI, and then transporting the resulting configuration, l-equivalence and multi-step derivation back. □

Only weak isomorphism is necessary for TINI. Intuitively, this is because it is not necessary to back-translate reduction sequences in L' to L; by the definition of TINI, we have both reduction sequences in L by assumption.

Theorem 6 (Weak isomorphism preserves TINI)**.** *If a language L is weakly isomorphic to a language L', and L' satisfies TINI, then L satisfies TINI.*

Proof. Shown by transporting configurations and reduction derivations from L to L', applying TINI and transporting the resulting equivalence back using functoriality of f^{-1} over l-equivalences. □

Unfortunately, an isomorphism is often too strong of a requirement. To obtain an isomorphism with our single heap semantics, we need to mimic the behavior of several heaps with a single actual heap. The interesting cases are when we sandbox an expression and when messages are sent and received. The rule for sandboxing is parametrized by the strategy κ (see Section 6.2), which defines what heap the new task should execute with. We have considered two choices:

- When we sandbox into an empty heap, existing addresses in the sandboxed expression are no longer valid and the task will get stuck (and then removed by I-NOSTEP). Thus, we must rewrite the sandboxed expression so that all addresses point to fresh addresses guaranteed to not occur in the heap. Similarly, sending a memory address should be rewritten.
- When we clone the heap, we have to copy everything reachable from the sandboxed expression and replace all addresses correspondingly. Even worse, the behavior of sending a memory address now depends on whether that address existed at the time the receiving task was sandboxed; if it did, then the address should be rewritten to the existing one.

Isomorphism demands we implement this convoluted behavior, despite our initial motivation of a more efficient implementation.

6.4.1 Restricting the IFC Language

A better solution is to forbid sandboxed expressions as well as messages sent to other tasks to contain memory addresses in the first place. In a statically typed language, the type system could prevent this from happening. In dynamically typed languages such as λ_{ES}, we might restrict the transition for **sandbox** and **send** to only allow expressions without memory addresses.

While this sounds plausible, it is worth noting that we are modifying the IFC language semantics, which raises the question of whether non-interference is preserved. This question can be subtle:

it is easy to remove a transition from a language and invalidate TSNI. Intuitively if the restriction depends on secret data, then a public thread can observe if some other task terminates or not, and from that obtain information about the secret data that was used to restrict the transition. With this in mind, we require semantic rules to get restricted only based on information observable by the task triggering them. This ensures that non-interference is preserved, as the restriction does not depend on confidential information. Below, we give the formal definition of this condition for the abstract IFC language $L_{\text{IFC}}(\alpha, \lambda)$.

Definition 12 (Restricted IFC language)**.** For a family of predicates $\mathscr{P}$ (one for every reduction rule), we call $L^{\mathscr{P}}_{\text{IFC}}(\alpha, \lambda)$ a restricted IFC language if its definition is equivalent to the abstract language $L_{\text{IFC}}(\alpha, \lambda)$, with the following exception: the reduction rules are restricted by adding a predicate $P \in \mathscr{P}$ to the premise of all rules other than I-NOSTEP. Furthermore, the predicate P can depend only on the *erased* configuration $\varepsilon_l(c)$, where l is the label of the first task in the task list and c is the full configuration.

By the following theorem, the restricted IFC language with an appropriate scheduling policy is non-interfering.

Theorem 7. *For any target language λ and family of predicates $\mathscr{P}$, the restricted IFC language $L^{\mathscr{P}}_{IFC}(\text{RR}, \lambda)$ is TSNI. Furthermore, the IFC language $L^{\mathscr{P}}_{IFC}(\text{SEQ}, \lambda)$ is TINI.*

In Appendix B.2 we give an example how this formalism can be used to show non-intereference of an implementation of IFC with a single heap.

6.5 Real World Languages

Our approach can be used to retrofit any language for which we can achieve isolation with information flow control. Unfortunately, controlling the external effects of a real-world language, as to achieve isolation, is language-specific and varies from one language to another.[5] Indeed, even for a single language (e.g., JavaScript), how one achieves isolation may vary according to the language runtime or embedding (e.g., server and browser).

In this section, we describe several implementations and their approaches to isolation. In particular, we describe two JavaScript IFC implementations building on the theoretical foundations of

[5] Though we apply our framework to several real-world languages, it is conceivable that there are languages for which isolation cannot be easily achieved.

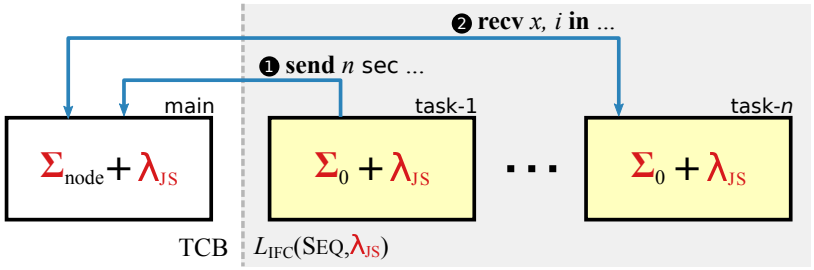

Figure 6.6: This example shows how our trusted monitor (left) is used to mediate communication between two tasks for which IFC is enforced (right).

this work. Then, we consider how our formalism could be applied to the C programming language and connect it to a previous IFC system for Haskell.

6.5.1 JavaScript

JavaScript, as specified by ECMAScript [62], does not have any built-in functionality for I/O. For this language, which we denote by λ_{JS}, the IFC system $L_{\text{IFC}}(\text{RR}, \lambda_{\text{JS}})$ can be implemented by exposing IFC primitives to JavaScript as part of the runtime, and running multiple instances of the JavaScript virtual machine in separate OS-level threads. Unfortunately, this becomes very costly when a system, such as a server-side web application, relies on many tasks.

Luckily, this issue is not unique to our work—browser layout engines also rely on isolating code executing in separate iframes (e.g., according to the same-origin policy). Since creating an OS thread for each iframe is expensive, both the V8 and SpiderMonkey JavaScript engines provide means for running JavaScript code in isolation within a single OS thread, on disjoint sub-heaps. In V8, this unit of isolation is called a *context*; in SpiderMonkey, it is called a *compartment*. (We will use these terms interchangeably.) Each context is associated with a global object, which, by default, implements the JavaScript standard library (e.g., `Object`, `Array`, etc.). Naturally, we adopt contexts to implement our notion of tasks.

When JavaScript is embedded in browser layout engines, or in server-side platforms such as Node.js, additional APIs such as the Document Object Model (DOM) or the file system get exposed as part of the runtime system. These features are exposed by extending the global object, just like the standard library. For this reason, it is easy to modify these systems to forbid external effects when implementing an IFC system, ensuring that important effects can be reintroduced in a safe manner.

Server-side IFC for Node.js: We have implemented $L_{\text{IFC}}(\text{SEQ}, \lambda_{\text{JS}})$ for Node.js in the form of a library, without modifying Node.js or the V8 JavaScript engine. Our implementation provides a library for creating new tasks, i.e., contexts whose global object only contains the standard JavaScript library and our IFC primitives (e.g., **send** and **sandbox**). When mapped to our formal treatment, **sandbox** is defined with $\kappa(\Sigma) = \Sigma_0$, where Σ_0 is the global object corresponding to the standard JavaScript library and our IFC primitives. These IFC operations are mediated by the trusted library code (executing as the main Node.js context), which tracks the state (current label, messages, etc.) of each task. An example for **send/recv** is shown in Figure 6.6. Our system conservatively restricts the kinds of messages that can be exchanged, via **send** (and **sandbox**), to string values. In our formalization, this amounts to restricting the IFC language rule for **send** in the following way:

JS-SEND

$$\frac{\Sigma' = \Sigma\,[i' \mapsto (l', i, v), \Theta] \qquad e = {}^{\text{IT}}\lceil e \rceil \qquad l \sqsubseteq l' \qquad \Sigma\,(i') = \Theta \qquad \mathscr{E}_{\Sigma}\,[\texttt{typeOf}(e)\ \texttt{===}\ \texttt{"string"}] \to \mathscr{E}_{\Sigma}\,[\texttt{true}]}{\Sigma; \langle \Sigma, E[\textbf{send}\ i'\ l'\ v]_I \rangle^i_l, \ldots \hookrightarrow \Sigma'; \alpha_{\text{step}}(\langle \Sigma, E[\langle\rangle]_I \rangle^i_l, \ldots)}$$

Of course, we provide a convenience library which marshals JSON objects to/from strings. We remark that this is not unlike existing message-passing JavaScript APIs, e.g., `postMessage`, which impose similar restrictions as to avoid sharing references between concurrent code.

While the described system implements $L_{\text{IFC}}(\text{SEQ}, \lambda_{\text{JS}})$, applications typically require access to libraries (e.g., the file system library fs) that have external effects. Exposing the Node.js APIs directly to sandboxed tasks is unsafe. Instead, we implement libraries (like a labeled version of fs) as message exchanges between the sandboxed tasks (e.g., task-1 in Figure 6.6) and the main Node.js task that implements the IFC monitor. While this is safer than simply wrapping unsafe objects, which can potentially be exploited to access objects outside the context (e.g., as seen with ADSafe, FBJS, and Caja [194, 123, 124]), adding features such as the fs requires the code in the main task to ensures that labels are properly propagated and enforced. Unfortunately, while imposing such a proof burden is undesirable, this also has to be expected: different language environments expose different libraries for handling external I/O, and the correct treatment of external effects is application specific. We do not extend our formalism to account for the particular interface to the file system, HTTP client, etc., as this is specific to the Node.js implementation and does not generalize to other systems.

Client-side IFC: This work provides the formal basis for the core part of the COWL client-side JavaScript IFC system, described in Chapter 3. Like our Node.js implementation, COWL takes a coarse-grained approach to providing IFC for JavaScript programs. However, COWL's IFC monitor is implemented in the browser layout engine instead (though still leaving the JavaScript engine unmodified).

Furthermore, COWL repurposes existing contexts (e.g., iframes and pages) as IFC tasks, only imposing additional constraints on how they communicate. As with Node.js, at its core, the global object of a COWL task should only contain the standard JavaScript libraries and `postMessage`, whose semantics are modeled by our JS-SEND rule. However, existing contexts have objects such as the DOM, which require COWL to restrict a task's external effects. To this end, COWL mediates any communication (even via the DOM) at the context boundary.

Simply disallowing all the external effects is overly-restricting for real-world applications (e.g., pages typically load images, perform network requests, etc.). In this light, COWL allows safe network communication by associating an implicit label with remote hosts (a host's label corresponds to its origin). In turn, when a task performs a request, COWL's IFC monitor ensures that the task label can flow to the remote origin label. While the external effects of COWL can be formally modeled, we do not model them in our formalism, since, like for the Node.js case, they are specific to this system.

6.5.2 Haskell

This chapter extends the ideas from the LIO coarse-grained IFC system described in Chapter 5 and in [186, 183]. LIO relies on Haskell's type system and monadic encoding of effects to achieve isolation and define the IFC sub-language. Specifically, LIO provides the `LIO` monad as a way of restricting (almost all) side-effects. In the context of our framework, LIO can be understood as follows: the *pure subset* of Haskell is the target language, while the monadic subset of Haskell, operating in the `LIO` monad, is the IFC language.

Unlike the work presented in this chapter, LIO originally associated labels with exceptions, in a similar style to fine-grained systems [188, 86]. In addition to being overly complex, the interaction of exceptions with clearance (which sets an upper bound on the floating label, see Appendix B.3.3) was incorrect: the clearance was restored to the clearance at point of the catch. Furthermore, pure exceptions (e.g., divide by zero) always percolated to trusted code, effectively allowing for denial of service attacks. The insights gained when viewing coarse-grained IFC as presented in this chapter led to a much cleaner, simpler treatment of exceptions, which has now been adopted by LIO, as

described in Chapter 5.

6.5.3 C

C programs are able to execute arbitrary (machine) code, access arbitrary memory, and perform arbitrary system calls. Thus, the confinement of C programs must be imposed by the underlying OS and hardware. For instance, our notion of isolation can be achieved using Dune's hardware protection mechanisms [19], similar to Wedge [19, 24], but using an information flow control policy. Using page tables, a (trusted) IFC runtime could ensure that each task, implemented as a lightweight process, can only access the memory it allocates—tasks do not have access to any shared memory. In addition, ring protection could be used to intercept system calls performed by a task and only permit those corresponding to our IFC language (such as **getLabel** or **send**). Dune's hardware protection mechanism would allow us to provide a concrete implementation that is efficient and relatively simple to reason about, but other sandboxing mechanisms could be used in place of Dune.

In this setting, the combined language of Section 6.2 can be interpreted in the following way: calling from the target language to the IFC language corresponds to invoking a system call. Creating a new task with the **sandbox** system call corresponds to *forking* a process. Using page tables, we can ensure that there will be no shared memory (effectively defining $\kappa(\Sigma) = \Sigma_0$, where Σ_0 is the set of pages necessary to bootstrap a lightweight process). Similarly, control over page tables and protection bits allows us to define a **send** system call that copies pages to our (trusted) runtime queue; and, correspondingly, a **recv** that copies the pages from the runtime queue to the (untrusted) receiver. Since C is not memory safe, conditions on these system calls are meaningless. We leave the implementation of this IFC system for C as future work.

