Thesis for the Degree of Doctor of Philosophy

Securing functional programs with floating-label information-flow control

Pablo Buiras

CHALMERS | UNIVERSITY OF GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Göteborg, Sweden 2016

Securing functional programs with floating-label information-flow control
Pablo Buiras
ISBN 978-91-7597-368-5

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 4049
ISSN 0346-718X

Technical Report 125D
Department of Computer Science and Engineering
Research group: Language-based security

Department of Computer Science and Engineering
Chalmers University of Technology and
University of Gothenburg
SE-412 96 Göteborg, Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2016

Abstract

The work presented in this thesis focuses on information-flow control systems for functional programs, particularly on the LIO library in Haskell. The thesis considers three main aspects in this area: timing covert channels, dynamic policies and enforcement mechanisms that improve precision of the analysis.

Timing channels are dangerous in the presence of concurrency. We start with the design, formalisation and implementation of a concurrent version of LIO which is secure against them. More specifically, we remove leaks due to non-terminating behaviour of programs (termination covert channel) and leaks produced by forcing certain interleavings of threads, as a result of affecting their timing behaviour (internal timing covert channel). The key insight is to decouple computations so that threads observing the timing or termination behaviour of other threads are required to be at the same confidentiality level. This work only deals with internal timing that can be exploited through language-level operations. We also mitigate leaks that result from the precise measurement of the timing of observable events (external timing covert channel), e.g. by using a stopwatch. In further work, we tackle leaks that result from hardware-based shared resources, such as the processor cache. This thesis presents a cache-based attack on LIO and proposes two solutions that rely on time-agnostic scheduling: the first one consists in a modification to the Haskell runtime and the other one is a purely language-based implementation. We also present a new manifestation of internal timing in Haskell, by exploiting lazy evaluation to encode sensitive information as timing perturbations.

Dynamic policies arise when the set of allowed flows of information is permitted to change as the program runs. Declassification can be viewed as a special case of dynamic policies. This thesis introduces an extension to LIO which supports dynamic policies and can encode well-known label formats such as the DLM and DC labels. Moreover, we also present the notion of restricted privileges, giving principals the ability to restrict the ways in which their authority can be used in the system, and supporting robust declassification.

We also add flow-sensitivity to LIO, which consists in the ability for the security labels of references to mutate, depending on the sensitivity of what is stored in them. Finally, we introduce a hybrid enforcement which mixes static and dynamic analyses. In particular, we leverage advanced type system features in Haskell to give the programmer control over which parts of the program are dynamically checked and which parts are statically checked. The core of this library is a general technique for deferring checking of type-class constraints to runtime which is applicable to other domains beyond language-based security.

Acknowledgements

The journey towards this thesis would not have been possible without the help and support of many people.

Firstly, I would like to thank my advisor Alejandro Russo for all his encouragement, his guidance, and for believing in me in the first place. Thank you for the stimulating discussions, for helping me develop my ideas to fruition, and for the awesome teamwork. Most of all, thanks for all the fun we have working together.

I also want to thank my collaborators and co-authors. I have learnt a lot from all of you, and I enjoyed every moment, even those last-minute rushes to meet the deadlines! The stress was worth it in the end.

I would also like to thank my colleagues at the department for providing such a friendly work environment. It has been a pleasure to share so many lunches and *fikas* with all of you over the past four years. And I also thank the secretaries and other administrative staff, who work so hard to make our experience here as smooth as possible. To my office mates, old and new, thanks for making the Arch/zsh/XMonad office a great place to work: you're awesome.

To my friends in Sweden, thanks for the good times we spent together, for your support and for hanging out with me. The late nights, the parties, Britcom evenings, the video games and tons of other activities and trips together have all made my time with you truly unforgettable. You've been great and I couldn't have asked for more.

A mis docentes, gracias José María y Guido por inculcarme pasión por la ciencia y a Fidel y Mauro por llevarme al mundo de la programación funcional.

Gracias a mis amigos de Argentina, porque a pesar de que nos vemos poco la amistad sigue y siempre cuento con ustedes. Al SELEN, a mis amigos del secundario, y al club de go de Rosario; mis vacaciones en Argentina no serían las mismas sin nuestros encuentros.

A mi familia, gracias por todo el apoyo incondicional y por el afecto que me brindan. Nada de esto sería posible sin ustedes. A mi hermano Ignacio, por estar siempre ahí y compartir tantas cosas conmigo.

Contents

CHAPTER

ONE

INTRODUCTION

There is no arguing that Computer Science is one of the driving forces behind innovation and development in the modern world. No other science has ever managed to transform the world as a whole in such a radical way as computing technology has during the last half of the 20th century. Our lives changed dramatically as we entered the so-called Information Age, and we started to become more and more reliant on computers for everything, including critical tasks in our society such as managing the social security system or the banking system. Information, and the way it disseminates, is a crucial part of this infrastructure.

Nowadays, personal information has become a valuable commodity. Many people own a smart phone, where they can install and use *apps*, and access social media websites. These apps are usually given access to potentially sensitive information such as contacts, text messages, and notes. Leaking sensitive information to third-parties can have serious consequences for the lives of the users, so it is necessary to develop mechanisms to secure this information and control its propagation. The most widespread approach is known as *access control*, where the user must give explicit permission to the application to access sensitive information or functionality. Once this access has been granted, there is no way of knowing how the application will use this information, and where it will be propagated. For example, an app with both read access to the phone's contacts and Internet access might send the contacts to a server on the Internet without explicit consent from the user. More generally, data breaches can have potentially catastrophic effects on the lives of individuals, including serious economic and personal consequences such as property or identity theft. Recent prominent data breaches, such as those at Target [Krebs, 2014], Home Depot [Sidel, 2014] and JP Morgan [Agrawal et al., 2014] – believed to be one of the largest in history – bring to light the impact of such incidents.

Information flow control (IFC) [Sabelfeld and Myers, 2003] is an alternative to access control that tracks how information is disseminated in a given program, and ensures that it is used according to a given policy. An information-flow-aware system is usually pictured as having information that concerns a number of agents or *principals*. A security policy specifies how these principals are related to each other, specifically in terms of how information is allowed to flow among them. For example, Alice, Bob and Charlie might be principals in the system of the company where they work, and Alice might trust Bob with her data but not Charlie, so she could use a security policy to specify that her data should be allowed to flow only to Bob. There are two sides to information flow control: confidentiality and integrity of data. In the most typical scenario, we are mainly interested in confidentiality, i.e. ensuring that secret information is not visible to unauthorised principals. Information-flow control aims to provide *end-to-end* security [Saltzer et al., 1984].

Lately, concurrency has become a necessity for practical applications. In the last decade, multi-core processors have become commonplace, so programmers expect to leverage this capability by writing multi-threaded programs. However, in the context of an information-flow control system, naively adding concurrency introduces a new possibility to leak information through *covert channels* [Lampson, 1973], i.e. leaking information by exploiting system features not intended for communication.

This thesis mainly focuses on *dynamic information flow control*, which involves enforcing security at runtime by checking all potentially insecure operations as they are performed by the program. When such an operation occurs, the program execution is stopped in some way, for example by simply aborting the program or, in some cases, by throwing a runtime exception.

This work is developed in the context of a specific kind of dynamic enforcement, based on a *floating-label* approach, which borrows ideas from the operating systems security research community [Zeldovich et al., 2006], and brings them into the field of language-based security. The main example of such an enforcement is LIO, a Haskell library for dynamic information-flow control which allows programmers to write programs with security guarantees.

Contributions The main contributions of this thesis revolve around making embedded floating-label IFC systems like LIO more useful in practice, by exploring how to add features that make it more resilient against attacks or make it more convenient for the programmer. The work in this thesis is divided into parts which contribute to the state of the art in the following three general research directions:

Part I: Covert channels and concurrency This part introduces a version of LIO with concurrency, while protecting against timing and termination

covert channels. These chapters address attacks that arise both from hardware-based timing perturbations and language-based channels.

Part II: Policy facets This part introduces mechanisms that allow for richer policies, such as policies that may change at runtime or language-based features to control information release.

Part III: Alternative enforcement mechanisms Information-flow control relies on language-based mechanisms to enforce the desired security properties. This part builds on the concurrent version of LIO and introduces alternative enforcement mechanisms that are more convenient – reducing programmer burden – and can improve efficiency and/or permissiveness, e.g. by rejecting fewer source programs.

In what follows, we provide a brief overview of IFC, timing covert channels, policy languages, enforcement mechanisms and LIO.

1.1 Information flow control

Information flow control first arose from the need to track the propagation of information in military contexts. The classic scenario for information-flow control is a system which contains both secret and public information, and we want to ensure that the public outputs of the program cannot be influenced by secret information, a security condition known as noninterference. One way of enforcing this is to think of a program as having endpoints (inputs and outputs) where information is consumed and produced. A *label* is attached to each of these endpoints, which indicates whether the information at that point is public or secret. Whenever the program attempts a write operation into a public output, the information-flow control system must check whether the information that is being written comes from (or, more generally, depends on) any secret input. If that is the case, the program is in violation of the security policy, and therefore considered insecure.

1.1.1 Policies

A *policy* is formalised as a relation among the security levels in the system, which specifies how information is allowed to flow between different levels. Typically, it is defined as a *lattice* structure [Denning, 1976] which induces an ordering relation, usually written $\sqsubseteq$. In general, $l_1 \sqsubseteq l_2$ means that information from level l_1 is allowed to flow to level l_2. A lattice also includes, among others, a binary operation known as *join*, written $\sqcup$, such that $l_1 \sqcup l_2$ is the least upper bound on l_1 and l_2. This operation is important for information-flow control as it can be used to find the least restrictive level in the lattice to which information from l_1 and l_2 may flow. The canonical example of a lattice is the two-point lattice with two levels, L and

H, which respectively stand for *low* (public) and *high* (secret), and where the only allowed flows are $L \sqsubseteq L$, $H \sqsubseteq H$, and $L \sqsubseteq H$. In general, the elements of the security lattice are used by the enforcement mechanism as labels for the information flowing through the system. The lattice elements can also be interpreted as actors/components in the system rather than just levels of confidentiality, which allows to express policies in a mutual distrust scenario [Myers and Liskov, 1997, Stefan et al., 2011]. Fig. 1 shows an example of such a security lattice. The arrows indicate allowed flows. This policy allows public information to flow to hospitals (Public $\sqsubseteq$ Hospitals) and insurance companies (Public $\sqsubseteq$ Insurance), but it does not allow medical records in the hospital to be disclosed to the insurance companies.

Downgrading and Dynamic policies In practice, most programs need to allow certain information leaks as part of their functionality. An intentional violation of the security policy is known as *downgrading*. A typical example of the need for downgrading is a password checker: given a guess for a user's password, the program must check whether or not the guess is correct. The answer to this question naturally conveys some information about the password, even if the guess is incorrect.

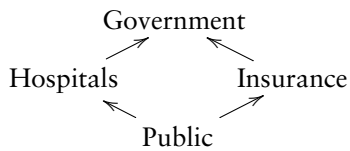

Fig. 1. Security lattice

The password checker example involves a downgrading of the boolean that represents whether the guess matches the password. Its security label would be changed from H (secret) to L (public), going "against the flow" specified by the usual two-point lattice. When the downgrading operation pertains to confidentiality, it is also known as *declassification*. Dually, downgrading also makes sense in terms of integrity, where it is known as *endorsement*.

Most information-flow systems regard the policy as static and fixed for the whole run on the program. However, in the real world there are situations which should be modelled as a change in the policy while the program runs. For example, these can include changes in organisation structure or subscription status in subscription-based services. An information-flow control system that allows such changes is said to support *dynamic policies*. Downgrading can be viewed as a special case of dynamic policies, where the current policy is temporarily weakened to allow the desired leak.