6.6 Extensions and Limitations

While the IFC language presented thus far provides the basic information flow primitives, actual IFC implementations may wish to extend the minimal system with more specialized constructs. For example, COWL provides a labeled version of the XMLHttpRequest (XHR) object, which is used to make network requests. Our system can be extended with constructs such as labeled values, labeled mutable references, clearance, and privileges. For space reasons, we provide details of this, including the soundness proof with the extensions, in Appendix B.3. Here, we instead discuss a limitation of our formalism: the lack of external effects.

Specifically, our embedding assumes that the target language does not have any primitives that

can induce external effects. As discussed in Section 6.5, imposing this restriction can be challenging. Yet, external effects are crucial when implementing more complex real-world applications. For example, code in an IFC browser must load resources or perform XHR to be useful.

Like labeled references, features with external effects must be modeled in the IFC language; we must reason about the precise security implications of features that otherwise inherently leak data. Previous approaches have modeled external effects by internalizing the effects as operations on labeled channels/references [183]. Alternatively, it is possible to model such effects as messages to/from certain labeled tasks, an approach taken by our Node.js implementation. These "special" tasks are trusted with access to the unlabeled primitives that can be used to perform the external effects; since the interface to these tasks is already part of the IFC language, the proof only requires showing that this task does not leak information. Instead of restricting or wrapping unsafe primitives, COWL allow for controlled network communication at the context boundary. (By restricting the default XHR object, for example, COWL allows code to communicate with hosts according to the task's current label.)

6.7 Related Work on Language-level Concurrent IFC

The information flow control system described in this chapter is closely related to the coarse-grained information systems used in operating systems such as Asbestos [63], HiStar [228], and Flume [104], as well as language-based *floating-label IFC systems* such as LIO [186], and Breeze [86], where there is a monotonically increased label associated with threads of execution. Our treatment of termination-sensitive and termination-insensitive interference originates from Smith and Volpano [177, 203].

One information flow control technique designed to handle legacy code is secure multi-execution (SME) [55, 157]. SME runs multiple copies of the program, one per security level, where the semantics of I/O interactions is altered. Bielova et al. [23] use a transition system to describe SME, where the details of the underlying language are hidden. Zanarini et al. [223] propose a novel semantics for programs based on interaction trees [93], which treats programs as black-boxes about which nothing is known, except what can be inferred from their interaction with the environment. Similar to SME, our approach mediates I/O operations; however, our approach only runs the program once.

One of the primary motivations behind this work is the application of information flow control to JavaScript. Previous systems retrofitted JavaScript with fine-grained IFC [79, 99, 78]. While fine-grained IFC can result in fewer false alarms and target legacy code, it comes at the cost of

complexity: the system must accommodate the entirety of JavaScript's semantics [78]. By contrast, coarse-grained approaches to security tend to have simpler implications [221, 50].

The constructs in our IFC language, as well as the behavior of inter-task communication, are reminiscent of distributed systems like Erlang [6]. In distributed systems, isolation is required due to physical constraints; in information flow control, isolation is required to enforce non-interference. Papagiannis et al. [148] built an information flow control system on top of Erlang that shares some similarities to ours. However, they do not take a floating-label approach (processes can find out when sending a message failed due to a forbidden information flow), nor do they provide security proofs.

There is limited work on general techniques for retrofitting arbitrary languages with information flow control. However, one time-honored technique is to define a fundamental calculus for which other languages can be desugared into. Abadi et al. [1] motivate their core calculus of dependency by showing how various previous systems can be encoded in it. Tse and Zdancewic [198], in turn, show how this calculus can be encoded in System F via parametricity. Broberg and Sands [33] encode several IFC systems into Paralocks. However, this line of work is primarily focused on static enforcements.

6.8 Conclusion

In this chapter, we argued that when designing a coarse-grained IFC system, it is better to start with a fully isolated, multi-task system and work one's way back to the model of a single language equipped with IFC. We showed how systems designed this way can be proved non-interferent without needing to rely on details of the target language, and we provided conditions on how to securely refine our formal semantics to consider optimizations required in practice. We connected our semantics to two IFC implementations for JavaScript based on this formalism, explained how our methodology improved an exiting IFC system for Haskell, and proposed an IFC system for C using hardware isolation. By systematically applying ideas from IFC in operating systems to programming languages for which isolation can be achieved, we hope to have elucidated some of the core design principles of coarse-grained, dynamic IFC systems.

Chapter 7

Conclusion

Building secure web applications is notoriously difficult. This is particularly the case because existing programming models make it easy to write insecure code and today's ad-hoc solutions do little to prevent bugs from leading to large scale breaches. This dissertation proposed a principled way to address web application security using dynamic, language-level information flow control.

On the server-side, we presented the design of Hails, a Haskell web framework. Hails makes it easier for developers to build secure applications by separating the security and privacy concerns of an application from the rest of the functionality. Using the MPVC paradigm, Hails developers specify security policies alongside the data models, in the MPs, using a policy specification language. Using dynamic, language-level IFC, Hails ensures that these policies are enforced across the rest of the application components, including the views and controller, which typically contain the bulk of the application code. We implemented Hails and showed that its performance is comparable to widely-deployed web frameworks (e.g., Apache/PHP). We also showed that our system and APIs, as a result of numerous design iterations, are usable by average developers. To our knowledge, Hails is the first IFC system to be used by non-experts to build real-world applications. A recent commercial implementation has shown that the ideas behind Hails are equally applicable to JavaSript.

On the browser-side, we presented COWL, a new confinement system that extends the Web platform with dynamic IFC. COWL allows untrusted code to compute over sensitive data and display results to the user, but prohibits the untrusted code from arbitrarily exfiltrating sensitive data (e.g., by sending it to an untrusted remote party). COWL adopts abstractions from Hails by largely repurposing existing security mechanisms to enforce IFC. Importantly, the system does so by retaining backwards compatibility and interoperability with legacy web applications and services. We presented implementations of COWL for Firefox and Chromium, and several applications on top

of COWL; the system's end-to-end performance degradation, when compared to unsecured applications, is unobservable. COWL has recently been adopted as a new W3C specification and is on track to be standardized.

Building Hails and COWL required addressing a number of technical challenges. For instance, this dissertation addressed the challenge of policy specification in IFC systems, which has historically been a complex, manual, expert-only process. In part, we accomplished this with the design of DC Labels, a new, simple label format that allows labels to reflect the concerns of multiple stakeholders, as is necessary for the web. In part, we accomplished this by designing applications around the MPVC paradigm, by tying policies with data models, and by providing a declarative policy DSL to automate labeling.

This dissertation also bridged the gap between formal IFC models and implementations. In particular, we presented a new design point for dynamic language-level IFC called LIO (and its generalization, IFC-Inside). LIO incorporates ideas from practical IFC operating systems, at the language-level, to support complex, real-world features, such as exceptions, policy inspection, monitor recovery, and concurrency. This dissertation showed that, while LIO and its generalization are more flexible and permissive than most existing IFC systems, the systems still provide strong security guarantees (some of which were proved in Coq). To our knowledge, LIO is the first flexible, dynamic, language-level IFC system to satisfy termination-sensitive noninterference.

Finally, we showed that, when used together, Hails (with LIO) and COWL not only provide end-to-end security to modern web applications, but also open up the possibility to build new kinds of applications (e.g., privacy-preserving web platforms, secure third-party mashups, password managers as web sites, etc.), previously not possible because of security concerns.

Bibliography

[1] ABADI, M., BANERJEE, A., HEINTZE, N., AND RIECKE, J. A Core Calculus of Dependency. In *Symposium on Principles of Programming Languages* (1999), ACM. 3, 120, 154, 181

[2] ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. A Calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems 15*, 4 (Oct. 1993), 706–734. 52

[3] AGAT, J. Transforming out timing leaks. In *Symposium on Principles of Programming Languages* (2000), ACM. 129

[4] AGTEN, P., VAN ACKER, S., BRONDSEMA, Y., PHUNG, P. H., DESMET, L., AND PIESSENS, F. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC* (2012). 83

[5] AKHAWE, D., LI, F., HE, W., SAXENA, P., AND SONG, D. Data-Confined HTML5 Applications. In *ESORICS* (2013). 76, 82

[6] ARMSTRONG, J. *Making reliable distributed systems in the presence of sodware errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003. 181

[7] ASKAROV, A., HUNT, S., SABELFELD, A., AND SANDS, D. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Proc. of the European Symp. on Research in Computer Security (ESORICS)* (2008), Springer-Verlag. 129, 172

[8] ASKAROV, A., AND SABELFELD, A. Catch Me if You Can: Permissive Yet Secure Error Handling. In *Programming Languages and Analysis for Security* (2009), ACM. 156

[9] ASKAROV, A., AND SABELFELD, A. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *Computer Security Foundations Symposium* (2009), IEEE Computer Society. 140

[10] ATKEY, R. Parameterised notions of computation. *Journal of Functional Programming 19*, 3-4 (2009). 155

[11] AUSTIN, T. H., AND FLANAGAN, C. Efficient Purely-Dynamic Information Flow Analysis. In *Workshop on Programming Languages and Analysis for Security* (2009), ACM. 3, 106, 134, 140, 153

[12] AUSTIN, T. H., AND FLANAGAN, C. Permissive Dynamic Information Flow Analysis. In *Workshop on Programming Languages and Analysis for Security* (June 2010), ACM. 3, 106, 140

[13] AUSTIN, T. H., KNOWLES, K., AND FLANAGAN, C. Typed Faceted Values for Secure Information Flow in Haskell. *MIT Technical Report UCSC-SOE-14-07* (2014). 50

[14] AZEVEDO DE AMORIM, A., COLLINS, N., DEHON, A., DEMANGE, D., HRIŢCU, C., PICHARDIE, D., PIERCE, B. C., POLLACK, R., AND TOLMACH, A. A verified information-flow architecture. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 165–178. 2

[15] BADGER, L., STERNE, D. F., SHERMAN, D. L., WALKER, K. M., AND HAGHIGHAT, S. A. Practical Domain and Type Enforcement for UNIX. In *Security and Privacy* (1995). 81

[16] BARTH, A. The Web Origin Concept. Tech. rep., IETF, Dec. 2011. `https://tools.ietf.org/html/rfc6454`. 10, 17, 31, 58

[17] BARTH, A., JACKSON, C., AND MITCHELL, J. Securing frame communication in browsers. *Communications of the ACM 52*, 6 (2009), 83–91. 59

[18] BARTHE, G., REZK, T., RUSSO, A., AND SABELFELD, A. Security of Multithreaded Programs by Compilation. In *ESORICS* (2007). 171

[19] BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe User-level Access to Privileged CPU Features. In *OSDI* (2012). 179

[20] BELL, D. E., AND PADULA, L. L. Secure Computer System: Unified Exposition and Multics Interpretation. Tech. Rep. MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976. 157

[21] BHARGAVAN, K., DELIGNAT-LAVAUD, A., AND MAFFEIS, S. Language-based Defenses Against Untrusted Browser Origins. In *Proceedings of the 22nd USENIX Conference on Security* (2013), USENIX, pp. 653–670. 47, 83

[22] BIBA, K. J. Integrity Considerations for Secure Computer Systems. Tech. Rep. ESD-TR-76-372, MITRE Corp., April 1977. 110

[23] BIELOVA, N., DEVRIESE, D., MASSACCI, F., AND PIESSENS, F. Reactive non-interference for a browser model. In *NSS* (2011). 180

[24] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting Applications into Reduced-privilege Compartments. In *NSDI* (2008). 179

[25] BLANKSTEIN, A., AND FREEDMAN, M. J. Automating isolation and least privilege in web services. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 133–148. 51

[26] BLAZE, M., FEIGENBAUM, J., IOANNIDIS, J., AND KEROMYTIS, A. The KeyNote Trust-Management System Version 2. RFC 2704 (Informational), Sept. 1999. 52

[27] BLAZE, M., FEIGENBAUM, J., AND LACY, J. Decentralized trust management. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on* (1996), pp. 164–173. 52

[28] BOMBERGER, A. C., FRANTZ, A. P., FRANTZ, W. S., HARDY, A. C., HARDY, N., LANDAU, C. R., AND SHAPIRO, J. S. The KeyKOS Nanokernel Architecture. In *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (April 1992). 103

[29] BONEH, D., DING, X., TSUDIK, G., AND WONG, C. A method for fast revocation of public key certificates and security capabilities. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10* (2001), USENIX Association, pp. 22–22. 96

[30] BORTZ, A., AND BONEH, D. Exposing Private Information by Timing Web Applications. In *Proceedings of the 16th World Wide Web* (2007), ACM, pp. 621–628. 20

[31] BOUDOL, AND CASTELLANI. Noninterference for Concurrent Programs. In *ICALP* (2001). 170

[32] BROBERG, N., AND SANDS, D. Flow locks: Towards a core calculus for dynamic flow policies. In *Programming Languages and Systems*. Springer, 2006, pp. 180–196. 3