DC labels The systems that we consider in this thesis support rich security policies known as Disjunction Category (DC) labels [Stefan et al., 2011] that can model scenarios with several mutually-distrusting principals with mixed security concerns on data. DC labels are pairs of confidentiality and integrity policies. Confidentiality policies describe who may learn information. Integrity policies describe who takes responsibility or vouches for information. Both confidentiality and integrity policies are positive propositional formulas in conjunctive normal form, where propositional constants represent *principals*. For example, the formula $\mathsf{Alice} \vee \mathsf{Bob}$, when interpreted as a confidentiality formula, means that data with this label should be readable by either Alice or Bob. As an integrity policy, $\mathsf{Alice} \vee \mathsf{Bob}$ means that Alice and Bob collectively are responsible for the data, i.e., both may have contributed to, or influenced the computation of the data, but none of them takes sole responsibility. A conjunction $\mathsf{Alice} \wedge \mathsf{Bob}$ for confidentiality means that both principals need to agree or be involved in order to read the data, while for integrity it means that they both take responsibility for the data individually.

DC labels use logical implication to check for valid flows. For confidentiality, C_1 can flow to C_2 if C_2 logically implies C_1. Implication is used since it reflects the intuition that stronger formulas represent more sensitive information. Conversely, for integrity we have that I_1 can flow to I_2 if I_1 implies I_2. For example, data with confidentiality $\mathsf{Alice} \vee \mathsf{Bob}$ can flow to a sink labelled Alice since, as propositional formulas, $\mathsf{Alice} \Rightarrow \mathsf{Alice} \vee \mathsf{Bob}$. This matches the intuition that, if something is readable for either of two principals, it can be safely sent to one of them without breaking confidentiality. An example of an integrity flow would be for data labelled $\mathsf{Alice} \wedge \mathsf{Bob}$ to flow to a sink with integrity Alice, since $\mathsf{Alice} \wedge \mathsf{Bob} \Rightarrow \mathsf{Alice}$. Intuitively, this means that if both Alice and Bob take responsibility for some data individually, then certainly Alice can take responsibility for it.

Privileges DC labels also support a notion of downgrading by means of *privileges*, special tokens that represent the authority of one or more principals. Every principal has an associated privilege, and any code holding a principal's privilege is allowed to perform downgrading on that principal's behalf. DC label privileges are usually first-class values, which means that principals can pass their privileges around to delegate their authority to trusted parties. For example, consider a system where Alice and Bob are principals. If a piece of data is labelled as $\mathsf{Alice} \wedge \mathsf{Bob}$, Alice can use her privilege to declassify the data and relabel it to Bob, effectively removing herself from the conjunction and relinquishing her concerns on the data. This is how Alice can express that she trusts Bob with this piece of data. After this, Bob can do what he pleases with the data, including declassifying it to the public by exercising his own privilege. Alternatively, Alice could give Bob her privilege, which allows Bob to declassify all data labelled with

Alice as if he were Alice. This is how Alice can express complete trust in Bob, giving him the power to act on her behalf in the future.

1.1.2 Security property

In order to connect the meaning of security policies with the semantics of programs, we need a *security property*, a predicate that defines what it means for a program to be secure.

In information-flow control, the kind of security properties we would usually like to guarantee are known as *noninterference* [Goguen and Meseguer, 1982] properties. They can be informally stated in terms of a general malicious entity (the *adversary*), which has the ability to observe data below or at a given security level, but would like to observe confidential information not below that level. Then, we consider two independent runs of a program with inputs that are indistinguishable to the adversary, i.e. they only (potentially) differ in the parts that the adversary cannot see. We say that the program in question is *noninterfering* if the observable effects of these runs (outputs, return values, etc.) are also indistinguishable, as far as the adversary is concerned. In the scenarios that concern this thesis, the adversary is the provider of the source program that will be run.

Sometimes noninterference is too strong for practical purposes. Going back to the password checker example, we see that for the program to work as expected it must leak some information about the password after each attempt, which breaks noninterference. More generally, in contexts where dynamic policies or some form of information release is expected, we need a weaker property that allows downgrading but still provides some guarantees. In the example, we still want to ensure that the only data that is potentially leaked is the boolean representing whether the guess is correct. In order to express these richer conditions, in this work we will resort to knowledge-based properties [Askarov and Chong, 2012], which model how the knowledge of the adversary changes as the program produces observable outputs.

1.1.3 Enforcement mechanisms

In this thesis, we take a mostly *dynamic* approach to IFC. In a dynamic IFC system, the program is run alongside an *execution monitor*, a software component that is in charge of supervising the operations performed by the program (input/output in general) and checking that they comply with the security policy. The monitor will interrupt the execution of the program if a forbidden flow of information is detected. Apart from LIO, other examples of tools for dynamic IFC are JSFlow [Hedin et al., 2014], an information-flow-aware JavaScript interpreter which runs as a browser plugin, and Breeze [Hritcu et al., 2013], a high-level programming language for writing secure software with fine-grained IFC.

The alternative to the dynamic approach is *static information flow control* [Volpano et al., 1996], which consists in statically analysing a program, just by examining its text, and classifying it as either secure or insecure depending on how information is propagated by it. There are several examples of static IFC systems, such as Jif [Myers and Liskov, 2000] and Paragon [Broberg et al., 2013] (based on Java) and SecLib [Russo et al., 2008a] (based on Haskell). Jif and Paragon are extensions to the Java language that allow programmers to express security policies, which are enforced using a type system. SecLib is embedded as a library and leverages Haskell's type system to enforce information-flow security.

Lately, there has been much work attempting to combine these two approaches into *hybrid* enforcement mechanisms [Fennell and Thiemann, 2013, Askarov et al., 2015, Hedin et al., 2015]. A hybrid enforcement combines static and dynamic analyses in one system, typically by running a static analysis on the code at compile time and an execution monitor at runtime. From the point of view of type systems, a hybrid enforcement usually consists in a system which mixes static and dynamic types, such as gradual typing [Siek and Taha, 2006] or hybrid types [Flanagan, 2006].

1.2 Timing covert channels

Covert channels arise when information is leaked through mechanisms that were not originally designed for that purpose [Lampson, 1973]. For example, the execution time of a program, the number of open files, or even the current volume of the speakers can be used by malicious programs to convey information to each other. In particular, timing channels affect the timing of programs to cause observable events to depend on secrets. Covert channels are, in some cases, easy to exploit if the adversary has access to the source code of the program, and especially so if the adversary is the one who writes it.

One of the main lines of work in this thesis concerns the *internal timing covert channel* [Smith and Volpano, 1998]. This is a timing channel that exploits the interleaving of threads in a concurrent system to make the outcome of data races depend on sensitive information. The adversary can learn some of this information by observing these outcomes. In the rest of the thesis, the way in which the threads can encode bits of the secret into a data race usually depends on shared resources among the threads. Note that we do not assume the adversary to have the ability to precisely measure time when considering this channel. The channel where the information is conveyed by the measurement of time (using a stopwatch) is known as *external timing covert channel*. Termination can be regarded as a special case, where the program takes an infinite amount of time to produce an output, and this fact can be used by the adversary to obtain sensitive information.

It has been shown that termination and timing channels are capable of leaking a considerable amount of information in a concurrent setting [Askarov et al., 2008]. For this reason, care must be taken when adding concurrency to an IFC system. In this work, we start with an existing dynamic IFC system, LIO, and show how to make it secure against certain classes of timing covert channels. The following section is a brief introduction to this IFC system.

1.3 LIO

LIO, which stands for *Labelled IO*, is a dynamic information flow control system for Haskell, a purely functional language with strong static typing [Peyton Jones et al., 2003]. Purity, or the absence of side-effects, means that pure code is only vulnerable to flows of information from parameters to return values. Additionally, the effectful part of an LIO program is written in an embedded language for describing computations, which allows direct control over all side-effects performed by the program. Unlike mainstream programming languages, where any effects are allowed anywhere by default, Haskell in principle *disallows* effects everywhere, except for special parts of the program where effects are explicitly marked by the type system. These blocks are written using *monads* [Moggi, 1991], an abstract construction for specifying and combining effectful computations. As can be seen from earlier work [Russo et al., 2008b], this makes Haskell a suitable language for information-flow analysis. LIO leverages this functionality to restrict the side-effects that the program can perform in order to enforce security.

LIO is embedded in Haskell as a *library*: programmers write their programs using the LIO interface, and their execution will also perform security checks to enforce a given policy. This library provides security guarantees in the form of noninterference properties, in the sense that every valid LIO program is noninterfering by construction (modulo covert channels). LIO is also parameterised in the security policy, which is specified as a lattice over a type of security labels.

LIO uses a *floating-label* approach to information flow control. An LIO computation has a *current label* attached to it, which is an upper bound on the sensitivity of the data in scope. When a computation with current label L_C observes an object A with label L_A, its current label must change (possibly rise) to the least upper bound or *join* of the two labels, written $L_C \sqcup L_A$. The current label effectively "floats above" the labels of all objects it observes. When performing a side-effect that will be visible to label L, LIO first checks that the current label flows to L ($L_C \sqsubseteq L$) before allowing the operation to take place.

Fig. 2 shows a basic example of how LIO works, assuming the security lattice from Fig. 1. The code is the definition of a malicious func-

tion *stealInfo*, which attempts to steal confidential medical information and send it to an insurance company. The function takes one argument, *medicalRecord*, which has the label Hospital, and is supposed to contain the medical record of a person. This is an example of an LIO *labelled value*, which is simply a value protected by a security label. Labelled values must be unlabelled using the **unlabel** primitive, which returns the value itself and raises the current label of the LIO computation accordingly. In this example, we assume that the LIO computation has the label Public as its current label. After the first line has been executed, the current label of the computation would be Hospital. The medical record itself would be bound to m. In the second line, the program attempts to send[1] m to some insurance company, which has label Insurance, so the security check Hospital $\sqsubseteq$ Insurance is performed. Since the policy does not allow this flow, the program would be stopped at this point, and m would not reach *insuranceCompany*.

```
stealInfo medicalRecord =
  do m ← unlabel medicalRecord
     sendTo insuranceCompany m
```

Fig. 2. Simple LIO code.

When writing LIO programs, one must be careful in the way that the program is structured and how the operations interact with the current label. It is a common mistake to unlabel too many values from several sources in the same context, inadvertently raising the current label so much that no useful outputs can be performed any more. This effect is known as *label creep*, and can be alleviated by a combination of mindful programming and a local scoping primitive called *toLabeled*. The changes to the current label in a *toLabeled* block are undone after the block is executed, restoring the current label to what it was before running the block. In this sense, the floating-label approach seems to be a double-edged sword: on the one hand, it is perhaps a good idea to force programmers to structure their programs as a collection of blocks of code with different labels; on the other hand, the current label imposes a restrictive style that may constrain the programmer too much. Practical experience from the developers of GitStar [Giffin et al., 2012], an information-flow-aware system built on top of LIO, seem to indicate that the model is appropriate for writing such systems and that the programmers did not feel overly constrained by it.

[1] The operation *sendTo* does not exist as a primitive, but it is just meant as an example of an effectful operation that performs an output.

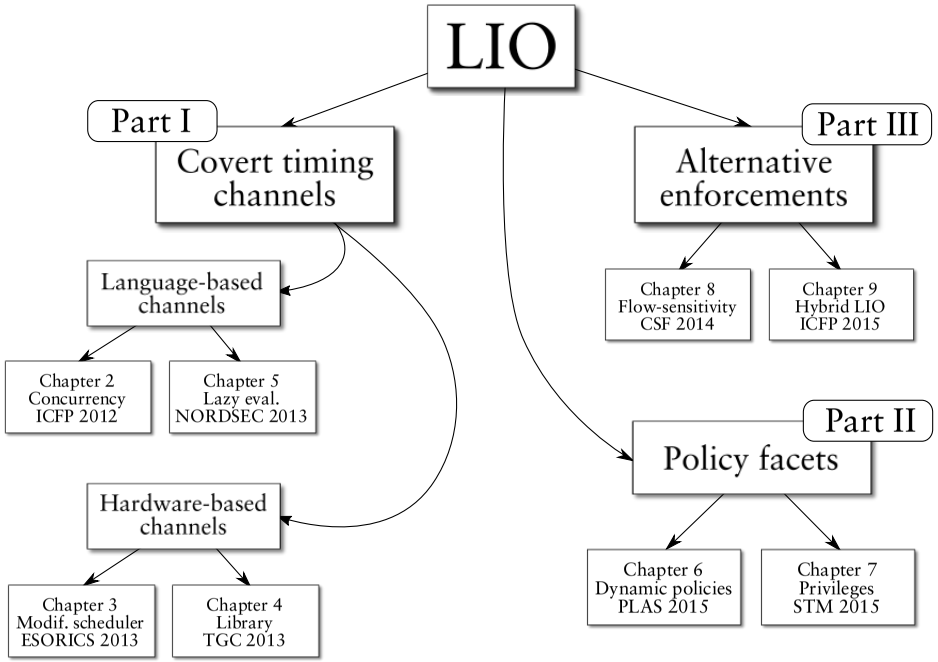

Fig. 3. Overview of work in this thesis

1.4 Thesis overview

The contents of this thesis have been published as individual papers in the proceedings of peer-reviewed conferences and symposia. Each of the eight chapters that follow presents one of these papers. This section briefly outlines their contents and states the contributions of the author.