[33] BROBERG, N., AND SANDS, D. Paralocks: role-based information flow control and beyond. In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2010), POPL '10, pp. 431–444. 103, 181

[34] BUIRAS, P., LEVY, A., STEFAN, D., RUSSO, A., AND MAZIÈRES, D. A Library for Removing Cache-Based Attacks in Concurrent Information Flow Systems. In *TGC* (2013). 7, 8, 168

[35] BUIRAS, P., STEFAN, D., AND RUSSO, A. On Dynamic Flow-sensitive Floating-Label Systems. In *Computer Security Foundations Symposium (CSF)* (July 2014), IEEE. 7, 126, 127

[36] BUIRAS, P., VYTINIOTIS, D., AND RUSSO, A. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (2015), ACM, pp. 289–301. 6, 10

[37] CARLINI, N., FELT, A. P., AND WAGNER, D. An evaluation of the Google Chrome extension security architecture. In *USENIX Security* (2012). 76

[38] CHANDRA, R., GUPTA, P., AND ZELDOVICH, N. Separating web applications from user data storage with BSTORE. In *Proceedings of the 2010 USENIX conference on Web application development* (2010), pp. 1–1. 52

[39] CHEN, E. Y., GORBATY, S., SINGHAL, A., AND JACKSON, C. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In *Web 2.0 Security and Privacy* (2012). 60, 80

[40] CHENG, R., SCOTT, W., ELLENBOGEN, P., HOWELL, J., AND ANDERSON, T. Radiatus: Strong User Isolation for Scalable Web Applications. *University of Washington Technical Report* (2014). 51

[41] CHENG, W., PORTS, D., SCHULTZ, D., POPIC, V., BLANKSTEIN, A., COWLING, J., CURTIS, D., SHRIRA, L., AND LISKOV, B. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012). 30

[42] CHLIPALA, A. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (2010), USENIX, pp. 105–118. 49

[43] CHODOROW, K., AND DIROLF, M. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2010. 20

[44] CHONG, S., LIU, J., MYERS, A. C., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. Secure Web Applications via Automatic Partitioning. In *ACM Symposium on Operating Systems Principles* (Oct. 2007), pp. 31–44. 49

[45] CHONG, S., VIKRAM, K., AND MYERS, A. C. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proc. USENIX Security Symposium* (Aug. 2007), pp. 1–16. 49

[46] CHUNG TSAI, T., RUSSO, A., AND HUGHES, J. A Library for Secure Multi-threaded Information Flow in Haskell. In *Computer Security Foundations Symp., 2007. CSF '07. 20th IEEE* (July 2007), pp. 187–202. 155

[47] CODEMIRROR. CodeMirror, November 2015. `http://codemirror.net/`. 34

[48] CRARY, K., KLIGER, A., AND PFENNING, F. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming 15*, 2 (March 2005). 154

[49] CROCKFORD, D. Making JavaScript Safe for Advertising. `http://adsafe.org/`. 46

[50] DE GROEF, W., DEVRIESE, D., NIKIFORAKIS, N., AND PIESSENS, F. FlowFox: a web browser with flexible and precise information flow control. In *CCS* (2012). 2, 9, 10, 82, 181

[51] DENNING, D. E. A Lattice Model of Secure Information Flow. *Communications of the ACM 19*, 5 (May 1976), 236–243. 3, 90, 112, 164

[52] DENNING, D. E., AND DENNING, P. J. Certification of programs for secure information flow. *Communications of the ACM 20*, 7 (1977), 504–513. 3, 106, 114

[53] DEPARTMENT OF DEFENSE. *Trusted Computer System Evaluation Criteria (Orange Book)*, DoD 5200.28-STD ed., December 1985. 98, 157

[54] DETREVILLE, J. Binder, a Logic-Based Security Language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy* (May 2002), IEEE Computer Society Press, pp. 105–113. 52

[55] DEVRIESE, D., AND PIESSENS, F. Noninterference through secure multi-execution. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 109–124. 3, 82, 98, 180

[56] DEVRIESE, D., AND PIESSENS, F. Information flow enforcement in monadic libraries. In *Proc. of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (2011), ACM. 140, 155

[57] DISQUS. Disqus, November 2015. https://disqus.com/. 35

[58] DOCKER. Introduction to Container Security, March 2015. https://d3oypxn00j2a10.cloudfront.net/assets/img/Docker%20Security/WP_Intro_to_container_security_03.20.2015.pdf. 46

[59] DOCTOROW, C. United website breach let fliers see each others' private data, January 2015. https://boingboing.net/2015/01/28/united-website-breach-let-flie.html. 1, 4, 15, 43

[60] DUSSEAULT, L., AND SNELL, J. M. PATCH Method for HTTP. Tech. rep., IETF, March 2010. https://tools.ietf.org/html/rfc5789. 37

[61] DYCKHOFF, R. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic* (1992), 795–807. 102

[62] ECMA INTERNATIONAL. ECMAScript Language Specification. http://www.w3.org/TR/workers, 2011. 176

[63] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and Event Processes in the Asbestos Operating System. In *OSDI* (2005). 2, 5, 7, 66, 103, 116, 154, 180

[64] FACEBOOK. FBJS (Facebook JavaScript). http://developers.facebook.com/docs/fbjs/. 46

[65] FELLEISEN, M., AND HIEB, R. The Revised Report on the Syntactic Theories of Sequential Control and State. *TCS 103*, 2 (1992). 163

[66] FELT, A. P., FINIFTER, M., WEINBERGER, J., AND WAGNER, D. Diesel: applying privilege separation to database access. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), ACM, pp. 416–422. 51

[67] FELTEN, E. W., AND SCHNEIDER, M. A. Timing Attacks on Web Privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security* (2000), ACM, pp. 25–32. 20

[68] FP COMPLETE. School of Haskell, November 2015. https://www.fpcomplete.com/school. 34

[69] FRIEDMAN, D. P., AND WISE, D. S. The impact of applicative programming on multiprocessing. In *International Conference on Parallel Processing* (1976). 138

[70] GALLIER, J. Constructive logics Part I: A tutorial on proof systems and typed λ-calculi. *Theoretical computer science 110*, 2 (1993), 249–339. 102

[71] GARRETT, M. Container Security with SELinux and CoreOS, September 2015. https://coreos.com/blog/container-security-selinux-coreos/. 46

[72] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIÈRES, D., MITCHELL, J., AND RUSSO, A. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation* (October 2012), USENIX. 3, 6, 13, 23, 43, 44, 46, 127, 131, 138, 140

[73] GOGUEN, J., AND MESEGUER, J. Security policies and security Models. In *Proc of IEEE Symp. on Security and Privacy* (April 1982), I. C. S. Press, Ed., pp. 11–20. 3, 88, 110, 162, 169

[74] GOOGLE. Google Code Prettify, September 2012. http://code.google.com/p/google-code-prettify/. 32

[75] GOOGLE. Google Caja. A source-to-source translator for securing JavaScript-based web content. http://code.google.com/p/google-caja/, 2013. 83

[76] GUNTER, C., AND JIM, T. Generalized certificate revocation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2000), ACM, pp. 316–329. 97

[77] HARRISON, W. L. Achieving information flow security through precise control of effects. In *Computer Security Foundations Workshop* (2005), IEEE Computer Society. 154

[78] HEDIN, D., BIRGISSON, A., BELLO, L., AND SABELFELD, A. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC* (2014). 3, 4, 9, 10, 82, 160, 172, 180, 181

[79] HEDIN, D., AND SABELFELD, A. Information-Flow Security for a Core of JavaScript. In *Computer Security Foundations Symposium* (2012), IEEE Computer Society. 106, 156, 163, 180

[80] HEDIN, D., AND SANDS, D. Noninterference in the presence of non-opaque pointers. In *Computer Security Foundations Workshop* (2006), IEEE Computer Society. 132

[81] HEINTZE, N., AND RIECKE, J. G. The SLam calculus: programming with secrecy and integrity. In *Symposium on Principles of Programming Languages* (1998), ACM. 3, 153

[82] HEULE, S., STEFAN, D., YANG, E., MITCHELL, J. C., AND RUSSO, A. IFC Inside: Retrofitting Languages with Dynamic Information Flow Control. In *Proceedings of the 4th Conference on Principles of Security and Trust* (April 2015), Springer. 3, 7, 8, 10, 20, 29, 41, 159

[83] HICKS, M., TSE, S., HICKS, B., AND ZDANCEWIC, S. Dynamic Updating of Information-Flow Policies. In *Foundations of Computer Security* (2005), p. 7. 97

[84] HOWARD, W. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 479–490. 102

[85] HOWELL, J., PARNO, B., AND DOUCEUR, J. R. Embassies: Radically Refactoring the Web. In *NSDI* (2013). 83

[86] HRIŢCU, C., GREENBERG, M., KAREL, B., PIERCE, B. C., AND MORRISETT, G. All Your IFCException Are Belong To Us. In *Security and Privacy* (2013). 6, 7, 8, 30, 66, 106, 108, 112, 116, 128, 131, 134, 140, 141, 142, 153, 157, 162, 163, 165, 178, 180, 215, 216, 217

[87] HUGHES, J. Generalising monads to arrows. *Science of Computer Programming 37*, 1–3 (2000), 67–111. 154

[88] HUILGOL, M. Facebook's Latest Security Vulnerability Allows Third Party Applications to Delete Facebook Pages Permanently, August 2015. `http://techpp.com/2015/08/27/facebook-pages-security-vulnerability/`. 1, 4, 15

[89] HUNT, T. Searching the Snapchat data breach with "Have I been pwned?". http://www.troyhunt.com/2014/01/searching-snapchat-data-breach-with.html, Jan. 2014. 1

[90] INGRAM, L., AND WALFISH, M. TreeHouse: JavaScript sandboxes to help web developers help themselves. In *USENIX ATC* (2012). 83

[91] ISAACS, S. Microsoft Web Sandbox. http://www.websandbox.org/. 46

[92] ISO, W. 9241-11. Ergonomic requirements for office work with visual display terminals (VDTs). *The international organization for standardization* (1998). 45

[93] JACOBS, B., AND RUTTEN, J. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS 62* (1997). 180

[94] JASKELIOFF, M., AND RUSSO, A. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics* (June 2011), LNCS, Springer Verlag. 101

[95] JONES, S. P., GORDON, A., AND FINNE, S. Concurrent Haskell. In *POPL* (1996). 215

[96] KAINDA, R., FLECHAIS, I., AND ROSCOE, A. Security and usability: Analysis and evaluation. In *Proceedings of the 5th International Conference on Availability, Reliability, and Security* (2010), CPS, pp. 275–282. 42, 43

[97] KASHYAP, V., WIEDERMANN, B., AND HARDEKOPF, B. Timing-and termination-sensitive secure information flow: Exploring a new approach. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 413–428. 98

[98] KERSCHBAUMER, C. Faster Content Security Policy (CSP). https://blog.mozilla.org/security/2014/09/10/faster-csp/, 2014. 78

[99] KERSCHBAUMER, C., HENNIGAN, E., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. ConDOM: Containing the DOM for Safe Browsing. Tech. Rep. 12-01, Department of Information and Computer Science, Univ. of California Irvine, October 2012. 180

[100] KHANDELWAL, S. Facebook vulnerability allows hacker to delete any photo album, February 2015. https://thehackernews.com/2015/02/hacking-facebook-photo-album.html. 1, 4, 15, 43

[101] KOTCHER, R., PEI, Y., JUMDE, P., AND JACKSON, C. Cross-origin pixel stealing: timing attacks using CSS filters. In *CCS* (2013). 80

[102] KOVACH, S. Nearly 7 Million Dropbox Passwords Have Been Hacked, October 2014. http://www.businessinsider.com/dropbox-hacked-2014-10. 14

[103] KROHN, M. Building Secure High-Performance Web Services with OKWS. In *Proceedings of the 2014 USENIX Annual Technical Conference* (2004), USENIX, pp. 185–198. 51

[104] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information Flow Control for Standard OS Abstractions. In *Proc. of the 21st Symp. on Operating Systems Principles* (October 2007). 2, 7, 30, 103, 106, 112, 113, 154, 180

[105] KROHN, M., YIP, A., BRODSKY, M., MORRIS, R., AND WALFISH, M. A World Wide Web Without Walls. In *Proceedings of the 6th ACM Workshop on Hot Topics in Networking* (2007), ACM. 51

[106] KUMAR, M. jQuery Official Website Compromised To Serve Malware. http://thehackernews.com/2014/09/jquery-official-website-compromised-to.html, Sept. 2014. 9

[107] LAMPSON, B. W. A note on the confinement problem. *Communications of the ACM 16*, 10 (1973), 613–615. 97, 128

[108] LANDWEHR, C. E. Formal Models for Computer Security. *Computing Survels 13*, 3 (September 1981), 247–278. 154

[109] LAROCHELLE, D., AND EVANS, D. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of theUSENIX Security Symposium* (August 2001). 17, 32

[110] LATIF, L. Github suffers a Ruby on Rails public key vulnerability, March 2012. http://www.theinquirer.net/inquirer/news/2157093/github-suffers-ruby-rails-public-key-vulnerability. 1, 43

[111] LI, N., AND MITCHELL, J. C. RT: A Role-based Trust-management Framework. In *The Third DARPA Information Survivability Conference and Exposition (DISCEX III)* (Apr. 2003), IEEE Computer Society Press. 52

[112] LI, N., WINSBOROUGH, W. H., AND MITCHELL, J. C. Distributed Credential Chain Discovery in Trust Management. *Journal of Computer Security 11*, 1 (Feb. 2003), 35–86. 52

[113] LI, P., AND ZDANCEWIC, S. Practical Information-flow Control in Web-Based Information Systems. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations* (2005), IEEE Computer Society. 52

[114] LI, P., AND ZDANCEWIC, S. Encoding Information Flow in Haskell. In *Computer Security Foundations Workshop* (2006), IEEE Computer Society. 3, 154

[115] LI, P., AND ZDANCEWIC, S. Arrows for secure information flow. In *Theoretical Computer Science* [116]. 101

[116] LI, P., AND ZDANCEWIC, S. Arrows for secure information flow. *Theoretical Computer Science 411*, 19 (2010). 107, 143, 169, 194

[117] LIANG, S., HUDAK, P., AND JONES, M. Monad transformers and modular interpreters. In *Symposium on Principles of programming languages* (1995), ACM. 114

[118] LIFTON, F. Advancing Docker Security: Docker 1.4.0 and 1.3.3 Releases, December 2014. https://blog.docker.com/2014/12/advancing-docker-security-docker-1-4-0-and-1-3-3-releases/. 46

[119] LIU, J., GEORGE, M. D., VIKRAM, K., QI, X., WAYE, L., , AND MYERS, A. C. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles* (2009). 2, 7, 29, 52

[120] LOURENÇO, L., AND CAIRES, L. Information flow analysis for valued-indexed data security compartments. In *Trustworthy Global Computing*. Springer, 2014, pp. 180–198. 52

[121] LOURENÇO, L., AND CAIRES, L. Dependent information flow types. In *Proceedings of the 42nd Symposium on Principles of Programming Languages* (2015), ACM, pp. 317–328. 52

[122] MACFARLANE, J. Pandoc:a universal document converter. http://johnmacfarlane.net/pandoc/. 33, 34

[123] MAFFEIS, S., MITCHELL, J. C., AND TALY, A. Object capabilities and isolation of untrusted web applications. In *SP* (2010). 177

[124] MAFFEIS, S., AND TALY, A. Language-based isolation of untrusted Javascript. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE* (2009), pp. 77–91. 46, 177

[125] MATTHEWS, J., AND FINDLER, R. B. Operational Semantics for Multi-language Programs. In *POPL* (2007). 10, 160, 163, 167, 168

[126] MAYER, J., AND MITCHELL, J. Third-Party Web Tracking: Policy and Technology. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 413–427. 14

[127] MILLER, M., SAMUEL, M., LAURIE, B., AWAD, I., AND STAY, M. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`, June 2008. 46

[128] MILLER, M., AND SHAPIRO, J. Paradigm regained: Abstraction mechanisms for access control. *Advances in Computing Science–ASIAN 2003* (2003), 224–242. 97

[129] MILLER, M. S. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006. 74

[130] MILLER, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. 97, 103, 138

[131] MILLER, M. S., AND SHAPIRO, J. S. Paradigm regained: Abstraction mechanisms for access control. In *ASIAN* (2003). 74

[132] MILLER, M. S., YEE, K.-P., AND SHAPIRO, J. Capability Myths Demolished. Tech. Rep. SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. `http://zesty.ca/capmyths/usenix.pdf`. 70

[133] MINT. Mint. `http://www.mint.com/`, 2013. 56

[134] MITCHELL, N. *HLint Manual*. `http://community.haskell.org/~ndm/darcs/hlint/hlint.htm`. 32

[135] MOITOZO, S. `http://www.geekwisdom.com/js/passwordmeter.js`, 2006. 78

[136] MONTAGU, B., PIERCE, B. C., AND POLLACK, R. A Theory of Information-Flow Labels. In *CSF* (June 2013). 5, 6, 29, 81, 101

[137] MOORE, S., ASKAROV, A., AND CHONG, S. Precise enforcement of progress-sensitive security. In *CCS* (2012). 171

[138] MORGENSTERN, J., AND LICATA, D. R. Security-typed programming within dependently typed programming. In *International Conference on Functional Programming* (2010), ACM. 155

[139] MORT BAY CONSULTING. Jetty WebServer, March 2012. http://jetty.codehaus.org/jetty/. 40

[140] MOSBERGER, D., AND JIN, T. httperf-a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review* (1998), 31–37. 40

[141] MOZILLA. Add-on Builder and SDK. https://addons.mozilla.org/en-US/developers/docs/sdk/, 2013. 76

[142] MOZILLA. Persona, October 2015. https://developer.mozilla.org/en-US/Persona. 18

[143] MYERS, A., AND LISKOV, B. Complete, safe information flow with decentralized labels. In *IEEE Security and Privacy, 1998.* (1998), IEEE, pp. 186–197. 102

[144] MYERS, A. C., AND LISKOV, B. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles* (1997). 2, 5, 6, 7, 15, 81, 87, 88, 95, 102, 107, 109, 112, 113, 140, 150, 217

[145] MYERS, A. C., AND LISKOV, B. Protecting Privacy using the Decentralized Label Model. *ACM Trans. on Computer Systems 9*, 4 (October 2000), 410–442. 3, 7, 30, 102, 112, 116, 117, 156, 217

[146] MYERS, A. C., ZHENG, L., ZDANCEWIC, S., CHONG, S., AND NYSTROM, N. Jif: Java Information Flow. Software release. Located at http://www.cs.cornell.edu/jif, 2001. 119, 172

[147] PAPADIMITRIOU, C. *Complexity Theory*. Addison Wesley, 1993. 91

[148] PAPAGIANNIS, I., MIGLIAVACCA, M., EYERS, D. M., SH, B., BACON, J., AND PIETZUCH, P. Enforcing User Privacy in Web Applications using Erlang. In *W2SP* (2010). 181

[149] PARKER, J. L. LMonad: Information Flow Control for Haskell Web Applications. Master's thesis, University of Maryland, College Park, 2014. 6

[150] PETAZZONI, J. Containers & Docker: How Secure Are They?, August 2013. https://blog.docker.com/2013/08/containers-docker-how-secure-are-they/. 46

[151] PEYTON JONES, S. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering theories of software construction 180* (2001), 47. 131

[152] POPA, R. A., REDFIELD, C., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd Symposium on Operating Systems Principles* (2011), ACM, pp. 85–100. 51

[153] POPA, R. A., STARK, E., HELFER, J., VALDEZ, S., ZELDOVICH, N., KAASHOEK, M. F., AND BALAKRISHNAN, H. Building web applications on top of encrypted data using Mylar. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation* (2014), USENIX, pp. 157–172. 51

[154] POTTIER, F., AND SIMONET, V. Information Flow Inference for ML. In *Symposium on Principles of Programming Languages* (2002), ACM. 3, 107

[155] PRESTON-WERNER, T. Public Key Security Vulnerability and Mitigation, March 2012. https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation. 4, 15, 43

[156] QUANTCAST. jQuery Usage Statistics: Websites using jQuery. http://trends.builtwith.com/javascript/jQuery, 2015. 9, 55

[157] RAFNSSON, W., AND SABELFELD, A. Secure multi-execution: fine-grained, declassification-aware, and transparent. In *CSF* (2013). 180

[158] RECORDON, D., AND REED, D. OpenID 2.0: a platform for user-centric identity management. In *Proceedings of the 2nd ACM workshop on Digital Identity Management* (2006), ACM, pp. 11–16. 18

[159] REDELL, D., AND FABRY, R. Selective revocation of capabilities. In *Proceedings of the International Workshop on Protection in Operating Systems* (1974), pp. 192–209. 97

[160] REIS, C., DUNAGAN, J., WANG, H. J., DUBROVSKY, O., AND ESMEIR, S. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *TWEB 1*, 3 (Sept. 2007). 83

[161] REISG, J. Dromaeo: JavaScript performance testing. http://dromaeo.com/, 2014. 78

[162] RONDON, P. M., KAWAGUCI, M., AND JHALA, R. Liquid types. In *Proceedings of the 29th Conference on Programming Language Design and Implementation* (2008), ACM, pp. 159–169. 48, 113

[163] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: Practical Fine-grained Decentralized Information Flow Control. In *Proceedings of the 30th Conference on Programming Language Design and Implementation* (2009), ACM, pp. 63–74. 3, 5, 7, 50, 108, 109, 155

[164] RUSSO, A., CLAESSEN, K., AND HUGHES, J. A library for light-weight information-flow security in Haskell. In *Haskell '08: Proc. of the first ACM SIGPLAN symposium on Haskell* (2008), pp. 13–24. 10, 107, 143, 155, 169, 172

[165] RUSSO, A., AND SABELFELD, A. Securing Interaction between threads and the scheduler. In *CSFW* (2006). 171

[166] RUSSO, A., AND SABELFELD, A. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Computer Security Foundations Symposium* (2010), IEEE Computer Society. 126

[167] SABELFELD, A., AND MYERS, A. C. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications 21*, 1 (January 2003). 2, 116, 124, 148

[168] SABELFELD, A., AND RUSSO, A. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Conference on Perspectives of System Informatics* (2009), Springer. 107, 140, 155

[169] SABELFELD, A., AND SANDS, D. Dimensions and Principles of Declassification. In *Proc. IEEE Computer Security Foundations Workshop* (June 2005), pp. 255–269. 92

[170] SALTZER, J. H., AND SCHROEDER, M. D. The Protection of Information in Computer Systems. *Proc. of the IEEE 63*, 9 (September 1975), 1278–1308. 36, 97, 128, 130

[171] SCHOEPE, D., HEDIN, D., AND SABELFELD, A. SeLINQ: tracking information across application-database boundaries. In *Proceedings of the 19th International Conference on Functional Programming* (2014), ACM, pp. 25–38. 52

[172] SCHULTZ, D., AND LISKOV, B. IFDB: decentralized information flow control for databases. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 43–56. 52

[173] SHACKEL, B. Usability-context, framework, definition, design and evaluation. *Human factors for informatics usability* (1991), 21–37. 45

[174] SIMONET, V. The Flow Caml System. Software release. Located at http://cristal.inria.fr/~simonet/soft/flowcaml/, 2003. 116, 119, 156, 172

[175] SINATRA. Sinatra, September 2012. http://www.sinatrarb.com/. 39

[176] SIRER, E., DE BRUIJN, W., REYNOLDS, P., SHIEH, A., WALSH, K., WILLIAMS, D., AND SCHNEIDER, F. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 249–264. 49, 81

[177] SMITH, G., AND VOLPANO, D. Secure Information Flow in a Multi-threaded Imperative Language. In *POPL* (1998). 180

[178] SON, S., AND SHMATIKOV, V. The postman always rings twice: Attacking and defending postMessage in HTML5 websites. In *NDSS* (2013). 82

[179] STARK, E., HAMBURG, M., AND BONEH, D. Symmetric cryptography in JavaScript. In *ACSAC* (2009). 79

[180] STEEL, E., AND FOWLER, G. Facebook in Privacy Breach. *The Wall Street Journal 18* (October 2010). http://online.wsj.com/article/SB10001424052702304772804575558484075236968.html. 14

[181] STEFAN, D. Confinement with Origin Web Labels, October 2015. http://www.w3.org/TR/2015/WD-COWL-20151015/. 10, 25, 26, 28, 31, 57, 84

[182] STEFAN, D., BUIRAS, P., YANG, E. Z., LEVY, A., TEREI, D., RUSSO, A., AND MAZIÈRES, D. Eliminating Cache-based Timing Attacks with Instruction-based Scheduling. In *The 18th European Symposium on Research in Computer Security (ESORICS) 2013* (2013). 8, 42, 168

[183] STEFAN, D., RUSSO, A., BUIRAS, P., LEVY, A., MITCHELL, J. C., AND MAZIÈRES, D. Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems. In *The 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)* (2012). 3, 7, 10, 20, 29, 30, 41, 66, 137, 150, 154, 169, 170, 171, 178, 180, 215

[184] STEFAN, D., RUSSO, A., MAZIÈRES, D., AND MITCHELL, J. C. Disjunction Category Labels. In *NordSec* (2011). 3, 5, 6, 29, 65, 81, 86, 113, 121

[185] STEFAN, D., RUSSO, A., MAZIÈRES, D., AND MITCHELL, J. C. Flexible Dynamic Information Flow Control in the Presence of Exceptions. *Journal of Functional Programming* (2012). Under revision. 3, 7, 41, 43

[186] STEFAN, D., RUSSO, A., MITCHELL, J. C., AND MAZIÈRES, D. Flexible Dynamic Information Flow Control in Haskell. In *Symposium on Haskell* (2011), ACM SIGPLAN. 3, 7, 66, 70, 105, 108, 128, 150, 165, 169, 172, 178, 180, 200, 215, 216, 217

[187] STEFAN, D., RUSSO, A., MITCHELL, J. C., AND MAZIÈRES, D. Flexible Dynamic Information Flow Control in Haskell. In *Symposium on Haskell* [186]. 29, 30, 41, 98, 102