Fig. 3 gives an overview of the work in this thesis. Part I concerns covert timing channels: Chapters 2 and 5 deal with language-based covert channels, while Chapters 3 and 4 present two different ways of addressing hardware-based timing perturbations such as those caused by the processor cache.

Taking the system in chapter 2 as a baseline, this work presents several extensions or modifications that make it more convenient to use. Part II (Chapters 6 and 7) explores different facets of security policies, such as dynamic policies and restricting the use of privileges for downgrading in DC labels. Finally, Part III (Chapters 8 and 9) explores alternative enforcement mechanisms for floating-labels systems, such as a system where the security label of references can change at runtime and a language-based hybrid enforcement of information-flow security fully embedded in Haskell using advanced type system features.

Chapter 2: Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems

As explained before, confidential information may be leaked through termination and timing channels. The termination covert channel has limited bandwidth for sequential programs [Askarov et al., 2008], but it is a more dangerous source of information leakage in concurrent settings.

In this chapter, we present an information-flow control system that is secure against the termination and internal timing channels, i.e. situations in which the outcome of a race among several threads depends on confidential information. Intuitively, we leverage concurrency by placing such potentially sensitive actions in separate threads, each with its own floating label. Then, we require other threads to raise their current label accordingly before observing termination and timing of higher-confidentiality contexts. Additionally, we show how to mitigate external timing in this setting using ideas from Askarov et al [Askarov et al., 2010]. The chapter introduces the concurrent version of LIO, which is, to the best of our knowledge, the first concurrent dynamic IFC system that deals with timing channels.

Statement of contributions The paper was co-authored with Deian Stefan, Alejandro Russo, Amit Levy, John C. Mitchell, and David Mazières. Pablo was mainly responsible for the contents of the Soundness section.

This chapter was published as a paper in the proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP) 2012.

Chapter 3: Eliminating cache-based timing attacks with instruction-based scheduling

In this chapter, we show that concurrent deterministic IFC systems that use time-based scheduling are vulnerable to a cache-based internal timing channel. We demonstrate this vulnerability with a concrete attack on LIO, which can be used to attack GitStar. The secret is encoded in the hardware cache, a resource that is implicitly shared among all threads and not modelled in the previous chapter. As a result, the cache is not subject to LIO's usual IFC mechanisms, and the attack succeeds.

To eliminate this internal timing channel, we implement *instruction-based scheduling*, a new kind of scheduler that is indifferent to timing perturbations from underlying hardware components, such as the cache, TLB, and CPU buses. We show this scheduler is secure against cache-based internal timing attacks for applications using a single CPU.

Statement of contributions This paper was co-authored with Deian Stefan, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Pablo discovered the cache-based attack for LIO, contributed to the design of the cache-aware semantics and was responsible for the Semantics and Formal guarantees sections.

This chapter was published as a paper in the proceedings of the 18th European Symposium on Research in Computer Security (ESORICS) 2013.

Chapter 4: A library for removing cache-based attacks in concurrent information flow systems

In the previous chapter we present cache-based attacks in concurrent information flow systems, and provide a solution which involves modifying the scheduler in the Haskell runtime. In this chapter, we tackle the same problem from a purely language-based perspective, by providing a Haskell library that can be used as a replacement for concurrent LIO and which is resilient against cache-based attacks. We leverage *resumptions* – a tame form of continuations – to attain fine-grained control over the interleaving of thread computations at the library level. Specifically, we remove cache-based attacks by ensuring that every thread yields after executing an "instruction", i.e., an atomic action, in analogy with the behaviour of the instruction-based scheduler from the previous chapter. In addition, this library supports running pure computations in parallel with the side-effecting ones.

Statement of contributions This paper was co-authored with Amit Levy, Deian Stefan, Alejandro Russo and David Mazières. Pablo contributed to the design and implementation of the library, and was responsible for the formalisation and proofs.

This chapter was published as a paper in the proceedings of the 8th International Symposium on Trustworthy Global Computing (TGC) 2013.

Chapter 5: Lazy Programs Leak Secrets

Haskell's evaluation mechanism is *lazy*, which means that arguments to functions are not evaluated until they are needed in the body of the function. Crucially, when such an argument (also known as a *thunk*) is finally evaluated, its value gets cached and is reused in subsequent occurrences of the same argument. In this chapter, we describe a novel exploit of lazy evaluation to reveal secrets in IFC systems through internal timing. We illustrate our claim with an attack on LIO. This attack is analogous to the cache-based attack, since thunks work like caches and can be shared by multiple threads by merely holding a pointer to them. We propose a countermeasure based on restricting the implicit sharing caused by lazy evaluation, but we do not implement these ideas, leaving them for future work.

Statement of contributions This paper was co-authored with Alejandro Russo. Pablo discovered the attack and contributed to the design of the proposed solution.

This chapter was published as a paper in the proceedings of the 8th Nordic Conference on Secure IT Systems (NordSec) 2013.

Chapter 6: Dynamic enforcement of dynamic policies

This chapter presents Stateful LIO, an information-flow control mechanism enforcing dynamic policies: security policies which change the relation between security levels while the system is running. We identify an implicit flow arising from the decision to change the policy based on sensitive information and introduce a corresponding check in the enforcement mechanism. We provide a formal security guarantee for Stateful LIO, presented as a knowledge-based property, which specifies that observers can only learn information in accordance with the level ordering. To illustrate the applicability of our results, we implement well-known label models such as DLM, the Flowlocks framework, and DC labels in Stateful LIO.

Statement of contributions This paper was co-authored with Bart van Delft. Both authors contributed equally to the writing of the paper.

This chapter was published as a paper in the proceedings of the 10th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS) 2015.

Chapter 7: It's my privilege: controlling downgrading in DC labels

As explained before, DC labels use capability-like privileges to downgrade information. Inappropriate use of privileges can compromise security, but DC labels provide no mechanism to ensure appropriate use. We extend DC labels with the novel notions of bounded privileges and robust privileges. Bounded privileges specify and enforce upper and lower bounds on the labels of data that may be downgraded. Bounded privileges are simple and intuitive, yet can express a rich set of desirable security policies. Robust privileges can be used only in downgrading operations that are *robust*, i.e., the code exercising privileges cannot be abused to release or

certify more information than intended. Surprisingly, robust downgrades can be expressed in DC labels as downgrading operations using a weakened privilege. We provide sound and complete run-time security checks to ensure downgrading operations are robust. We illustrate the applicability of bounded and robust privileges in a case study as well as by identifying a vulnerability in an existing application based on DC labels.

Statement of contributions This paper was co-authored with Lucas Waye, Dan King, Stephen Chong and Alejandro Russo. All authors worked equally on the formal aspects of the paper. Additionally, Pablo was in charge of devising and implementing the case study.

This chapter was published as a paper in the proceedings of the 11th International Workshop on Security and Trust Management (STM) 2015.

Chapter 8: On dynamic flow-sensitive floating-label systems

Flow-sensitivity consists in allowing data structures to have mutable security labels, i.e., labels that can change over the course of the computation, depending on the sensitivity of the data stored in the structure. This feature is often used to boost the permissiveness of the IFC monitor, by rejecting fewer programs, and to reduce the burden of explicit label annotations. However, when added naively, in a purely dynamic setting, mutable labels can expose a high bandwidth covert channel. In this work, we present an extension for LIO that safely handles flow-sensitive references. The key insight to safely manipulating the label of a reference is to not only consider the label on the data stored in the reference, i.e., the reference label, but also the label on the reference label itself. Taking this into consideration, we provide an upgrade primitive that can be used to change the label of a reference in a safe manner. To eliminate the burden of determining when a reference should be upgraded, we additionally provide a mechanism for automatic upgrades. Our approach naturally extends to a concurrent setting, not previously considered by dynamic flow-sensitive systems. For both our sequential and concurrent calculi, we prove non-interference by embedding the flow-sensitive system into the flow-insensitive LIO calculus, a surprising result on its own.

Statement of contributions This paper was co-authored with Deian Stefan and Alejandro Russo. Pablo was in charge of the formal results and all authors contributed equally to the writing of the paper.

This chapter was published as a paper in the proceedings of the 27th IEEE Computer Security Foundations Symposium (CSF) 2014.

Chapter 9: HLIO: Mixing static and dynamic typing for information-flow control in Haskell

In this chapter, we show how to give programmers the flexibility of deferring IFC checks to runtime, while also providing static guarantees – and the absence of runtime checks – for parts of their programs that can be statically verified. We present the design and implementation of

our approach, HLIO (Hybrid LIO), as an embedding in Haskell that uses a novel technique for deferring IFC checks based on singleton types and constraint polymorphism. We formalise HLIO, prove non-interference, and show how interesting IFC examples can be programmed. Although our motivation is IFC, our technique for deferring constraints goes well beyond and offers a methodology for programmer-controlled hybrid type checking in Haskell.

Statement of contributions This paper was co-authored with Dimitrios Vytiniotis and Alejandro Russo. All authors contributed equally to the writing of the paper.

This chapter was published as a paper in the proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP) 2015.

BIBLIOGRAPHY

Tanya Agrawal, David Henry, and Jim Finkle. JPMorgan hack exposed data of 83 million, among biggest breaches in history, 2014. URL `http://www.reuters.com/article/us-jpmorgan-cybersecurity-idUSKCN0HR23T20141003`.

A. Askarov, S. Chong, and H. Mantel. Hybrid Monitors for Concurrent Noninterference. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pages 137–151, July 2015. doi: 10.1109/CSF.2015.17.

Aslan Askarov and Stephen Chong. Learning is Change in Knowledge: Knowledge-based Security for Dynamic Policies. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 308–322, Piscataway, NJ, USA, June 2012. IEEE Press.

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88312-8.

Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proc. of the 17th ACM CCS*. ACM, 2010.

Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In Chung-chieh Shan, editor, *Programming Languages and Systems*, volume 8301 of *Lecture Notes in Computer Science*, pages 217–232. Springer International Publishing, 2013. ISBN 978-3-319-03541-3.

Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

L. Fennell and P. Thiemann. Gradual Security Typing with References. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 224–239, June 2013. doi: 10.1109/CSF.2013.22.

Cormac Flanagan. Hybrid type checking. *SIGPLAN Not.*, 41(1):245–256, January 2006. ISSN 0362-1340.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 47–60, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387886`.

Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

D. Hedin, L. Bello, and A. Sabelfeld. Value-Sensitive Hybrid Information Flow Control for a JavaScript-Like Language. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pages 351–365, July 2015. doi: 10.1109/CSF.2015.31.

Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. *Proc. 29th ACM Symposium on Applied Computing*, 2014.

Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All Your IFCException Are Belong to Us. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 3–17, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4.

Brian Krebs. The Target Breach, By the Numbers, 2014. URL `http://krebsonsecurity.com/2014/05/the-target-breach-by-the-numbers/`.

Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973. ISSN 0001-0782.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.

Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.

Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM Press, September 2008a.

Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in Haskell, 2008b.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in systems design. *ACM Trans. on Computer Systems*, 2(4): 277–288, 1984.

Robin Sidel. Home Depot's 56 Million Card Breach Bigger Than Target's, 2014. URL `http://www.wsj.com/articles/home-depot-breach-bigger-than-targets-1411073571`.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Prog. Languages*, January 1998.

Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Proc. of the NordSec 2011 Conference*, October 2011.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996. ISSN 0926-227X. URL `http://dl.acm.org/citation.cfm?id=353629.353648`.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

Part I

Covert timing channels

CHAPTER
TWO

ADDRESSING COVERT TERMINATION AND TIMING CHANNELS IN CONCURRENT INFORMATION FLOW SYSTEMS

Deian Stefan, Alejandro Russo, Pablo Buiras
Amit Levy, John C. Mitchell, David Mazières

Abstract. When termination of a program is observable by an adversary, confidential information may be leaked by terminating accordingly. While this termination covert channel has limited bandwidth for sequential programs, it is a more dangerous source of information leakage in concurrent settings. We address concurrent termination and timing channels by presenting an information-flow control system that mitigates and eliminates these channels while allowing termination and timing to depend on secret values. Intuitively, we leverage concurrency by placing such potentially sensitive actions in separate threads. While termination and timing of these threads may expose secret values, our system requires any thread observing these properties to raise its information-flow label accordingly, preventing leaks to lower-labeled contexts. We develop our approach in a Haskell library and demonstrate its applicability by implementing a web server that uses information-flow control to restrict untrusted web applications.

2.1 Introduction

Covert channels arise when programming language features are misused to leak information [Lampson, 1973]. For example, when termination of a program is observable to an adversary, a program may intentionally or accidentally communicate a confidential bit by terminating according to the value of that bit. While this termination covert channel has limited bandwidth for sequential programs, it is a significant source of information leakage in concurrent settings. Similar issues arise with covert timing channels, which are potentially widespread because so many programs involve loops or recursive functions. These channels, based on either internal observation by portions of the system or external observation, are also effective in concurrent settings.

We present an information-flow system that mitigates and eliminates termination and timing channels in concurrent systems, while allowing timing and termination of loops and recursion to depend on secret values. Because the significance of these covert channels depends on concurrency, we fight fire with fire by leveraging concurrency to mitigate these channels: we place potentially nonterminating actions, or actions whose timing may depend on secret values, in separate threads. While termination and timing of these threads may expose secret values, our system requires any thread observing these properties to raise its information-flow label accordingly. We develop our approach in a Haskell library that uses the Haskell type system to prevent code from circumventing dynamic information-flow tracking. We demonstrate the applicability of this approach by implementing a web server that applies information-flow control to untrusted web applications. Although we do not address underlying hardware issues such as cache timing, our language-level methods can be combined with hardware-level mechanisms as needed to provide comprehensive defenses against covert channels.

Termination covert channel Askarov et al. [Askarov et al., 2008] show that for sequential programs with outputs, the termination covert channel can only be exploited by exponentially complex brute-force: no attacker can reliably learn the secret in time polynomial in the size of the secret. Moreover, if secrets are uniformly distributed, the attacker's advantage (after observing a polynomial amount of output) is negligible in comparison with the size of the secret. Because of this relatively low risk, accepted sequential information-flow tools such as Jif [Myers et al., 2001], and FlowCaml [Simonet, 2003], are only designed to address termination-insensitive noninterference. In a concurrent setting, however, the termination covert channel may be exploited more significantly [Hedin and Sabelfeld, 2011]. We therefore focus on termination covert channels in concurrent programs and present an extension to our Haskell LIO library [Stefan et al., 2011], which provides dynamic tracking of labeled values. By providing labeled

`forkLIO` and `waitLIO`, our extension removes the termination covert channel from sequential and concurrent programs while allowing loops whose termination conditions depend on secret information.

Internal timing channel Multi-threaded programs can leak information through an *internal timing covert channel* [Volpano and Smith, 1999] when the observable timing behavior of a thread depends on secret data. This occurs when the time to produce a public event, such as placing public data on a public channel, depends on secret data, or, more generally, when a race to acquire a shared resource may be affected by secrets. We close this covert channel using the same approach as termination leaks: we decouple the execution of public events from computations that manipulate secret data. Using labeled `forkLIO` and `waitLIO`, computation depending on secret data proceeds in a new thread, and the number of instructions executed before producing public events does not depend on secrets. Therefore, a possible race to a shared public resource does not depend on the secret, eliminating internal timing leaks.

External timing channel External timing covert channels, which involve externally measuring the time used to complete operations that may depend on secret information, have been used in practice to leak information [Felten and Schneider, 2000, Bortz and Boneh, 2007] and break cryptography [Kocher, 1996, Handschuh and Heys, 1999, Wong, 2005]. While several mechanisms exist to mitigate external timing channels [Agat, 2000, Hedin and Sands, 2005, Barthe et al., 2006], external timing channels are not addressed by conventional information-flow tools and in fact most of the previous techniques for language-based information-flow control appear to have limited application. Our contribution to external timing channels is to bring the mitigation techniques from the OS community into the language-based security setting. Generalizing previous work [Askarov et al., 2010], Zhang et al. [Zhang et al., 2011] propose a black-box mitigation technique that we adapt to a language-based security setting. In this approach, the source of observable events is wrapped by a timing mitigator that delays output events so that they contain only a bounded amount of information. We take advantage of the way Haskell makes it possible to identify when outputs are produced and implement the mitigator as part of the `LIO` library. Leveraging Haskell monad transformers [Liang et al., 1995], we show how to modularly extend `LIO`, or any other library performing side-effects in Haskell, to provide a suitable form of Zhang et al.'s mitigator.

In summary, the main contributions of this paper are:

- We present an information flow control (IFC) system that eliminates the termination and internal timing covert channels, while mitigating the external timing one. The system provides support for threads, lightweight synchronization primitives, and allows loops and branches to

depend on sensitive (high) values. We believe this is the first implementation of a language-based IFC system for concurrency that does not rely on cooperative-scheduling.

- We eliminate termination and internal-timing covert channels using concurrency, with potentially sensitive actions run in separate threads. This is implemented in a Haskell library that uses labeled concurrency primitives[1].
- We provide language-based support for resource-usage mitigation using monad transformers. We use this method to implement the black-box external timing mitigation approach of Zhang et al.; the method is also applicable to other covert channels, such as storage.
- We demonstrate the language implementation by building a simple server-side web application framework. In this framework, untrusted applications have access to a persistent key-value store. Moreover, requests to apps may be from malicious clients colluding with the application in order to learn sensitive information. We show several potential leaks through timing and termination and show how our library is used to address them.

Section 2.2.3 provides background on information flow, Haskell, and the Haskell LIO monad. We discuss the termination covert channel and its elimination in Section 2.3, the internal timing covert channel and its elimination in Section 2.4, and the external timing channel and its mitigation in Section 2.5. Formalization of the library is given in Section 2.6 and the security guarantees in Section 2.7. The implementation and experimental evaluation are presented in Section 2.8. Related work is described in Section 2.9. We conclude in Section 2.10.

2.2 Background

We build on a dynamic information flow control library in Haskell called LIO [Stefan et al., 2011]. This section describes LIO and some of its relevant background. We first give an overview of IFC in abstract terms. We then give a brief overview of Haskell. Finally, we describe how LIO is implemented taking advantage of Haskell's static typing and pure functional nature.

2.2.1 Information flow control

IFC's goal is to track and control the propagation of information. In an IFC system, every observable bit has an associated *label*. Moreover, labels form a lattice [Denning, 1976] governed by a partial order $\sqsubseteq$ pronounced

[1] The library implementations discussed in this paper can be found at `http://www.scs.stanford.edu/~deian/concurrent_lio`

"can flow to." The value of a bit labeled L_{out} can depend on a bit labeled L_{in} only if $L_{\text{in}} \sqsubseteq L_{\text{out}}$.

In a *floating-label* system, every execution context has a label that can rise to accommodate reading more sensitive data. For a process P labeled L_P to observe an object labeled L_O, P's label must rise to the least upper bound or *join* of the two labels, written $L_P \sqcup L_O$. P's label effectively "floats above" the labels of all objects it observes. Furthermore, systems frequently associate a *clearance* with each execution context that bounds its label.

Specific label formats depend on the application and are not the focus of this work. Instead, we will focus on a very simple two-point lattice with labels `Low` and `High`, where `Low` $\sqsubseteq$ `High` and `High` $\not\sqsubseteq$ `Low`. We, however, note that our implementation is polymorphic in the label type and any label format that implements a few basic relations (e.g., $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$) can be used when building applications. The LIO library supports *privileges* which are used to implement decentralized information flow control as originally presented in [Myers and Liskov, 1997]; though we do not discuss privileges in this paper, our implementation also provides privileged-versions of the combinators described in later sections.

2.2.2 Haskell

We choose the Haskell programming language because its abstractions allow IFC to be implemented in a library [Li and Zdancewic, 2006]. Building a library is far simpler than developing a programming language from scratch (or heavily modifying a compiler). Moreover, a library offers backwards compatibility with a large body of existing Haskell code.

From a security point of view, Haskell's most distinctive feature is a clear separation of pure computations from those with side-effects. Any computation with side-effects must have a type encapsulated by the monad `IO`. The main idea behind the LIO library is that untrusted actions must be specified with a new `LIO` monad instead of `IO`. Because the types are different, untrusted code cannot bind `IO` actions to `LIO` ones. The only `IO` actions that can be executed within `LIO` actions are the ones that have been wrapped in the `LIO` type using a private constructor only visible to trusted code. All such wrapped `IO` actions perform label checks to enforce IFC.

2.2.3 The `LIO` monad

In this section, we give an overview of LIO. LIO dynamically enforces IFC, but without the features described in this paper, provides only *termination-insensitive* IFC [Askarov et al., 2008] for sequential programs. At a high level, LIO provides a monad called `LIO` (Labeled I/O) intended to be used in place of `IO`. The library furthermore contains a collection of `LIO` actions, many of them similar to `IO` actions from standard Haskell libraries,

except that the `LIO` versions contain label checks that enforce IFC. For instance, LIO provides file operations that look like those of the standard library, except that they confine the application to a dedicated portion of the file system where they store a label along with each file.

LIO is a floating-label system. The `LIO` monad keeps a *current label*, L_{cur}, that is effectively a ceiling over the labels of all data that the current computation may depend on. `LIO` also maintains a *current clearance*, C_{cur}, which specifies an upper bound on permissible values of L_{cur}.

LIO does not individually label definitions and bindings. Rather, all symbols in scope are identically labeled with L_{cur}. The only way to observe or modify differently labeled data is to execute actions that internally access privileged symbols. Such actions are responsible for appropriately validating and adjusting the current label.

As an example, the LIO file-reading function `readFile`, when executed on a file labeled L_F, first checks that $L_F \sqsubseteq C_{\text{cur}}$, throwing an exception if not. If the check succeeds, the function raises L_{cur} to $L_{\text{cur}} \sqcup L_F$ before returning the file content. The LIO file-writing function, `writeFile`, throws an exception if $L_{\text{cur}} \not\sqsubseteq L_F$.

As previously mentioned, allowing experimentation with different label formats, LIO actions are parameterized by the label type. For instance, simplifying slightly:

```
readFile :: (Label l) => FilePath -> LIO l String
```

To be more precise, it is really `(LIO l)` that is a replacement for the `IO` monad, where `l` can be any label type. The context `(Label l)=>` in `readFile`'s type signature restricts `l` to types that are instances of the `Label` typeclass, which abstracts the label specifics behind the basic methods $\sqsubseteq$, $\sqcup$, and $\sqcap$.

2.2.4 Labeled values

Since LIO protects all nameable values with L_{cur}, we need a way to manipulate differently-labeled data without monotonically increasing L_{cur}. For this purpose, LIO provides explicit references to labeled, immutable data through a polymorphic data type called `Labeled`. A locally accessible symbol (at L_{cur}) can name, say, a `Labeled l Int` (for some label type `l`), which contains an `Int` protected by a different label.

Several functions allow creating and using `Labeled` values:

- `label :: (Label l)=> l -> a -> LIO l (Labeled l a)`
  Given label $l : L_{\text{cur}} \sqsubseteq l \sqsubseteq C_{\text{cur}}$ and value v, action `label` l v returns a `Labeled` value guarding v with label l.
- `unlabel :: (Label l)=> Labeled l a -> LIO l a`
  If `lv` is a `Labeled` value v with label l, `unlabel lv` raises L_{cur} to $L_{\text{cur}} \sqcup l$ (provided $L_{\text{cur}} \sqsubseteq C_{\text{cur}}$ still holds, otherwise it throws an exception) and returns v.

Listing 1 Exploiting the termination channel by brute-force

```
bruteForce:: String -> Int -> Labeled l Int -> LIO l ()
bruteForce msg n secret = forM_ [0..n] $ \i -> do
  toLabeled High $ do
    s <- unlabel secret
    if s == i then undefined else return ()
  outputLow (msg ++ show i)
```

- `toLabeled :: (Label l) => l -> LIO l a -> LIO l (Labeled l a)`
  The dual of `unlabel`: given an action `m` that would raise L_{cur} to L'_{cur} where $L'_{\text{cur}} \sqsubseteq l \sqsubseteq C_{\text{cur}}$, `toLabeled` l `m` executes `m` without raising L_{cur}, and instead encapsulates `m`'s result in a `Labeled` value protected by label l.
- `labelOf :: (Label l)=> Labeled l a -> l`
  Returns the label of a `Labeled` value.

As an example, we show an `LIO` action that adds two `Labeled Int`s:

```
addLIO lA lB = do a <- unlabel lA
                  b <- unlabel lB
                  return (a + b)
```

If the inputs' labels are L_A and L_B, this action raises L_{cur} to $L_A \sqcup L_B \sqcup L_{\text{cur}}$ and returns the sum of the values.

We note that in an imperative language with labeled variables, dynamic labels can lead to implicit flows [Denning and Denning, 1977]. The canonical example is as follows:

```
public := 0;     // public has a Low label
if (secret)      // secret has a High label
  public := 1;   // public depends on secret
```

To avoid directly leaking the `secret` bit into `public`, one should track the label of the program counter and determine that execution of the assignment `public := 1` depends on `secret`, and raise `public`'s label when assigning `public := 1`. However, since the assignment executes conditionally depending on `secret`, now `public`'s *label* leaks the `secret` bit. LIO does not suffer from implicit flows. When branching on a secret, L_{cur} becomes `High` and therefore no public events are possible.

2.3 The termination covert channel

As mentioned in the introduction, information-flow control results and techniques for sequential settings do not naturally generalize to concurrent settings. In this section we highlight that the sequential LIO library allows leaks due to termination and show that a naive (but typical) extension that adds concurrency drastically amplifies this leak. We present a modification to the LIO library that eliminates the termination covert channel from

Listing 2 A concurrent termination channel attack

```
concurrentAttack :: Int -> Labeled l Int -> LIO l ()
concurrentAttack k secret = forM_ [0..k] $ \i -> do
  iBit <- toLabeled High $ do
            s <- unlabel secret
            return (extractBit i s)
  fork $ bruteForce (show k ++ "-bit:") 1 iBit
    where extractBit :: Int -> Int -> Int
          extractBit i n = (shiftR n i) .&. (bit 0)
```

both sequential and concurrent programs; our solution allows for flexible programming patterns, even writing loops whose termination condition depends on secret data.

Listing 1 shows an implementation of an attack (previously described by Askarov et al. in [Askarov et al., 2008]) that leaks a secret in a brute-force way through the termination covert channel. Function `bruteForce` takes three arguments: a string message, the public maximum value that a non-negative secret `Int` can have, and a secret labeled `Int`. Given these arguments the function returns an `LIO` action that when executed returns unit `()`, but producing intermediate side-effects. Namely, `bruteForce` writes to a `Low` labeled channel using `outputLow` while L_{cur} is `Low`. We assume that `bruteForce` is executed with the initial L_{cur} as `Low`.

The attack consists of iterating (variable `i`) over the domain of the secret (`forM_ [0..n]`), producing a publicly-observable output if the guess, `i`, is not the value of the secret. When `i` is equal to the secret, the program diverges (`if s == i then undefined`). We use the constant `undefined` to denote any non-terminating computation. Observe that on every iteration L_{cur} is raised to the label of the secret within the `toLabeled` block. However, as described in Section 2.2.3, the current label outside the `toLabeled` block remains unaffected, and so the computation can continue producing publicly-observable outputs. The leak due to termination is obvious: when the attacker, observing the `Low` labeled output channel, no longer receives any data, the value of the secret can be inferred given the previous outputs. For instance, to leak a 16-bit bounded `secret`, we can execute `bruteForce "It is not: " 65536 secret`. Assuming the value of the secret is 4, executing the action produces the outputs "It is not: 0", "It is not: 1", "It is not: 2", "It is not: 3" before diverging. An observer that knows the implementation of `bruteForce` can directly infer that the value of the secret is 4. Observe that the code producing public outputs (`outputLow (msg ++ show i)`) does not inspect secret data at all, which makes it difficult to avoid termination leaks by simply tracking the flow of labeled data inside programs.

Suppose that we (naively) add support for concurrency to LIO using a hypothetical primitive `fork`, which simply spawns computations in new threads. Although we can preserve termination-insensitive non-interference

(i.e., retain the property of no-explicit nor implicit-flows), we can extend the previous brute force attack to leak information in linear, as opposed to exponential, time in the length of the secret. In general, adding concurrency primitives in a straight-forward manner makes attacks that leverage the termination covert channel very effective [Hedin and Sabelfeld, 2011]. To illustrate this point, the attack of Listing 2 leaks the bit-contents of a secret value in linear time as follows. Given the bit-length `k` of a secret and the labeled `secret`, `concurrentAttack` returns an action which, when executed, extracts the bits of the secret (`extractBit i s`) and spawns a corresponding thread to recover them by executing the brute-force attack of Listing 1 (`bruteForce (show k ++ "-bit:") 1 iBit`). Hence, by collecting the public outputs generated by the different threads (having the form "0-bit:0", "1-bit:0", "2-bit:0", etc.), it is directly possible to recover the secret value.

2.3.1 Removing the termination covert channel in LIO

Since LIO is a floating-label system and at each point in the computation the evaluation context has a current label, a leak to a `Low` channel due to termination *cannot* occur after the current label is raised to `High`, unless the label-raise is within an enclosed `toLabeled` computation. Unless enclosed within a `toLabeled`, having L_{cur}`=High` implies that publicly-observable side-effects are no longer allowed. Hence, we can deduce that a piece of LIO code can exploit the termination covert channel only when using `toLabeled`. The key insight is that `toLabeled` is the single primitive in LIO that effectively allows a piece of code to temporarily raise its current label, perform a computation, and then continue with the starting current label. The attack in Listing 1 is a clear example that leverages this property of `toLabeled` to leak information.

Consider the necessary conditions for eliminating the termination channel present in Listing 1: the execution of the publicly-observable `outputLow` action must not depend on, or wait for, the secret computation executed within the `toLabeled` block. More generally, to close the termination covert channel, it is necessary to decouple the execution of computations enclosed by `toLabeled`. To achieve such decoupling, instead of using `toLabeled`, we provide an alternative primitive that executes computations that might raise the current label (as in `toLabeled`) in a newly-spawned thread. Moreover, to observe the result (or non-termination) of a spawned computation, the current label is firstly raised to the label of the (possibly) returned result. In doing so, after observing a secret result (or non-termination) of a spawned computation, actions that produce publicly-observable side-effects can no longer be executed. In this manner, the termination channel is closed.

In Listing 1, the execution of `outputLow` is bound to the termination of the computation described by `toLabeled`. However, using our proposed approach of spawning a new thread when performing `toLabeled`, if the code following the `toLabeled` wishes to observe whether or not the `High`

computation has terminated, it would first need to raise the current label to `High`. Thereafter, an `outputLow` action cannot be executed regardless of the result (or termination) of the `toLabeled` computation.

Concretely, we close the termination channel by removing the insecure function `toLabeled` from `LIO` and, instead, provide the following (termination sensitive) primitives.

```
forkLIO :: Label l => l -> LIO l a -> LIO l (Result l a)
waitLIO :: Label l => Result l a -> LIO l a
```

Intuitively, `forkLIO` can be considered as a concurrent version of `toLabeled`. `forkLIO l lio` spawns a new thread to perform the computation `lio`, whose current label may rise, and whose result is a value labeled with `l`. Rather than block, immediately after spawning a new thread, the primitive returns a value of type `Result l a`, which is simple a handler to access the labeled result produced by the spawned computation. Similar to `unlabel`, we provide `waitLIO`, which inspects values returned by spawned computations, i.e., values of type `Result l a`. The labeled wait, `waitLIO`, raises the current label to the label of its argument and then proceeds to inspect it.

In principle, rather than forking threads, it would be enough to prove that computations involving secrets terminate, e.g., by writing them in Coq or Agda. However, while this idea works in theory, it is still possible to crash an Agda or Coq program at runtime: for example, with a stack overflow. Generally, abnormal termination due to resource exhaustion exploits the termination channel and it could be hard to counter. In this light, forking threads is a manner to remove the termination channel by design. Although it might seem expensive, forking threads in Haskell is a light-weight operation[2].

We note that adding concurrency to `LIO` is a major modification which introduces security implications beyond that of handling the termination channel. In the following section, we describe the *internal timing covert channel*, a channel present in programming languages that have support for concurrency and shared-resources.

2.4 The Internal timing covert channel

In a concurrent setting, the possibility that threads have to share resources opens up new information channels. Specifically, multi-threaded programs can leak information through the *internal timing covert channel* [Volpano and Smith, 1999]. The source of the leaks comes from the ability of threads to affect their timing behavior based on secret data and thus affect, via the scheduler, the order of public events.

[2] `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent.html`

Listing 3 Internal timing leak

```
sthread :: String -> Int -> Labeled l Bool -> LIO l ()
sthread msg n secret = do toLabeled High
                                ( do s <- unlabel secret
                                     if s then sleep n
                                     else return () )
                          outputLow msg

pthread :: String -> Int -> LIO l ()
pthread msg n = do sleep n
                   outputLow msg

attack :: Labeled l Bool -> LIO l ()
attack secret = do fork (sthread "True" 5000 secret)
                   fork (pthread "False" 1000)
```

To illustrate internal timing attacks, we consider the LIO library from Section 2.2.3 with the added hypothetical primitive `fork` used to spawn a new thread. Listing 3 illustrates an internal timing attack. It consists of two threads: `sthread` and `pthread`. Command `sleep n` puts a thread to sleep for `n` milliseconds. Thread `sthread` takes a string to output in a public channel (`outputLow msg`) and the number of milliseconds to sleep (`sleep n`) if the secret boolean taken as argument (`secret`) is true. Thread `pthread`, on the other hand, does not take any secret but it writes a message (`msg`) to the same output channel as `sthread` after sleeping some milliseconds. Observe that both threads share a resource (i.e., the output channel) and that the timing behavior of `sthread` depends on the secret boolean.

With this example, `sthread` should take longer to execute the `outputLow` action than `pthread` if and only if the secret boolean is true. In isolation, both threads are secure, i.e., they satisfy non-interference. In fact, when considering them in isolation, both threads always produce the public output given by the argument `msg`. However, by running them concurrently, it is possible to leak information about `secret`. Function `attack` spawns two threads that execute `sthread` and `pthread` concurrently. Under many reasonable schedulers, if `secret` is true, it is more likely that the instruction `outputLow "False"` is executed first. On the other hand, if `secret` is false, it is more likely that `outputLow "True"` is executed first. An attacker can then observe the value of `secret` by just observing the second produced output.

Unlike other timing channel attacks, internal timing attacks do not require an attacker to measure the actual execution time to deduce secret information. The interleaving of threads is simply responsible for producing leaks! Although the example in Listing 3 shows how to leak one bit, it is easy to place the attack in a loop that leaks bit by bit a whole secret value in linear time. Tsai et al. [Tsai et al., 2007] show how effective the attack is even without having much information about the run-time system (e.g., the

scheduler). The authors implemented the magnified version of the attack in Listing 3 and showed how to leak a credit card number.