[188] STEFAN, D., RUSSO, A., MITCHELL, J. C., AND MAZIÈRES, D. Flexible Dynamic Information Flow Control in the Presence of Exceptions. *Arxiv preprint arXiv:1207.1457* (2012). 108, 134, 141, 156, 157, 162, 163, 178

[189] STEFAN, D., YANG, E. Z., MARCHENKO, P., RUSSO, A., HERMAN, D., KARP, B., AND MAZIÈRES, D. Protecting Users by Confining JavaScript with COWL. In *Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2014), USENIX. 3, 6, 9, 25, 26, 28, 31, 39, 54, 161, 163

[190] STOUGHTON, A. Access Flow: A Protection Model which Integrates Access Control and Information Flow. In *Symposium on Security and Privacy* (1981), IEEE Computer Society. 128

[191] STOUGHTON, A., JOHNSON, A., BELLER, S., CHADHA, K., CHEN, D., FONER, K., AND ZHIVICH, M. You Sank My Battleship!: A Case Study in Secure Programming. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security* (2014), ACM, p. 2. 5

[192] SULZMANN, M., DUCK, G. J., PEYTON JONES, S., AND STUCKEY, P. J. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming 17*, 1 (2007). 154

[193] TALY, A., ERLINGSSON, Ú., MITCHELL, J. C., MILLER, M. S., AND NAGRA, J. Automated Analysis of Security-Critical JavaScript APIs. In *Proceedings of theIEEE Symposium on Security and Privacy* (2011). 47

[194] TALY, A., MITCHELL, J. C., MILLER, M. S., AND NAGRA, J. Automated analysis of security-critical javascript apis. In *SP* (2011). 177

[195] TEAM, H. The hails [Haskell] package. http://hackage.haskell.org/package/hails. 24

[196] TER LOUW, M., PHUNG, P. H., KRISHNAMURTI, R., AND VENKATAKRISHNAN, V. N. SafeScript: JavaScript Transformation for Policy Enforcement. In *Secure IT Systems* (2013). 83

[197] TEREI, D., MARLOW, S., JONES, S. P., , AND MAZIÈRES, D. Safe Haskell. In *Proceedings of the 5th Symposium on Haskell* (September 2012). 30, 106, 118

[198] TSE, S., AND ZDANCEWIC, S. Translating dependency into parametricity. In *Proc. of the Ninth ACM SIGPLAN International Conference on Functional Programming* (2004), ACM. 154, 181

[199] VAN ACKER, S., DE RYCK, P., DESMET, L., PIESSENS, F., AND JOOSEN, W. WebJail: least-privilege integration of third-party components in web mashups. In *ACSAC* (2011). 83

[200] VAN KESTEREN, A. Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/, 2012. 58

[201] VANDEBOGART, S., EFSTATHOPOULOS, P., KOHLER, E., KROHN, M., FREY, C., ZIEGLER, D., KAASHOEK, F., MORRIS, R., AND MAZIÈRES, D. Labels and Event Processes in the Asbestos Operating System. *ACM Trans. on Computer Systems 25*, 4 (December 2007), 11:1–43. A version appeared in *Proc. of the 20th ACM Symp. on Operating System Principles*, 2005. 106

[202] VIBBER, B. CSRF token-stealing attack (user.tokens). https://bugzilla.wikimedia.org/show_bug.cgi?id=34907, 2014. 80

[203] VOLPANO, D., AND SMITH, G. Eliminating Covert Flows with Minimum Typings. In *CSFW* (1997). 180

[204] VOLPANO, D., AND SMITH, G. Probabilistic Noninterference in a Concurrent Language. *J. Computer Security 7*, 2–3 (Nov. 1999). 3

[205] WAGNER, G., GAL, A., WIMMER, C., EICH, B., AND FRANZ, M. Compartmental memory management in a modern web browser. *SIGPLAN Notices 46*, 11 (2011). 74

[206] WANG, H. J., FAN, X., HOWELL, J., AND JACKSON, C. Protection and communication abstractions for web browsers in MashupOS. *ACM SIGOPS Operating Systems Review 41*, 6 (2007). 55

[207] WANG, X., CHEN, H., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Improving Integer Security for Systems with KINT. In *Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2012), USENIX. 1

[208] WAYE, L., BUIRAS, P., KING, D., CHONG, S., AND RUSSO, A. It's My Privilege: Controlling Downgrading in DC-Labels. In *Security and Trust Management - 11th International Workshop, STM 2015, Vienna, Austria, September 21-22, 2015, Proceedings* (2015), pp. 203–219. 129

[209] WC3. Content Security Policy 1.0. http://www.w3.org/TR/CSP/, 2012. 31, 55, 59

[210] WC3. HTML5 Web Messaging. http://www.w3.org/TR/webmessaging/, 2012. 59, 165

[211] WC3. Web Workers. http://www.w3.org/TR/workers/, 2012. 58, 66

[212] WC3. Content Security Policy 1.1. https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html, 2013. 10, 76

[213] WC3. Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/, 2013. 55, 59

[214] WC3. HTML5. http://www.w3.org/TR/html5/, 2013. 55, 57, 67, 76

[215] WHATWG. HTML Living Standard. http://developers.whatwg.org/, 2013. 76

[216] WINSKEL, G. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993. 111

[217] YANG, E., STEFAN, D., MITCHELL, J., MAZIÈRES, D., MARCHENKO, P., AND KARP, B. Toward Principled Browser Security. In *HotOS* (2013). 2, 6, 31, 61, 64, 65

[218] YANG, E. Z., AND MAZIÈRES, D. Dynamic Space Limits for Haskell. In *PLDI* (2014). 6, 170

[219] YANG, J., HANCE, T., AUSTIN, T. H., SOLAR-LEZAMA, A., FLANAGAN, C., AND CHONG, S. End-To-End Policy-Agnostic Security for Database-Backed Applications. *CoRR abs/1507.03513* (2015). 4, 5, 50

[220] YANG, J., YESSENOV, K., AND SOLAR-LEZAMA, A. A language for automatically enforcing privacy policies. 85–96. 4, 50

[221] YIP, A., NARULA, N., KROHN, M., AND MORRIS, R. Privacy-preserving browser-side scripting with BFlow. In *EuroSys* (2009). 2, 3, 9, 10, 60, 82, 181

[222] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving application security with data flow assertions. In *Proceedings of the 22nd Symposium on Operating Systems Principles* (2009), ACM, pp. 291–304. 51

[223] ZANARINI, D., JASKELIOFF, M., AND RUSSO, A. Precise Enforcement of Confidentiality for Reactive Systems. In *Proc. IEEE Computer Sec. Foundations Symposium* (2013), IEEE, pp. 18–32. 180

[224] ZDANCEWIC, S., AND MYERS, A. C. Robust Declassification. In *Proc. IEEE Computer Security Foundations Workshop* (June 2001), pp. 15–23. 94, 129

[225] ZDANCEWIC, S., AND MYERS, A. C. Observational Determinism for Concurrent Program Security. In *CSFW* (2003), pp. 29–43. 148

[226] ZDANCEWIC, S., ZHENG, L., NYSTROM, N., AND MYERS, A. C. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *ACM Symposium on Operating Systems Principles* (Oct. 2001). 49

[227] ZDANCEWIC, S. A. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002. 134, 153

[228] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making Information Flow Explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and*

Implementation (Seattle, WA, November 2006), pp. 263–278. 2, 5, 7, 8, 30, 38, 48, 66, 70, 98, 99, 101, 102, 103, 106, 112, 113, 116, 129, 154, 157, 160, 180, 217

[229] ZELDOVICH, N., BOYD-WICKIZER, S., AND MAZIÈRES, D. Securing Distributed Systems with Information Flow Control. In *Proc. of the 6th Symp. on Networked Systems Design and Implementation* (San Francisco, CA, April 2008), pp. 293–308. 2, 5, 25, 93, 95, 102, 103

[230] ZELDOVICH, N., KANNAN, H., DALTON, M., AND KOZYRAKIS, C. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proc. of the Eighth Symp. on Operating Systems Design and Implementation* (San Diego, CA, December 2008), pp. 225–240. 2

[231] ZELTSER, L. Security Risks and Benefits of Docker Application Containers, June 2015. https://zeltser.com/security-risks-and-benefits-of-docker-application/. 46

[232] ZELWSKI, M. Browser Security Handbook, part 2. http://code.google.com/p/browsersec/wiki/Part2, 2011. 58

[233] ZHENG, L., CHONG, S., MYERS, A. C., AND ZDANCEWIC, S. Using Replication and Partitioning to Build Secure Distributed Systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2003), SP '03, IEEE Computer Society. 49

[234] ZHU, Y. backdooring your javascript using minifier bugs. https://zyan.scripts.mit.edu/blog/backdooring-js/, Aug. 2015. 9

Appendix A

Detailed proofs for the sequential LIO calculus

In this section, we provide expand the proof details for the results in Section 5.6.

Proposition 4 (Idempotence and distribution properties of the erasure function)**.** *The erasure function is:*

1. *Idempotent over terms:* $\varepsilon_l(t) = \varepsilon_l(\varepsilon_l(t))$
2. *Idempotent over memory*✎*:* $\varepsilon_l(m) = \varepsilon_l(\varepsilon_l(m))$
3. *Idempotent over configurations:* $\varepsilon_l(k) = \varepsilon_l(\varepsilon_l(k))$
4. *Homomorphic over substitution:* $\varepsilon_l(\{t_1 \,/\, x\}\, t_2) = \{\varepsilon_l(t_1) \,/\, x\}\, \varepsilon_l(t_2)$

Proof. The first property follows by induction on term t; all cases follow trivially from the inversion of the induction hypothesis. The second and third properties follow from the definition of the erasure function for memories and configurations and first property. The fourth property follows by induction on t_2; most cases follow directly from the induction hypothesis and definition of substitution. □

Since a number statements rely on several inversion and distribution properties for the erasure function, we give these below.

Proposition 12 (Inversion properties of the erasure function)**.** 1. *Labeled values:*

- *If* $l_1 \not\sqsubseteq l$ *then* $Labeled_b^{TCB}\ l_1\ \bullet = \varepsilon_l(Labeled_b^{TCB}\ l_1\ t)$ *for any* t.

- *If* $l_1 \sqsubseteq l$ *then* $Labeled_b^{TCB}\ l_1\ \varepsilon_l(t) = \varepsilon_l(Labeled_b^{TCB}\ l_1\ t)$.

2. *Monadic values:* $LIO_b^{TCB}\ \varepsilon_l(t) = \varepsilon_l(LIO_b^{TCB}\ t)$.

3. *Configurations:*

   - *If* $l_{cur} \not\sqsubseteq l$ *then* $\langle \bullet, \bullet, \bullet \mid \bullet \rangle = \varepsilon_l(\langle l_{cur}, c_{cur}, m \mid t \rangle)$ *for any* c_{cur}, m *and* t.
   - *If* $l_{cur} \sqsubseteq l$ *then* $\langle l_{cur}, c_{cur}, \varepsilon_l(m) \mid \varepsilon_l(t) \rangle = \varepsilon_l(\langle l_{cur}, c_{cur}, m \mid t \rangle)$.

Proof. All properties follow directly from the definition of the erasure function. □

This proposition states that, in certain cases, we can invert the application of the erasure function to labeled values, *LIO* values, and configurations.[1]

Simulation

Our simulation lemma follows by induction on the number of executed *toLabeled* blocks. The two lemma, Lemma 3 and 4, rely on several supporting propositions. We give these below.

Our first base-case simulation proposition considers the case when both the starting and end configuration labels can flow to the attacker observation level. In other words, the current term t does not raise the current label (e.g., with *unlabel*) nor does it execute any *toLabeled* blocks.

Proposition 13. *For any label* l, *such that* $l_{cur} \sqsubseteq l$ *and* $l'_{cur} \sqsubseteq l$, *if* $\langle l_{cur}, c_{cur}, m \mid t \rangle \xrightarrow{0} \langle l'_{cur}, c'_{cur}, m' \mid t' \rangle$ *then* $\varepsilon_l(\langle l_{cur}, c_{cur}, m \mid t \rangle) \xrightarrow{0} \varepsilon_l(\langle l'_{cur}, c'_{cur}, m' \mid t' \rangle)$

Proof. By induction on t. Most cases follow directly from inversion of the first $\xrightarrow{0}$ reduction hypothesis or Lemma 2. The $\gg\!=$ and *catchLIO* cases follow from the definition of the single- and multi-step relations, using Propositions 12 and a supporting proposition (not given here) whose statement is the multi-step version of this proposition. The terms for which there is a context reduction rule (e.g., *label*, *unlabel*, etc.), we further rely on Proposition 5. □

The next proposition considers the case when initial configuration cannot be observed by the attacker, i.e., the initial current label does not flow to the attacker label.

Proposition 14. *For any label* l, *such that* $l_{cur} \not\sqsubseteq l$, *if* $\langle l_{cur}, c_{cur}, m \mid t \rangle \xrightarrow{n} \langle l'_{cur}, c'_{cur}, m' \mid t' \rangle$ *then it is also the case that* $\varepsilon_l(\langle l_{cur}, c_{cur}, m \mid t \rangle) \xrightarrow{n} \varepsilon_l(\langle l'_{cur}, c'_{cur}, m' \mid t' \rangle)$

[1] We note that, while we can prove inversion for all terms (and cases), we only need properties 1 and 2 to prove the more interesting simulation property.