2.4.1 Removing the internal timing channel

As indicated by the code in Listing 3, the internal timing covert channel can be exploited when the time to produce public events (e.g., sending some data in a public channel) depends on secrets. In other words, internal timing arises when there is a race to acquire a shared resource that may be affected by secret data. In order to close this channel, we apply the same technique as for dealing with termination leaks: we decouple the execution of public events from computations that manipulate secret data. By using `forkLIO` and `waitLIO`, computations dealing with secrets are spawned in a new thread. In that manner, any possible race to a shared public resource does not depend on the secret anymore and thus internal timing leaks are no longer possible.

2.4.2 Synchronization primitives in concurrent LIO

In the presence of concurrency, synchronization is vital. This section introduces an IFC-aware version of `MVar`s, which are well-established synchronization Haskell primitives [Jones et al., 1996]. As with `MVar`s, `LMVar`s can be used in different manners: as synchronized mutable variables, as channels of depth one, or as building blocks for more complex communication and synchronization primitives.

A value of type `LMVar l a` is mutable location that is either empty or contains a value of type `a` labeled with `l`. `LMVar`s are associated with the following operations:

```
newEmptyLMVar :: (Label l) => l -> LIO l (LMVar l a)
putLMVar      :: (Label l) => LMVar l a -> a -> LIO l ()
takeLMVar     :: (Label l) => LMVar l a -> LIO l a
```

Function `newEmptyLMVar` takes a label `l` and creates an empty `LMVar l a` for any desired type `a`. The creation succeeds only if the label `l` is between the current label and clearance of the `LIO` computation that creates it. Function `putLMVar` fills an `LMVar l a` with a value of type `a` if it is empty and blocks otherwise. Dually, `takeLMVar` empties an `LMVar l a` if it is full and blocks otherwise.

Note that both `takeLMVar` and `putLMVar` check if the `LMVar` is empty in order to proceed, and they both end up modifying it in some way. It might seem like `takeLMVar` is only a read and `putLMVar` is only a write, but `takeLMVar` also does a write by emptying the location, and `putLMVar` also does a read by checking if the location is empty. Therefore, `takeLMVar` and `putLMVar` perform both a read and a write of the mutable location, which means that from a security point of view, operations on a given `LMVar l a` are executed only when the label `l` is below or equal to the clearance (i.e.,

$l \sqsubseteq C_{\text{cur}}$ due to the read) and above or equal to the current label (i.e., $L_{\text{cur}} \sqsubseteq l$ due to the write). Moreover, after either operation, L_{cur} is raised to l.

Many communication channels used in practice are often *bi-directional*, i.e., a read produces a write (and vice versa). For instance, reading a file may modify the access time in the inode; writing to a socket may produce an observable error if the connection is closed, etc. As described above, `LMVar` are bi-directional channels. If we were to treat them as uni-directional, observe that a termination leak would be possible: a thread, whose current label is `Low` can use a `LMVar` labeled `Low` to send information to a computation whose current label is `High`; the `High` thread can then decide to empty the `LMVar` according to a secret value and thus leak information to the `Low` thread.

2.5 The external timing covert channel

In a real-world scenario IFC applications interact with unlabeled, or publicly observable, resources. For example, a server-side IFC web application interacts with a browser, which may itself be IFC-unaware, over a public network channel. Consequently, an adversary can take measurements *external* to the application (e.g., the application response time) from which they may infer information about confidential data computed by the application. Although our results generalize (e.g., to the storage covert channel), in this section we address the *external timing covert channel*: an application can leak information over a public channel to an observer that precisely measures message-arrival timings. Note that the content of a message does not need to be public (hence why the channel is considered *covert*); this is the case in a web application where a message may be encrypted with SSL, but the actual placement of a message on the channel is observable by a network attacker.

Most of the language-based IFC techniques that consider external timing channels are limited. Despite the successful use of external timing attacks to leak information in web [Felten and Schneider, 2000, Bortz and Boneh, 2007] and cryptographic [Kocher, 1996, Handschuh and Heys, 1999, Wong, 2005] applications, they remain widely unaddressed by mainstream, practical IFC tools, including Jif [Myers et al., 2001]. Furthermore, most techniques that provide IFC in the presence of the external timing channel [Agat, 2000, Hedin and Sands, 2005, Barthe et al., 2006] are overly restrictive, e.g., they do not allow folding over secret data.

2.5.1 Mitigating the external timing channel

Recently, a predictive black-box mitigation technique for external timing channels has been proposed [Askarov et al., 2010, Zhang et al., 2011]. The predictive mitigation technique assumes that the attacker has control

of the application (which computes on secret data) and can measure the time a message is placed on a channel (e.g., when a response is sent to the browser). Treating the application as a black-box source of events, a mitigator is interposed between the application and the system output.

Internally, the mitigator keeps a *schedule* describing when outputs are to be produced. For example, the time mitigator might keep a schedule "predicting" that an output is to be produced every 1ms. If the application delivers events according to the schedule, or at a higher rate, the mitigator will be able to produce an output at every 1ms interval, according to the schedule, and thus leak no information.

Of course, the application may fail to deliver an event to the mitigator on time, and thus render the mitigator's schedule prediction false. At this point, the mitigator must handle the misprediction by selecting, or "predicting", a new schedule for the application. In most cases, this corresponds to doubling the application's *quantum*. For instance, following a misprediction of a quantum of 1 ms, an application will be then expected to produce an output every 2 ms. It is at the point of switching schedules where an attacker learns information: rather than seeing events spaced at 1 ms intervals, the attacker now observes outputs at 2 ms intervals, indicating that the application violated the predicted behavior (a decision that can be affected by secret data). However, Askarov et al. [Askarov et al., 2010] show that the amount of information leaked by this *slow-doubling* mitigator is polylogarithmic in the application runtime.

Furthermore, the aspects of the predictive mitigation technique of [Askarov et al., 2010, Zhang et al., 2011] that makes it particularly attractive for use in LIO are:

- The mitigator can adaptively reduce the quantum, as to increase the throughput of a well-behaved application in a manner that bounds the covert channel bandwidth (though the leakage factor is still greater than that of the slow-doubling mitigator);
- The mitigator can leverage public factors to decide a schedule. For example, in a web application setting where responses are mitigated, the arrival of an HTTP request can be used as a "reset" event. This is particularly useful as a quiescent application would otherwise be penalized (by increasing its quantum) for not producing an output according to the predicted schedule. Our web application of Section 2.8 implements this mitigation technique
- The amount of information leaked is bound by a combinatorial analysis on the number of observations an attacker can perform.

Monadic approach to black-box mitigation Pure functional programming languages, such as Haskell, are particularly suitable for mitigating external timing covert channels. Specifically, the use of monads for enforcing an evaluation-order and introducing side-effects allows for the reasoning and

control of output events. Among many others, LIO is an example library that leverages this property of monads; `LIO` is simply a monad that performs side-effects according to IFC.

The functionality of different monads, such as I/O and error handling, can be combined in a modular fashion using *monad transformers* [Liang et al., 1995]. A monad transformer t, when applied to a monad m, generates a new, combined monad $t\ m$, that shares the behavior of monad m as well as the behavior of the monad encoded in the monad transformer. The modularity of monad transformers comes from the fact that they consider the underlying monad m opaque, i.e., the behavior of the monad transformer t does not depend on the internal structure of m. In this light, we adopt Zhang et al.'s system-oriented predictive black-box mitigator to a language-based security setting in the form of a monad transformer.

2.5.2 Language-based mitigators

We envision the implementation of mitigators that address covert channels other than external timing. For example, our ongoing work includes the implementation of a storage mitigator that addresses attacks which vary message (packet) length to encode secret information. Hence, our mitigation monad transformer `MitM s q` is polymorphic in the mitigator-specific state `s` and quantum type `q` :

```
newtype MitM s q m a = MitM ...
```

The time-mitigated monad transformer is a special case:

```
type TimeMitM = MitM TStamp TStampDiff
```

where the internal state `TStamp` is a time stamp, and the quantum `TStampDiff` is a time difference. Superficially, a value of type `TimeMitM m a` is a computation that produces a value of type `a`. Internally, a time measurement is taken whenever an output is to be emitted in the underlying monad `m`, the internal state and quantum are adjusted to reflect the event, and the output is delayed if it was produced ahead of the predicted schedule.

We provide the function `evalMitM`, which takes an action of type `MitM s q m a` and returns an action of type `m a`, which when executed will mitigate the computation outputs. Observe that the monad transformer leaves the possibility to use (almost) any underlying monad `m`, not just `LIO` or `IO`; this makes the monad transformer approach to mitigation quite general.

Unfortunately, this generality comes with a trade-off: either every computation `m` is mitigated, or trustworthy programmers must define *what* objects they wish to mitigate and *how* to mitigate them. Given that the former design choice would not allow for distinguishing between inputs and outputs, we implemented the latter and more explicit mitigation approach.

To define *what* is to be mitigated (e.g., a file handle, a socket, a reference, etc.), we provide the data type:

```
data Mitigated s q a = Mitigated ...
```

For example, a time-mitigated I/O file handle is simply:

```
type TimeMitigated = Mitigated TStamp TStampDiff
type Handle = TimeMitigated IO.Handle
```

The use of `Mitigated` allows us to do mitigation at very fine grain level. Specifically, the monad transformer can be used to implement a mitigator for each `Mitigated` value (henceforth "handle"). This allows an application to write to multiple files, all of which are mitigated independently, and thus may be written to, at different rates[3]. It remains for us to address: *how* are the mitigators defined?

Mitigators are defined as instances of the type class `Mitigator`, which provides two functions:

```
class MonadConcur m => Mitigator m s q where
  -- | Create a Mitigated "handle".
  mkMitigated :: Maybe s   -- ^ Internal state
              -> q         -- ^ Quantum
              -> m a       -- ^ Handle constructor
              -> MitM s q m (Mitigated s q a)

  -- | Mitigate an operation
  mitigate :: Mitigated s q a  -- ^ Mitigated "handle"
           -> (a -> m ())      -- ^ Output computation
           -> MitM s q m ()
```

Firstly, we note the context `MonadConcur m` is used to impose the requirement that the underlying monad be an `IO`-like monad which allows forking new threads (as to separate the mitigator from the computation being mitigated) and operations on mutable `MVars` (which are internal to the `MitM` transformer). Secondly, we highlight the `mkMitigated` function, which is used to create a mitigated handle given an initial state, quantum, and underlying constructor. The default implementation of `mkMitigated` creates the mitigator state (internal to the transformer) corresponding to the handle. A simplified version of our `openFile` operation shows how `mkMitigated` is used:

```
openFile :: FilePath -> IOMode -> TimeMitigated IO.Handle
openFile f mode = mkMitigated Nothing q $ do
  h <- IO.openFile f mode   -- Handle constructor
  return h
    where q = mkQuant 1000 -- Initial quantum of 1ms
```

xo Here, the constructor `IO.openFile` creates a file handle to the file at path `f`. This constructor is supplied to `mkMitigated`, in addition to the

[3] In cases where schedule mispredictions are common, it is important to implement the *l-grace* period policy of [Zhang et al., 2011]. The policy states that when there are more than l mispredictions, the new scheduling should affect all mitigators.

"empty" state `Nothing`, and initial quantum `q` of 1 ms, which creates the corresponding mitigator and `Mitigated` handle (recall `Handle` is a type alias to `TimeMitigated IO.Handle`). We note that although the default definition of `mkMitigated` creates a mitigator per handle, instances may provide a definition that is more coarse-grained (e.g., associate mitigator with current thread).

Finally, each mitigator provides a definition for `mitigate`, which specifies how a computation should be mitigated. The function takes two arguments: the mitigated handle and a computation that produces an output on the handle. Our time mitigator instance

```
instance ... => Mitigator m TStamp TStampDiff where
  mitigate mh act = ...
```

provides a definition for `mitigate`. The action first retrieves the internal state of the mitigator corresponding to the mitigated handle `mh` and forks a new thread (allowing other mitigated actions to be executed). In the new thread, a time measurement t_1 is taken. Then, if the time difference between t_1 and the mitigator time stamp t_0 exceeds the quantum q, the new mitigator quantum is set to $2q$; otherwise, the computation is "suspended" for $t_1 - t_0$ microseconds. Following, `act` is executed, and the internal timestamp is replaced with the current time. Using MVars, we force operations on the same handle to be sequential and thus follow the latest schedule.