Proof. We break the proof into two cases:

- Case $l'_{\text{cur}} \sqsubseteq l$: follows from Proposition 6.
- Case $l'_{\text{cur}} \not\sqsubseteq l$: follows trivially from the single-step reduction rule of an erased configuration, (HOLE).

□

Here, the simplicity of the proof allows us to consider the case where the number of executed *toLabeled* blocks is any natural n.

The more interesting case—when the current label is raised by the current term t—is given below. As shown below, only *unlabel* actually raises the current label, hence, we can directly consider the simulation for an arbitrary number n of executed *toLabeled* blocks. However, we must consider the case when *unlabel* is executed as part of a bigger action (e.g., in $\gg\!=$ or *catchLIO*).

Proposition 15. *For any label l, such that $l_{cur} \sqsubseteq l$ and $l'_{cur} \not\sqsubseteq l$, if $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n} \langle l'_{cur}, c'_{cur}, m' \mid t'\rangle$ then $\varepsilon_l(\langle l_{cur}, c_{cur}, m \mid t\rangle) \xrightarrow{n}_l \varepsilon_l(\langle l'_{cur}, c'_{cur}, m' \mid t'\rangle)$*

Proof. By induction on t. Most cases follow directly by inversion of the $\xrightarrow{n}$ reduction rule and $l_{\text{cur}} \sqsubseteq l$ hypothesis. The remaining cases are:

- Case *unlabel* t_1: Breaks down into the three reduction rules for *unlabel*:
  - Case (UNLABELCTX): Follows directly from the definition of the $\xrightarrow{n}_l$ reduction rule and rule (UNLABELCTX), using Propositions 4 and 5.
  - Case (UNLABEL): Both sub-cases (where the label of the value being unlabel can and cannot flow to l) follow directly from the definition of the $\xrightarrow{n}_l$ reduction rule and rule (UNLABEL), using Propositions 4 and 12.
  - Case (UNLABELEX): Same as the (UNLABEL) case, but using the definition of (UNLABELEX) instead.
- Case $t_1 \gg\!= t_2$: Straight forward induction on t_1, using Propositions 4 and 12.
- Case *catchLIO* t_1 t_2: Straight forward induction on t_1.

□

These supporting statements are used to prove the base-case simulation, Lemma 3, where no *toLabeled* blocks are executed. However, all but one of the above supporting propositions consider the more general case, where any number of *toLabeled* blocks are executed. We need to extend Proposition 13 to arbitrary terms to prove the inductive case.

To do this, however, we must first show simulation for the big-step reduction relation holds (since *toLabeled* is defined in terms of a big-step), if the starting and end current labels can flow to the attacker label.

Proposition 16. *For any label l, such that* $l_{cur} \sqsubseteq l$, *if* $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{cur}, c'_{cur}, m' \mid LIO_b^{TCB}\ t'\rangle$ *then* $\langle l_{cur}, c_{cur}, m \mid \varepsilon_l(t)\rangle \xrightarrow{n}{}^{*} \langle l'_{cur}, c'_{cur}, m' \mid LIO_b^{TCB}\ \varepsilon_l(t')\rangle$

Proof. By induction on t, most cases follow by inversion of the first $\xrightarrow{n}{}^{*}$ reduction hypothesis and the resulting $\xrightarrow{n}$ hypothesis. This leaves us with the terms that reduce to *LIO* values: *LIO*, *return*, and *throwLIO*. The first follows by inversion and Lemma 2. The latter two follow directly from the definition of the $\xrightarrow{n}$ and $\xrightarrow{n}{}^{*}$ reduction relations. □

Using this proposition, the general version of Proposition 13 follows:

Proposition 17. *For any label l, such that* $l_{cur} \sqsubseteq l$ *and* $l'_{cur} \sqsubseteq l$, *if* $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n} \langle l'_{cur}, c'_{cur}, m' \mid t'\rangle$ *then* $\varepsilon_l(\langle l_{cur}, c_{cur}, m \mid t\rangle) \xrightarrow{n}_l \varepsilon_l(\langle l'_{cur}, c'_{cur}, m' \mid t'\rangle)$

Proof. We break this into two cases:

- Case $n = 0$: Trivially from Lemma 3.
- Case $n = n' + 1$: By induction on t. Most cases follow trivially by inversio The remaining cases are:
  - Case $t_1 \gg\!= t_2$: By inversion we break this down into the two sub-cases corresponding to the reduction rules (BIND) and (BINDEX). Both cases follow directly from the definition of the $\xrightarrow{n}_l$ reduction rule, using Propositions 4, 12, and 16.
  - Case *toLabeled* t_1 t_2: By inversion we break this down into the three sub-cases corresponding to the reduction rules:
    - Case (TOLABELEDCTX): Trivially by inversion.
    - Case (TOLABELED): Both sub-cases (where the label of the result can and cannot flow to l) follow directly from definition of the $\xrightarrow{n}_l$ reduction rule and rule (TOLABELED), using Propositions 4, 12, and 16.

  - Case (toLabeledEx): Like the (toLabeled) case, but using the definition of (toLabeledEx) instead.
- Case *catchLIO* t_1 t_2: By inversion we have two cases corresponding to (catchLIO) and (catchLIOEx), both of which follow in the same way as the $\gg\!=$ case.

□

Directly, the single-step simulation lemma, Lemma 4, for arbitrary terms follows.

Discretionary access control and isolation

First, we give the proof for Lemma 6, which states that the current computation cannot write to references below the current label:

Lemma 6 (No write-access below current label✎)**.** *Given a term t and memory m, such that $\varsigma\ (t)$ and $\varsigma\ (m \preceq c_{cur})$, if the term reduces to a value according to $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{cur}, c'_{cur}, m' \mid t'\rangle$, then $\overline{l_{cur} \preceq m} = \overline{l_{cur} \preceq m'}$.*

Proof. Observe that t can only modify m by creating a new reference or writing to an existing reference with (newLIORef) and (writeLIORef), respectively. Both of these rules require the label of the (potentially new) reference to be above l_{cur}. Hence we know that the memory below the current label will remain unchanged if t takes a single step. Using Proposition 6 we can directly extend this to an arbitrary number of steps. □

The somewhat dual statement, Lemma 7 states that the current computation cannot read or write to references above the current clearance (or create labeled values labeled as such):

Lemma 7 (No access above current clearance✎)**.** *Given term t and memory m, such that $\varsigma\ (t)$ and $\varsigma\ (m \preceq c_{cur})$, if the term reduces to a value according to $\langle l_{cur}, c_{cur}, m \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{cur}, c'_{cur}, m' \mid t'\rangle$, then $\overline{m \preceq c_{cur}} = \overline{m' \preceq c_{cur}}$.*

Proof. Observe that t can only modify m by creating a new reference or writing to an existing reference with (newLIORef) and (writeLIORef), respectively. Both of these rules require the label of the (potentially new) reference to be below c_{cur}. Hence we know that the memory above the current clearance will remain unchanged if t takes a single step. Using Proposition 6 we can directly extend this to an arbitrary number of steps. □

Indeed, since no memory above the current clearance can be accessed we can simply replace that part of the memory with arbitrary references:

Proposition 18 (Reduction is independent of memory above clerance✎)**.** *If* $\langle l_{cur}, c_{cur}, m_1 \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{cur}, c'_{cur}, m'_1 \mid t'\rangle$, $\varsigma\ (t)$, *and* $m_1 \preceq c_{cur} = m_2 \preceq c_{cur}$, *then* $\langle l_{cur}, c_{cur}, m_2 \mid t\rangle \xrightarrow{n}{}^{*} \langle l'_{cur}, c'_{cur}, m'_2 \mid t'\rangle$ *and* $m'_1 \preceq c_{cur} = m'_2 \preceq c_{cur}$.

Proof. Follows in the same way as the proof for Lemma 7. □

Before delving into the term isolation proof we first give two supporting propositions. First, a straightforward property for bind:

Proposition 19 (Term evaluation is obvlivious to memory above clearance✎)**.** *The reductions* $\langle l_{cur}, c_{cur}, m \mid t_1\rangle \xrightarrow{n_1}{}^{*} \langle l'_{cur}, c'_{cur}, m' \mid LIO^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$ *and* $\langle l'_{cur}, c'_{cur}, m' \mid t_2\rangle \xrightarrow{n_2}{}^{*} \langle l''_{cur}, c''_{cur}, m'' \mid LIO^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$ *hold iff* $\langle l_{cur}, c_{cur}, m \mid t_1 \gg t_2\rangle \xrightarrow{n}{}^{*} \langle l''_{cur}, c''_{cur}, m'' \mid LIO^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$ *holds and* $n = n_1 + n_2$.

Proof. Directly from definition of bind. □

Second, we give simple memory equivalence when store modifiers are used:

Proposition 20 (Equivalence of memory subsets✎)**.** *For labels* l_1, c_1, l_2, *and* c_2, *such that* $l_1 \sqsubseteq c_1$, $l_2 \sqsubseteq c_2$, $l_1 \not\sqsubseteq l_2$, $l_2 \not\sqsubseteq l_1$, $c_1 \not\sqsubseteq c_2$, *and* $c_2 \not\sqsubseteq c_1$, *if* $\overline{l_1 \preceq m \preceq c_1} = \overline{l_1 \preceq m' \preceq c_1}$ *then* $l_2 \preceq m \preceq c_2 = l_2 \preceq m' \preceq c_2$.

Proof. Since the labels are incomparable, it is easy to show that $(l_2 \preceq m \preceq c_2) \subset (\overline{l_1 \preceq m \preceq c_1})$ and $(l_2 \preceq m' \preceq c_2) \subset (\overline{l_1 \preceq m' \preceq c_1})$, from which the statement trivially holds. □

From these, the term isolation theorem follows in a mostly straightforward way.

Theorem 2 (Term isolation✎)**.** *Assume fresh*$(\cdot)$ *deterministically creates objects that are globally unique. Given safe terms* t_1 *and* t_2, *memory m, and labels* l_1, c_1, l_2, *and* c_2, *bounded by* l_{cur} *and* c_{cur}, *such that* $l_1 \sqsubseteq c_1$, $l_2 \sqsubseteq c_2$, $l_1 \not\sqsubseteq l_2$, $l_2 \not\sqsubseteq l_1$, $c_1 \not\sqsubseteq c_2$, *and* $c_2 \not\sqsubseteq c_1$, *if* $\langle l_{cur}, c_{cur}, m \mid$ *isolate* $l_1\ c_1\ t_1 \gg$ *isolate* $l_2\ c_2\ t_2\rangle \xrightarrow{n}{}^{*} \langle l_{cur}, c_{cur}, m' \mid LIO^{\mathsf{TCB}}\ ()\rangle$ *then* $\langle l_1, c_1, m \mid t_1\rangle \xrightarrow{n_1}{}^{*} \langle l'_1, c'_1, m_1 \mid LIO^{\mathsf{TCB}}\ ()\rangle$, *and* $\langle l_2, c_2, m \mid t_2\rangle \xrightarrow{n_2}{}^{*} \langle l'_2, c'_2, m_2 \mid LIO^{\mathsf{TCB}}\ ()\rangle$, $n = (n_1 + 1) + (n_2 + 1)$, *and* $\overline{l_1 \preceq m \preceq c_1} = \overline{l_1 \preceq m_1 \preceq c_1}$, $\overline{l_2 \preceq m \preceq c_2} = \overline{l_2 \preceq m_2 \preceq c_2}$, $l_1 \preceq m' \preceq c_1 = l_1 \preceq m_1 \preceq c_1$, *and* $l_2 \preceq m' \preceq c_2 = l_2 \preceq m_2 \preceq c_2$.

Proof. From Proposition 19, we have $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m \mid$ *isolate* $l_1\ c_1\ t_1\rangle \xrightarrow{n_1+1}{}^{*} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m_1 \mid LIO^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$ and $\langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m_1 \mid$ *isolate* $l_2\ c_2\ t_2\rangle \xrightarrow{n_2+1}{}^{*} \langle l_{\mathrm{cur}}, c_{\mathrm{cur}}, m' \mid LIO^{\mathsf{TCB}}_{\mathbf{true}}\ ()\rangle$.

Applying Lemma 8 to the first reduction we have $m_1 = (l_1 \preceq m_1 \preceq c_1) \cup (\overline{l_1 \preceq m \preceq c_1})$, $\langle l_1, c_1, m \mid t_1 \rangle \xrightarrow{n_1}{}^{*}$ $\langle l_1', c_1', m_1 \mid LIO^{\mathsf{TCB}}_{\mathbf{true}}\ () \rangle$, and $\overline{l_1 \preceq m \preceq c_1} = \overline{l_1 \preceq m_1 \preceq c_1}$.
From Proposition 20 and $\overline{l_1 \preceq m \preceq c_1} = \overline{l_1 \preceq m_1 \preceq c_1}$ we have $l_2 \preceq m \preceq c_2 = l_2 \preceq m_1 \preceq c_2$.
Applying Lemma 8 to the second reduction we have $m' = (l_2 \preceq m' \preceq c_2) \cup (\overline{l_2 \preceq m_1 \preceq c_2})$, $\langle l_2, c_2, m_1 \mid t_2 \rangle \xrightarrow{n_2}{}^{*}$ $\langle l_2', c_2', m' \mid LIO^{\mathsf{TCB}}_{\mathbf{true}}\ () \rangle$, and $\overline{l_2 \preceq m_1 \preceq c_2} = \overline{l_2 \preceq m' \preceq c_2}$.
From Proposition 20 and $\overline{l_2 \preceq m_1 \preceq c_2} = \overline{l_2 \preceq m' \preceq c_2}$ we have $l_1 \preceq m_1 \preceq c_1 = l_1 \preceq m' \preceq c_1$.
Further applying Proposition 18 we have $\langle l_2, c_2, m \mid t_2 \rangle \xrightarrow{n_2}{}^{*} \langle l_2', c_2', m_2 \mid LIO^{\mathsf{TCB}}_{\mathbf{true}}\ () \rangle$ and $m \preceq c_2 = m_2 \preceq c_2$. From Proposition 11 we have $\overline{l_2 \preceq m \preceq c_2} = \overline{l_2 \preceq m_2 \preceq c_2}$. □

Appendix B

Detailed semantics and proofs for the IFC-Inside calculus

B.1 Full Semantics for λ_{ES}

In Figure B.1 we give the full semantics for λ_{ES}. A subset of them has been given in Figure 6.1 earlier in the paper.

B.2 Example IFC Language with a Single Heap

As a concrete instantiation of this proof technique, we show how to make implement our IFC language using a single heap and ensure its non-interference using the techniques presented. First, we can construct the restricted language $L_{\text{IFC}}^{\mathscr{P}_{\text{norefs}}}(\alpha, \lambda_{ES})$, where $\mathscr{P}_{\text{norefs}}$ is the family of always valid predicates, except for the ones for I-SANDBOX and I-SEND, which we define as $P(e) = (\mathscr{AV}(e) = \emptyset)$ where $\mathscr{AV}(e)$ denotes the set of address variables in e. That is, we do not restrict any rules except for I-SANDBOX and I-SEND. Since P only depends on e, which is part of the current task and thus never erased w.r.t. the label of the first task, this language satisfies non-interference by Theorem 7.

The essential parts of the semantics for the concrete language with a single heap, which we call $L_{\text{IFC}}^{\text{Heap}}(\alpha)$, are given in Figure B.2. Most rules are straight-forward translations of the rules in Figs. 6.2 and 6.3 but for a single heap. For conciseness, we only show the interesting ones. Now, we can show an isomorphism between this language and $L_{\text{IFC}}^{\mathscr{P}_{\text{norefs}}}(\alpha, \lambda_{ES})$, which (by Theorem 5 and 6) guarantees non-interference for an appropriate scheduling policy α.

$$
\begin{array}{l}
v ::= \lambda x.e \mid \textbf{true} \mid \textbf{false} \mid a \\
e ::= v \mid x \mid e\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid \textbf{ref}\ e \mid !e \mid e := e \mid \textbf{fix}\ e \\
E ::= [\cdot]_T \mid E\ e \mid v\ E \mid \textbf{if}\ E\ \textbf{then}\ e\ \textbf{else}\ e \mid \textbf{ref}\ E \mid !E \mid E := e \mid v := E \mid \textbf{fix}\ E \\
e_1; e_2 \qquad \triangleq (\lambda x.e_2)\ e_1\ \textbf{where}\ x \notin \mathscr{FV}(e_2) \\
\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \triangleq (\lambda x.e_2)\ e_1
\end{array}
$$

$$\text{T-APP}\quad \frac{}{\mathscr{E}_\Sigma[(\lambda x.e)\ v] \to \mathscr{E}_\Sigma[\{v/x\}\ e]} \qquad \text{T-IFTRUE}\quad \frac{}{\mathscr{E}_\Sigma[\textbf{if true then } e_1 \textbf{ else } e_2] \to \mathscr{E}_\Sigma[e_1]}$$

$$\text{T-IFFALSE}\quad \frac{}{\mathscr{E}_\Sigma[\textbf{if false then } e_1 \textbf{ else } e_2] \to \mathscr{E}_\Sigma[e_2]} \qquad \text{T-REF}\quad \frac{\text{fresh}(a)}{\mathscr{E}_\Sigma[\textbf{ref}\ v] \to \mathscr{E}_{\Sigma[a \mapsto v]}[a]}$$

$$\text{T-DEREF}\quad \frac{(a, v) \in \Sigma}{\mathscr{E}_\Sigma[!a] \to \mathscr{E}_\Sigma[v]} \qquad \text{T-ASS}\quad \frac{}{\mathscr{E}_\Sigma[a := v] \to \mathscr{E}_{\Sigma[a \mapsto v]}[v]}$$

$$\text{T-FIX}\quad \frac{}{\mathscr{E}_\Sigma[\textbf{fix}\ (\lambda x.e)] \to \mathscr{E}_\Sigma[\{\textbf{fix}\ (\lambda x.e)/x\}\ e]}$$

Figure B.1: λ_{ES}: simple untyped lambda calculus extended with booleans, mutable references and general recursion. $\mathscr{FV}(e)$ returns the set of free variables in expression e.

$$\text{C-SANDBOX}\quad \frac{\mathscr{AV}(e) = \emptyset \qquad \Sigma' = \Sigma\left[i' \mapsto \textbf{nil}\right] \qquad t_1 = \langle E[i'] \rangle_l^i \qquad t_{\text{new}} = \langle_{\text{TI}} \lfloor e \rfloor \rangle_l^{i'} \qquad \text{fresh}(i')}{\Sigma; \Sigma; \langle E[\textbf{sandbox}\ e]_l \rangle_{l_1}^{i_1}, \ldots \hookrightarrow \Sigma'; \Sigma; \alpha_{\text{sandbox}}(t_1, \ldots, t_{\text{new}})}$$

$$\text{C-SEND}\quad \frac{\mathscr{AV}(e) = \emptyset \qquad l \sqsubseteq l' \qquad \Sigma\ (i') = \Theta \qquad \Sigma' = \Sigma\left[i' \mapsto (l', i, v), \Theta\right]}{\Sigma; \Sigma; \langle E[\textbf{send}\ i'\ l'\ v]_l \rangle_l^i, \ldots \to \Sigma; \Sigma; \alpha_{\text{step}}(\langle\langle\rangle\rangle_l^i, \ldots)}$$

Figure B.2: A selection of the reduction rules for $L_{\text{IFC}}^{\text{Heap}}(\alpha)$.

To this end, we represent addresses in the concrete language as pairs (i, a) where i is a task identifier, and a an address in the abstract system[1]. We also formulate the following well-formedness condition for configurations:

$$\mathrm{wf}(c) = \forall \langle e \rangle_l^i \in c.\ \{(i', e') \in \mathscr{AV}(e) \mid i \neq i'\} = \emptyset$$

Essentially, every address in a given task must have the correct identifier as the first part of the address. It is easy to see that the initial configuration satisfies this condition, and any step in the concrete semantics preserves the condition. Therefore, we only need to consider well-formed configurations, which allows us to give the two required functions f and f^{-1} for the isomorphism. For conciseness, we only give the interesting parts of their definition, and leave out the straight-forward proof that they actually provide an isomorphism.

- Addresses can be directly translated with $f((i, a)) = a$, and $f^{-1}(a) = (i, a)$ for an address a that occurs in task i.
- f splits the single heap into multiple heaps based on the i of the addresses. f^{-1} produces a single heap by translating the addresses and collapsing everything to a single store.

B.3 Extending the Core Calculus

As mentioned in the main body of this paper, actual IFC implementations may wish to extend the minimal system with more specialized constructs. In this section we show how to extend the language with several such constructs.

B.3.1 Labeled values

In traditional language-based dynamic IFC systems, a label is associated with values. Hence, a program that, for example, simply writes labeled messages to a labeled log can operate on both public and sensitive values. Similarly, a task that receives a sensitive value and forwards it to another task does not have be be at a sensitive level, if the value is not inspected. In its simplest form, our

[1]Note that this does not make the isomorphism trivial, as in the single heap, there is nothing preventing task 1 to access an address (2,a). Furthermore, it is common to represent addresses in this way for efficient garbage collection of dead tasks.

$$
\begin{array}{l}
v \ ::= \cdots \mid \textbf{Labeled}\ l\ e \\
e \ ::= \cdots \mid \textbf{label}\ e\ e \mid \textbf{unlabel}\ e \mid \textbf{labelOf}\ e \\
E ::= \cdots \mid \textbf{label}\ E\ e \mid \textbf{unlabel}\ E \mid \textbf{labelOf}\ E
\end{array}
$$

$$\textsc{I-label}\ \frac{l \sqsubseteq l'}{\mathscr{E}_{\Sigma}^{i,l}\left[\textbf{label}\ l'\ e\right] \rightarrow \mathscr{E}_{\Sigma}^{i,l}\left[\textbf{Labeled}\ l'\ e\right]} \qquad \textsc{I-unlabel}\ \mathscr{E}_{\Sigma}^{i,l}\left[\textbf{unlabel}\ (\textbf{Labeled}\ l'\ e)\right] \rightarrow \mathscr{E}_{\Sigma}^{i,l \sqcup l'}\left[e\right]$$

$$\textsc{I-labelOf}\ \frac{}{\mathscr{E}_{\Sigma}^{i,l}\left[\textbf{labelOf}\ (\textbf{Labeled}\ l'\ e)\right] \rightarrow \mathscr{E}_{\Sigma}^{i,l}\left[l'\right]}$$

Figure B.3: Syntax and semantics for labeled values. These rules are understood to be an addition to the existing rules given earlier.

coarse grained system requires that the current label of a task be at least at the level of the sensitive data to reflect the fact that such data is in scope.

If such fine-grained labeling of values is required, our base IFC system can be extended with explicitly labeled values, much like those of LIO and Breeze [186, 86]: $v ::= \cdots \mid \textbf{Labeled}\ l\ e$. Following LIO, we say that the expression e is protected by label l, while the label l itself is protected by the task's current label. The label of such values can be inspected the task without requiring the current label to be raised. However, when a task wishes to inspect the protected value e, it must first raise its label to at least l to reflect that it is incorporating data at such sensitivity level in its scope. When creating labeled values the label l must be above the current label; otherwise it cannot be said that protection has been transferred from the current label to l.

In Figure B.3, we formally show how to add this extension to the language. We assume that the constructor **Labeled** is not part of the surface syntax, but rather an internal construct.

B.3.2 Labeled mutable references/variables/channels

Extending the calculus with other labeled features, such as references, mutable variables [95], or channels, can be done in a similar manner: these references are implemented in the IFC language, separately from any preexisting notions of mutable references in the target language. There is some minor additional state to track: specifically, by amending Σ, as in [186, 183], we can allow threads to use these constructs to synchronize, or communicate with constructs other than **send/recv** in a safe manner. For example, when extending the calculus with labeled references, Σ additionally contains

$$
\begin{array}{rcl}
v & ::= & \cdots \mid a_l \\
e & ::= & \cdots \mid \textbf{new}\ e\ e \mid \textbf{read}\ e \mid \textbf{write}\ e\ e \\
E & ::= & \cdots \mid \textbf{new}\ E\ e \mid \textbf{new}\ l\ E \mid \textbf{read}\ E \\
 & & \mid \textbf{write}\ E\ e \mid \textbf{write}\ a_l\ E \\
\Sigma & ::= & \cdots \mid \Sigma\,[a_l \mapsto v]
\end{array}
$$

$$
\text{I-NEW}\quad \frac{l \sqsubseteq l' \qquad \text{fresh}(a) \qquad \Sigma' = \Sigma\,[a_{l'} \mapsto v]}{\mathscr{E}^{i,l}_{\Sigma}\,[\textbf{new}\ l'\ v] \to \mathscr{E}^{i,l}_{\Sigma'}\,[a_{l'}]}
$$

$$
\text{I-READ}\quad \mathscr{E}^{i,l}_{\Sigma}\,[\textbf{read}\ a_{l'}] \to \mathscr{E}^{i,l \sqcup l'}_{\Sigma}\,[\Sigma(a_{l'})]
$$

$$
\text{I-WRITE}\quad \frac{l \sqsubseteq l' \qquad \Sigma' = \Sigma\,[a_{l'} \mapsto v]}{\mathscr{E}^{i,l}_{\Sigma}\,[\textbf{write}\ a_{l'}\ v] \to \mathscr{E}^{i,l}_{\Sigma'}\,[\langle\rangle]}
$$

$$
\text{I-LABELOF2}\quad \mathscr{E}^{i,l}_{\Sigma}\,[\textbf{labelOf}\ a_{l'}] \to \mathscr{E}^{i,l}_{\Sigma}\,[l']
$$

Figure B.4: Syntax and semantics for labeled references. These rules are understood to be an addition to the existing rules given earlier.

a store that maps addresses to a value and a label which can be read and written to by different tasks through a labeled reference implementations.