Continuing the example, we can now define a function we wish to be mitigated:

```
hPut :: Handle -> ByteString -> TimeMitigated IO ()
hPut mH bs = mitigate mH (\h -> IO.hPut h bs)
```

If `hPut` is invoked according to schedule (at least every 1 ms), the actual output function `IO.hPut` is used to write the provided byte-strings every 1 ms. Conversely, if the function does not follow the predicted schedule, the quantum will be increased and write-throughput to the file will decrease. Of course, this does not affect the schedule on a different handle (until a large number of mispredictions occur).

Adapting an existing program to have mitigated outputs comes almost for free: a *trustworthy* programmer needs to define the constructor functions, such as `openFile`, and output functions, such as `hPut`, and simply *lift* all the remaining operations. Recall that `MitM` is a monad transformer, and thus we provide a definition for the function:

```
lift ::: Monad m => m a -> MitM s q m a
```

which lifts a computation in the `m` monad into the mitigation monad, without performing any actual mitigation. A simple example illustrating this is the definition of `hGet` which reads a specified number of bytes from a handle:

```
hGet :: Handle -> Int -> TimeMitigated IO ByteString
```

Listing 4 Syntax for values, expressions, and types.

$$
\begin{array}{rrl}
\text{Label:} & & l \\
\text{LMVar:} & & m \\
\text{Value:} & v ::= & \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid l \mid m \mid x \mid \lambda x.e \mid \texttt{fix}\ e \\
& & \mid \texttt{Lb}\ l\ e \mid (e)^{\texttt{LIO}} \mid \boxdot \mid \texttt{R}\ m \mid \bullet \\
\text{Expression:} & e ::= & v \mid e\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
& & \mid \texttt{return}\ e \mid e\ \texttt{>>=}\ e \mid \texttt{label}\ e\ e \\
& & \mid \texttt{unlabel}\ e \mid \texttt{lowerClr}\ e \mid \texttt{getLabel} \\
& & \mid \texttt{getClearance} \mid \texttt{labelOf}\ e \mid \texttt{out}\ e\ e \\
& & \mid \texttt{forkLIO}\ e\ e \mid \texttt{waitLIO}\ e \mid \texttt{newLMVar}\ e\ e \\
& & \mid \texttt{takeLMVar}\ e \mid \texttt{putLMVar}\ e\ e \mid \texttt{labelOfLMVar}\ e \\
\text{Type:} & \tau ::= & \texttt{Bool} \mid \texttt{()} \mid \tau \to \tau \mid \ell \mid \texttt{Labeled}\ \ell\ \tau \mid \texttt{Result}\ \ell\ \tau \\
& & \mid \texttt{LMVar}\ \ell\ \tau \mid \texttt{LIO}\ \ell\ \tau
\end{array}
$$

```
hGet mH = lift . IO.hGet . mitVal
```

where `mitVal` returns the handle encapsulated by `Mitigated`. It is worth noting that, although we focus on mitigating writing operations, in some systems a file read will be reflected in the file's inode `atime`, and thus should be also accordingly mitigated.

2.6 Formal semantics for LIO

In this section, we formalise our library for a simply typed Curry-style call-by-name λ-calculus with some extensions. Listing 4 defines the formal syntax for the language. Syntactic categories v, e, and τ represent values, expressions, and types, respectively. Values are side-effect free while expressions denote (possible) side-effecting computations. Due to lack of space, we only show the reduction and typing rules for the core part of the library. For more details, readers can refer to Appendix A available in the supplementary material.

Values The syntax category v includes the symbol `true` and `false` representing Boolean values. Symbol `()` represents the unit value. Symbol ℓ denotes security labels. Symbol m represents `MVar`s. Values include variables (x), functions ($\lambda x.e$), and recursive functions (`fix` e). Special syntax nodes are added to this category: `Lb` v e, $(e)^{\texttt{LIO}}$, `R` m, $\boxdot$, and $\bullet$. Node `Lb` v e denotes the run-time representation of a labeled value. Similarly, node $(e)^{\texttt{LIO}}$ denotes the run-time representation of a monadic `LIO` computation. Node $\boxdot$ denotes the run-time representation of an empty `MVar`. Node `R` m is the

Listing 5 Typing rules for special syntax nodes.

$$\frac{}{\Gamma \vdash \bullet : \tau} \qquad \frac{}{\Gamma \vdash m : \texttt{LMVar}\ \ell\ \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Lb}\ l\ e : \texttt{Labeled}\ \ell\ \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e)^{\texttt{LIO}} : \texttt{LIO}\ \ell\ \tau} \qquad \frac{}{\Gamma \vdash \boxdot : \tau} \qquad \frac{\Gamma \vdash m : \texttt{LMVar}\ \ell\ \tau}{\Gamma \vdash \texttt{R}\ m : \texttt{Result}\ \ell\ \tau}$$

run-time representation of a handle, implemented as an `MVar`, that is used to access the result produced by spawned computations. Alternatively, `R` m can be thought of as an explicit *future*. Node $\bullet$ represents an erased term (explained in Section 2.7). None of these special nodes appear in programs written by users and they are merely introduced for technical reasons.

Expressions Expressions are composed of values (v), function applications ($e\ e$), conditional branches (`if` e `then` e `else` e), and local definitions (`let` x = e `in` e). Additionally, expressions may involve operations related to monadic computations in the `LIO` monad. More precisely, `return` e and e `>>=` e represent the monadic return and bind operations. Monadic operations related to the manipulation of labeled values inside the `LIO` monad are given by `label`, and `unlabel`. Expression `unlabel` e acquires the content of the labeled value e while in a `LIO` computation. Expression `label` e_1 e_2 creates a labeled value, with label e_1, of the result obtained by evaluating the `LIO` computation e_2. Expression `lowerClr` e allows lowering of the current clearance to e. Expressions `getLabel` and `getClearance` return the current label and current clearance of an `LIO` computation. Expression `labelOf` e obtains the security label of labeled values. Expression `out` e_1 e_2 denotes the output of e_2 to the output channel at security level e_1. For simplicity, we assume that there is only one output channel per security level. Expression `forkLIO` e_1 e_2 spawns a thread that computes e_2 and returns a labeled value with label e_1. Expression `waitLIO` e inspects the value returned by the spawned computation whose result is accessed by the handle e. Monadic operations related to creating, reading, and writing labeled `MVars` are respectively captured by expressions `newLMVar`, `takeLMVar`, and `putLMVar`.

Types We consider standard types for Booleans (`Bool`), unit (`()`), and function ($\tau \to \tau$) values. Type ℓ describes security labels. Type `Result` ℓ τ denotes handles used to access labeled results produced by spawned computations, where the results are of type τ and labeled with labels of type ℓ. Type `LMVar` ℓ τ describes labeled `MVars`, with labels of type ℓ and storing values of type τ. Type `LIO` ℓ τ represents monadic `LIO` computations, with a result type τ and the security labels of type ℓ.

The typing judgments have the standard form $\Gamma \vdash e : \tau$, such that expression e has type τ assuming the typing environment Γ; we use Γ

for both variable and store typings. Typing rules for the special syntax nodes are shown in Listing 5. These rules are liberal on purpose. Recall that special syntax nodes are run-time representations of certain values, e.g., labeled `MVars`. Thus, they are only considered in a context where it is possible to uniquely deduce their types. The typing for the remaining terms and expressions are standard and we therefore do not describe them any further. We do not require any of the sophisticated features of Haskell's type-system, a direct consequence of the fact that security checks are performed at run-time. Since typing rules are straightforward, we assume that the type system is sound with respect to our semantics.

The `LIO` monad is essentially implemented as a State monad. To simplify the formalization and description of expressions, without loss of generality, we make the state of the monad part of the run-time environment. More precisely, each thread is accompanied by a local security run-time environment σ, which keeps track of the current label ($\sigma.\texttt{lbl}$) and clearence ($\sigma.\texttt{clr}$) of the running `LIO` computation. Common to every thread, the symbol Σ holds the global LMVar store ($\Sigma.\phi$) and the output channels ($\Sigma.\alpha_l$, one for every security label l). A store ϕ is a mapping from LMVars to labeled values, while an output channel is a queue of events of the form out(v) (output) or exit(v) (termination), for some value v. For simplicity, we assume that every store contains a mapping for every possible LMVar, which is initially the syntax node ($\bullet$). The run-time environments Σ, σ, and a `LIO` computation form a *sequential configuration* $\langle \Sigma, \langle \sigma, e \rangle \rangle$.

The relation $\langle \Sigma, \langle \sigma, e \rangle \rangle \xrightarrow{\alpha} \langle \Sigma', \langle \sigma', e' \rangle \rangle$ represents a single evaluation step from expression e, under the run-time environments Σ and σ, to expression e' and run-time environments Σ' and σ'. We define such relation in terms of a structured operational semantics via evaluation contexts [Felleisen, 1988]. We say that e reduces to e' in one step. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. Symbol α ranges over the *internal* events triggered by expressions (as illustrated in Listing 6 and explained below). We utilize internal events to communicate between the threads and the scheduler. Listing 6 shows the reductions rules for the core contributions in our library. Rules (LAB) and (UNLAB) impose the same security constrains as for the sequential version of `LIO` [Stefan et al., 2011]. Rule (LAB) generates a labeled value if and only if the label is between the current label and clearance of the `LIO` computation. Rule (UNLAB) requires that, when the content of a labeled value is "retrieved" and used in a `LIO` computation, the current label is raised ($\sigma' = \sigma[\texttt{lbl} \mapsto l']$, where $l' = \sigma.\texttt{lbl} \sqcup l$), thus capturing the fact that the remaining computation might depend on e. Output channels are treated as deques of events. We use a standard deque-like interface with operations ($\triangleleft$) and ($\triangleright$) for front and back insertion (respectively), and we also allow pattern-matching in the rules as a representation of deconstruction operations. Rule (OUTPUT)

adds the event $\texttt{out}(v)$ to the end of the output channel at security level l ($\Sigma.\alpha_l \triangleright \texttt{out}(v)$).

The main contributions of our language are related to the primitives for concurrency and synchronization. Rule (LFORK) allows for the creation of a thread and generates the internal event $\texttt{fork}(e)$, where e is the computation to spawn. The rule allocates a new `LMVar` in order to store the result produced by the spawned thread (e `>>=` $\lambda x.\texttt{putLMVar}\ m\ x$). Using that `LMVar`, the rule provides a handle to access to the thread's result (`return` (`R` m)). Rule (LWAIT) simply uses the `LMVar` for the handle. As mentioned in Section 2.4, operations on `LMVar` are *bi-directional* and consequently the rules (NLMVAR), (TLMVAR), and (PLMVAR) require not only that the label of the mentioned `LMVar` be between the current label and current clearance of the thread ($\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}$), but that the current labe be raised appropriately. Considering the security level of a `LMVar` (l), rule (TLMVAR) accordingly raises the current label ($\sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l]$) when emptying ($\Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]$) its content ($\Sigma.\phi(m) = \texttt{Lb}\ l\ e$). Similarly, considering the security level of a `LMVar` (l), rule (PLMVAR) accordingly raises the current label ($\sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l]$) when filling ($\Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]$) its content ($\Sigma.\phi(m) = \texttt{Lb}\ l\ \boxdot$). Finally, rule (GLABR) fetches a labeled `LMVar` from the `LMVar` store ($e = \Sigma.\phi(m)$, i.e., a value of the form $\texttt{Lb}\ l\ m$), and returns its label.