In Figure B.4 details labeled references formally. The construct a_l is internal in the labeled reference implementation, and not part of the surface syntax. The changes to the language for labeled values and references require us to update the erasure function ε_l, whose full definition is shown in Figure B.5.

B.3.3 Clearance

Systems like LIO, COWL, and Breeze additionally provide a discretionary access control (DAC) mechanism—called *clearance*—at the language level [86, 186]. This mechanisms is used to restrict a computation from allocating and accessing data (or communicating with entities) above a specified label, the clearance. Amending our IFC language with clearance is straight forward, and, can be done using our notation of a restricted language. To this end, we first extend tasks to track a clearance label alongside the current label, and amend the core IFC language with two new terminals for retrieving and setting this value. Since this extension only adds a per-task mutable variable whose value has no influence on the system, all security guarantees still hold, by essentially the same proofs. However, this does not implement any DAC mechanism yet. To do so, we

$$\varepsilon_l(\Sigma; ts) = \varepsilon_l(\Sigma); \text{filter } (\lambda t.t = \bullet)\ (\text{map } \varepsilon_l\ ts)$$

$$\langle \Sigma, e\rangle^i_{l'} \begin{cases} \bullet & l' \not\sqsubseteq l \\ \langle \varepsilon_l(\Sigma), \varepsilon_l(e)\rangle^i_{l'} & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\textbf{Labeled } l'\ e) = \begin{cases} \textbf{Labeled } l' \bullet & l' \not\sqsubseteq l \\ \textbf{Labeled } l'\ e & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\emptyset) = \emptyset$$

$$\varepsilon_l(\Sigma\,[i \mapsto \Theta]) = \begin{cases} \varepsilon_l(\Sigma) & l' \not\sqsubseteq l, \text{ where } l' \text{ is the label of thread } i \\ \varepsilon_l(\Sigma)\,[i \mapsto \varepsilon_l(\Theta)] & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\Sigma\,[a_{l'} \mapsto v]) = \begin{cases} \varepsilon_l(\Sigma)\,[a_{l'} \mapsto \bullet] & l' \not\sqsubseteq l \\ \varepsilon_l(\Sigma)\,[a_{l'} \mapsto \varepsilon_l(v)] & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\Theta) = \Theta \preceq l$$

Figure B.5: Erasure function for the full IFC language, with all extensions. In all cases that are not specified, including target-language constructs, ε_l is applied homomorphically (e.g., $\varepsilon_l(\textbf{setLabel}\ e) = \textbf{setLabel}\ \varepsilon_l(e)$). This definition replaces the one from Figure 6.5, which is for the IFC language without extensions.

can restrict the language with a family of predicates $\mathscr{P}_{\text{clearance}}$: All rules that raise the current label (e.g., I-SETLABEL), perform allocation (e.g., I-SANDBOX and I-send), or set the clearance (clearance should not be arbitrarily raised), a predicate that uses the clearance to impose DAC is used. For instance, the predicate for I-SETLABEL prevents the current label from being raised above the clearance (and thus permit reads above the clearance). The predicate $P := l \sqsubseteq l'$ achieves this restriction, where l' is the clearance and l is the current label. The other predicates are defined in a similar way and omitted for brevity.

B.3.4 Privileges

Decentralized IFC extends IFC with the decentralized label model of Myers and Liskov [144] to allow for more general applications, including systems consisting of mutually distrustful parties. In a decentralized system, a computation is executed with a set of *privileges*, which, when exercised, allow the computation to declassify data (e.g., by lowering the current label). Practical IFC systems (e.g., [228, 186, 86, 145]) rely on privileges to implement many applications. The challenge with such an extension lies in the precise security guarantees that must be proved, which to the best of

our knowledge is an open research problem.

Our implementation for Node.js and COWL both provide privileges, but we have not formalized this part any further.

B.4 Non-Interference Proof

In this section we prove the theorems we have stated in the paper. Note that we prove soundness of the system including the formally defined extensions from Appendix B.3. We first observe that the non-interference claims for the languages $L_{\text{IFC}}(\text{SEQ}, \lambda)$ and $L_{\text{IFC}}(\text{RR}, \lambda)$ in Theorems 3 and 4 follow directly from Theorem 7, where the set of predicates is the set of always valid predicates (i.e., no restriction).

Before we proceed with the proof of Theorem 7, we state and proof two lemmas we will use.

Lemma 9. *For any task t, task lists ts, store Σ, and label l, if $\varepsilon_l(t) = \bullet$, then there exists a task list ts' and a store Σ' such that*

$$\Sigma; t, ts \hookrightarrow \Sigma'; ts, ts' \tag{B.1}$$

$$\varepsilon_l(ts') = \textbf{nil} \tag{B.2}$$

$$\varepsilon_l(\Sigma') = \varepsilon_l(\Sigma) \tag{B.3}$$

Proof. From $\varepsilon_l(t) = \bullet$ we know that the current label l_{cur} of t must be above l. Furthermore, tasks can always take a step (if no regular rule applies, then I-NOSTEP can be used), and thus we consider all rules that could be applied to execute t.

Case I-NOSTEP and I-DONE In this case, the task t is dropped, and thus $ts' = \textbf{nil}$ and $\Sigma' = \Sigma$ satisfy conditions (B.2) and (B.3).

Case I-SANDBOX The newly created task has a label of at least l_{cur}, and will thus be erased, as required by condition (B.2). Furthermore, the state only changes for the newly created thread, and thus the state change is erased, showing (B.3).

In all other rules, no new tasks are created, and thus ts' consists of just the one task t', to which t executed. Since the tasks label can only increase, t' is still erased, showing condition (B.2). We are left to show condition (B.3) for the remaining rules.

Case I-SEND A new message triple with label l' gets added to the message queue of the receiving thread. However, since $l_{\text{cur}} \sqsubseteq l'$, the triple will get erased.

Case I-RECV and I-NORECV In this case, only the queue of task t can change, which gets erased.

Case I-NEW The newly allocated address has to be at a label at least as high as l_{cur}, and will thus be erased.

Case I-WRITE Only addresses with a label l' above l_{cur} can be written, thus the change in Σ_1 will get erased.

Otherwise. None of the other rules modify the state Σ, and thus $\Sigma' = \Sigma$ will trivially satisfy condition (B.3).

□

Lemma 10. *We consider, for any target language λ, the restricted IFC language $L^{\mathscr{P}}_{IFC}(\alpha,\lambda)$ (according to Definition 12). Then, for any configurations c_1, c_1', c_2, and label l where*

$$c_1 \approx_l c_2 \qquad \textit{and} \qquad c_1 \hookrightarrow c_1' \tag{B.4}$$

there exists a configuration c_2' such that

$$c_1' \approx_l c_2' \qquad \textit{and} \qquad c_2 \hookrightarrow^* c_2' \,. \tag{B.5}$$

Proof. First, we observe there must be at least one task in c_1, otherwise it could not take a step. Thus, c_1 is of the form $\Sigma_1;t_1,ts_1$. Furthermore, let c_2 be $\Sigma_2;ts_2$. Consider two cases:

- $\varepsilon_l(t_1) = \bullet$. By the definition of ε_l, we know that $l \sqsubseteq l_{\text{cur}}$ where l_{cur} is the label of t_1. In this case, we do not need to take a step for c_2, because $c_2' = c_2$ will already be l-equivalent to c_1'. To show this, note that the tasks ts_1 in c_1 are left in the same order and unmodified (the scheduling policy only modifies the first task). The task t_1 either gets dropped (by I-NOSTEP), or transforms into a task t_1' as well as potentially spawning a new task t_1''. Since both t_1' and t_1'' have a label that is at least as high as the label of t_1 (can be seen by inspecting all reduction rules), they will get filtered by ε_l in c_1'. Therefore, the l-equivalence of the task list is guaranteed. Lets consider the possible changes to Σ_1: Only five reduction interact with Σ_1, thus it suffices to consider these cases:

  Case I-SEND A new message triple with label l' gets added to the message queue of the receiving thread. However, since $l_{\text{cur}} \sqsubseteq l'$, the triple will get erased.

Case I-RECV and I-NORECV In this case, only the queue of task t_1 can change, which gets erased.

Case I-NEW The newly allocated address has to be at a label at least as high as l_{cur}, and will thus be erased.

Case I-WRITE Only addresses with a label l' above l_{cur} can be written, thus the change in Σ_1 will get erased.

This ensures that $c_1' \approx_l c_2'$, as well as $c_2 \hookrightarrow^* c_2'$ (in zero steps), as claimed.

- $\varepsilon_l(t_1) \neq \bullet$. By the definition of ε_l, the task list ts_2 in c_2 must be of the form ts_2', t_2, ts_2'' (for some task lists ts_2', ts_2'' and some task t_2) where

$$\varepsilon_l(ts_2') = \textbf{nil} \tag{B.6}$$

$$\varepsilon_l(t_2) = \varepsilon_l(t_1) \tag{B.7}$$

$$\varepsilon_l(ts_2'') = \varepsilon_l(ts_1) \tag{B.8}$$

(where **nil** is the empty list of tasks). Now, intuitively we will first execute a number of steps to process the tasks in ts_2' (execute them one step and move them to the back of the task list, or drop them if they are done or stuck). Then, the task t_2 can take the same step as t_1, which will result in a configuration c_2' with the desired properties. More formally, we can proceed as follows:

First, we can apply Lemma 9 continuously for all the task in ts_2', until we reach a configuration $c_2'' = \Sigma_2'; t_2, ts_2'', ts_2'''$ for some ts_2''' such that $\varepsilon_l(ts_2''') = \textbf{nil}$ and $\varepsilon_l(\Sigma_2) = \varepsilon_l(\Sigma_2')$. We note that $\varepsilon_l(c_1) = \varepsilon_l(c_2'')$ (by the definition of ε_l).

Now, the first task t_2 in c_2'' is l-equivalent to the task t_1. This implies that the two tasks must have the same id, label and can only differ in the expression or store if some subexpression is of the form **Labeled** l' e. In this case, the expression e could be different in the two threads if $l_{\text{cur}} \sqsubseteq l'$. However, none of the reduction rules depend on an expression in that position, and there is never a hole in that position where evaluation could take place. Thus, the same rules will syntactically match for both task, and we are left to argue that all premises evaluate to the same values for t_1 and t_2, as well as that the resulting states Σ_1' and Σ_2'' are l-equivalent. The additional premises P that follow the condition in Definition 12 are not a problem, since those predicates only depend on $\varepsilon_l(c_1)$, which is equivalent to $\varepsilon_l(c_2'')$, and thus those predicates

evaluate in the same way. All other premises are either on the threads labels (which are the same), or on the state Σ_1, or Σ_2', respectively. Because $\varepsilon_l(\Sigma_1) = \varepsilon_l(\Sigma_2')$, all of these also evaluate in the same way, as can be seen by simply considering all rules that involve or change the state:

Case I-SEND Here, the task t_2 will send the same message to the same receiver queue. This queue is either completely erased, or it is l-equivalent. In both cases, l-equivalence of Σ_1' and Σ_2' is preserved.

Case I-RECV and I-NORECV When the tasks are receiving a message, then by the reduction rules we know that they first filter the queue by the label l_{cur} of t_1. We also know that the queues are equivalent when filtered by the less restrictive label l, thus the messages received (or dropped) from the queue are equivalent.

Case I-NEW The newly allocated address can be the same for both t_1 and t_2, thus resulting in l-equivalent states.

Case I-WRITE By $\varepsilon_l(t_1) = \mathit{earse}\ l\ t_2$ both tasks write the same value, and therefore the resulting states will still be l-equivalent.

After t_2 has taken a step, we finally arrive in the desired configuration $c_2' = \Sigma_2'';ts_2'',ts_2''',ts_2''''$, where ts_2'''' contains the task resulting from executing t_2 (and might contain, zero (if the task was done or stuck), one (for most steps) or two tasks if a new task was launched). As required, we have

$$c_2 \hookrightarrow^* c_2'' \hookrightarrow c_2' \quad \wedge \quad c_1' \approx_l c_2'.$$

□

With this, it is easy to proof Theorem 7 as follows.

Proof of Theorem 7, TSNI. We proof the theorem by induction on the length of the derivation sequence in (6.1). The base case for derivations of length 0 is trivial, allowing us to simple chose $c_2' = c_2$. In the step case, we assume the theorem holds for derivation sequences of length up to n, and show that it also holds for those of length $n+1$. We split the derivation sequence from (6.1) as follows:

$$c_1 \hookrightarrow c_1'' \hookrightarrow^n c_1'$$

for some configuration c_1''. By Lemma 10, we get c'' with

$$c_1'' \approx_l c_2'' \qquad \text{and} \qquad c_2 \hookrightarrow^* c_2'' \tag{B.9}$$

Applying the induction hypothesis to $c_1'' \hookrightarrow^n c_1'$, we get c_2' with

$$c_1' \approx_l c_2' \qquad \text{and} \qquad c_2'' \hookrightarrow^* c_2' \tag{B.10}$$

Stitching together the derivation sequences from (B.9) and (B.10) directly gives us the right-hand side of the implication in the TSNI definition (6.2), which concludes the proof. □