Listing 7 shows the formal semantics for threadpools. The relation $\hookrightarrow$ represents a single evaluation step for the threadpool, in contrast with $\longrightarrow$ which is only for a single thread. We write $\hookrightarrow^*$ for the reflexive and transitive closure of $\hookrightarrow$. As mentioned, configurations are of the form $\langle \Sigma, t_s \rangle$, where Σ is the global runtime environment and t_s is a queue of sequential configurations. The front of the queue is the thread that is currently executing. Threads are scheduled in a round-robin fashion, like GHC. The thread at the front of the queue executes one step, and it is then moved to the back of the queue (see Rule (STEP)). If this step involves a fork (represented by $\xrightarrow{\texttt{fork}(e)}$), a new thread is created at the back of the queue (see Rule (FORK)). Threads are also moved to the back of the threadpool if they are blocked, e.g., waiting to read a value from an empty `LMVar` (see Rule (NO-STEP), we define $\not\longrightarrow$ as the impossibility to make any progress). When a thread finishes, i.e., it can no longer reduce, the final value is placed in the output channel indicated by the current label ($\sigma.\texttt{lbl}$), and the thread is removed from the queue (see Rule (EXIT)).

2.7 Security guarantees

In this section, we show that LIO computations have the property of termination-sensitive non-interference. As in [Li and Zdancewic, 2010, Russo et al., 2008, Stefan et al., 2011], we prove this property by using the

term erasure technique. The erasure function ε_L rewrites data at security levels that the attacker cannot observe into the syntax node $\bullet$.

Listing 8 defines the erasure function ε_L. This function is defined in such a way that $\varepsilon_L(e)$ contains no information above level L, i.e., the function ε_L replaces all the information more sensitive than L in e with a hole ($\bullet$). In most of the cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon_L(e_1\ e_2) = \varepsilon_L(e_1)\ \varepsilon_L(e_2)$). For threadpools, the erasure function is mapped into all sequential configurations; all threads with a current label above L are removed from the pool (filter $(\lambda\langle\sigma, e\rangle.e \not\equiv \bullet)$ (map $\varepsilon_L\ t_s$), where $\equiv$ denotes syntactic equivalence). The computation performed in a certain sequential configuration is erased if the current label is above L. For runtime environments and stores, we map the erasure function into their components. An output channel is erased into the empty channel (ϵ) if it is above L, otherwise the individual output events are erased according to ε_L. Similarly, a labeled value is erased if the label assigned to it is above L.

Following the definition of the erasure function, we introduce a new evaluation relation $\longrightarrow_L$ as follows:

$$\frac{\langle \Sigma, t_s\rangle \longrightarrow \langle \Sigma', t_s'\rangle}{\langle \Sigma, t_s\rangle \longrightarrow_L \varepsilon_L(\langle \Sigma', t_s'\rangle)}$$

The relation $\longrightarrow_L$ guarantees that confidential data, i.e., data not below level L, is erased as soon as it is created. We write $\longrightarrow_L^*$ for the reflexive and transitive closure of $\longrightarrow_L$. Similarly, we introduce a relation $\hookrightarrow_L$ as follows:

$$\frac{\langle \Sigma, t_s\rangle \hookrightarrow \langle \Sigma', t_s'\rangle}{\langle \Sigma, t_s\rangle \hookrightarrow_L \varepsilon_L(\langle \Sigma', t_s'\rangle)}$$

As usual, we write $\hookrightarrow_L^*$ for the reflexive and transitive closure of $\hookrightarrow_L$.

In order to prove non-interference, we will establish a simulation relation between $\hookrightarrow^*$ and $\hookrightarrow_L^*$ through the erasure function: erasing all secret data and then taking evaluation steps in $\hookrightarrow_L$ is equivalent to taking steps in $\hookrightarrow$ first, and then erasing all secret values in the resulting configuration. Note that this relation would not hold if information from some level above L was being leaked by the program. In the rest of this section, we only consider well-typed terms to ensure there are no stuck configurations.

For simplicity, we assume that the space address of the memory store is split into different security levels and that allocation is deterministic. Therefore, the address returned when creating an LMVar with label l depends only on the LMVars with label l already in the store. For the sake of brevity, the proofs have been shortened in this section, but more details can be found in Appendix A.

We start by showing that the evaluation relations $\longrightarrow_L$ and $\hookrightarrow_L$ are deterministic.

Proposition 1 (Determinacy of $\longrightarrow_L$). *If $\langle \Sigma, t\rangle \longrightarrow_L \langle \Sigma', t'\rangle$ and $\langle \Sigma, t\rangle \longrightarrow_L \langle \Sigma'', t''\rangle$, then $\langle \Sigma', t'\rangle = \langle \Sigma'', t''\rangle$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step.

Proposition 2 (Determinacy of $\hookrightarrow_L$). *If $\langle \Sigma, t_s \rangle \hookrightarrow_L \langle \Sigma', t'_s \rangle$ and $\langle \Sigma, t_s \rangle \hookrightarrow_L \langle \Sigma'', t''_s \rangle$, then $\langle \Sigma', t'_s \rangle = \langle \Sigma'', t''_s \rangle$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step and using Proposition 1.

The next lemma establishes a simulation between $\hookrightarrow^*$ and $\hookrightarrow^*_L$.

Lemma 1 (Many-step simulation). *Given a well-typed thread pool t_s (with no* `Lb`*, $()^{LIO}$, $\boxdot$, `R`, and $\bullet$), an attacker at level L, and a runtime environment Σ, if $\langle \Sigma, t_s \rangle \hookrightarrow^* \langle \Sigma', t'_s \rangle$, then $\varepsilon_L(\langle \Sigma, t_s \rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma', t'_s \rangle)$.*

Proof. In order to prove this result, we rely on properties of the erasure function, such as the fact that it is idempotent and homomorphic to the application of evaluation contexts and substitution. We show that the result holds by case analysis on the rule used to derive $\langle \Sigma, t_s \rangle \hookrightarrow^* \langle \Sigma', t'_s \rangle$, and considering different cases for threads whose current label is below (or not) level L. For more details, see Appendix A.

The L-equivalence relation $\approx_L$ is an equivalence relation between configurations (and their parts), defined as the equivalence kernel of the erasure function ε_L: $\langle \Sigma, t_s \rangle \approx_L \langle \Sigma', r_s \rangle$ iff $\varepsilon_L(\langle \Sigma, t_s \rangle) = \varepsilon_L(\langle \Sigma', r_s \rangle)$. If two configurations are L-equivalent, they agree on all data below or at level L, i.e., they cannot be distinguished by an attacker at level L. Note that two queues are L-equivalent iff the threads with current label no higher than L are pairwise L-equivalent in the order that they appear in the queue.

The next theorem shows the non-interference property. It essentially states that if we take two executions of a program with two L-equivalent inputs, then for every intermediate step of the computation of the first run, there is a corresponding step in the computation of the second run which results in an L-equivalent configuration. Note that this also includes the termination channel, since L-equivalence of configurations requires that output channels have matching events, and termination is modelled as a special kind of output event.

Theorem 1 (Termination-sensitive non-interference). *Given a computation e (with no* `Lb`*, $()^{LIO}$, $\boxdot$, `R`, and $\bullet$) where it holds that $\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau \rightarrow \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau')$, an attacker at level L, an initial securiy context σ, and runtime environments Σ_1 and Σ_2 where $\Sigma_1.\phi = \Sigma_2.\phi = \varnothing$ and $\Sigma_1.\alpha_k = \Sigma_2.\alpha_k = \epsilon$ for all levels k, then*

$$\begin{aligned} &\forall e_1 e_2.(\Gamma \vdash e_i : \texttt{Labeled}\ \ell\ \tau)_{i=1,2} \wedge\ e_1 \approx_L e_2 \\ &\qquad \wedge \langle \Sigma_1, \langle \sigma, e\ e_1 \rangle \rangle \hookrightarrow^* \langle \Sigma'_1, t^1_s \rangle \\ &\Rightarrow \exists \Sigma'_2 t^2_s.\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle \hookrightarrow^* \langle \Sigma'_2, t^2_s \rangle \wedge\ \langle \Sigma'_1, t^1_s \rangle \approx_L \langle \Sigma'_2, t^2_s \rangle \end{aligned}$$

Proof. The result follows by combining Lemma 1 and Proposition 2 (Determinacy).

2.8 Example Application: Dating Website

In this section we evaluate the feasibility of leaking information through timing-based covert channels as well as the effectiveness and expressiveness of our extensions to LIO.

We built a simple dating website that allows third-party developers to build applications that interact with a common database. Our website exposes a shared key-value store to third-party apps encoding interested-in relationships. A key correspond to a user ID and its associated value represent the users that he/she is interested in. For simplicity, we do not consider the list of users sensitive, but interested-in relationships should remain confidential. In particular, a user should be able to learn which other users are interested in them, but should not be able to learn the interested-in relationships of other users.

The website consists of two main components: 1) a trusted web server that executes apps written using LIO and 2) untrusted third-party apps that may interact with users and read and write to the database. The database is simply a list of tuples mapping keys (users) to `LMVars` storing lists of users. Apps are separated from each other by URL prefixes. For example, the URL `http://xycombinator.biz/App1` points to `App1`. Requests with a particular app's URL prefix are serviced by invoking the app's request handler in an IFC-constrained, and time-mitigated, environemt. We assume a powerful, but realistic adversary. In particular, malicious application writers may themselves be users of the dating site. We now consider the effectiveness of termination and timing channels in leaking the database.

Termination covert channel As detailed in Section 2.3, the implementation of LIO [Stefan et al., 2011], with `toLabeled`, is susceptible to a termination channel attack. In the context of our dating-website, a malicious application $term$, running on behalf of an (authenticated) user a can be used to leak information on another (target) user t as follows:

- Authenticated adversary a issues a request that contains a guess that user t has an interest in g: `GET /`$term$`?target=`t`&guess=`g
- The trusted app container invokes the app $term$ and forwards the request to it.
- The application $term$ then executes the following LIO code:

```
toLabeled ⊤ $ do v <- lookupDB t
                 if g == v then ⊥ else return ()
return $ mkHtmlResp200 "Bad guess"
```

Here, `lookupDB` t is used to perform a database lookup with key t. If g is present in the database entry, the app will not terminate, otherwise it will respond, denoting the guess was wrong.

We found the termination attack to be very effective. Specifically, we measured the time required to reconstruct a database of 10 users to be 73 seconds[4].

If `toLabeled` is prohibited and `forkLIO` is used instead, the termination attack cannot be mounted. This is because `waitLIO` first raises the label of the app request handler. An attempt to output a response to the client browser will not succeed since the current label of the handler cannot flow to the label of the client's browser. It is important to note that errors of this kind are made indistinguishable from non-terminating requests. To accomplish this, our dating site catches label violation errors and converts them to $\perp$.

Internal timing covert channel To carry out an internal timing attack, an app must execute two threads that share a common resource. Concretely, an app can use internal timing to leak information on a target user t as follows:

- Authenticated adversary a issue a request containing a guess that t is interested-in g: `GET /`$internal$`?target=`t`&guess=`g
- The trusted app container invokes the app $internal$.
- App $internal$ then executes the following LIO code:

```
varHigh <- fork $
  toLabeled ⊤ $ do
    v <- lookupDB t
    if g == v then sleep 5000 else return ()
  appendToAppStorage g
varLow <- fork $ do sleep 3000
                    appendToAppStore -1
wait varHigh
wait varLow
r <- readFromAppStore
return $ mkHtmlResp200 r
```

The code spawns two threads. The first reads the high value in a `toLabeled` then outputs the guess to a low-label store, however, if the guess is correct, it sleeps for five seconds before outputting the guess. The second thread simply outputs a place holder after waiting for three seconds. The result is that the ordering of outputs reveals whether the guess is correct. If the guess is incorrect, the store will read `g,-1`; if the guess is correct, the store will read `-1,g`.

We implemented a magnified version of the attack above by sending several requests to the server. The adversary repeatedly sends requests to $internal$ for each user in the system as a guess g. As with the termination channel attack, we found that internal timing attack is feasible. For a database of 10 users we managed to recover the entries in 66.92 seconds.

[4] All our measurements were conducted on a laptop with a Intel Core i7 2620M (2.7GHz) processor and 8GB of RAM, with GHC 7.4.1.

Our modifications to LIO can be used to address the internal timing attacks described above; replacing `toLabeled` with `forkLIO` eliminates the internal timing leaks. More generally, we observe that by using `forkLIO`, the time when the app writes to the persistent storage (`appendToAppStore`) cannot be influenced by sensitive data. Similarly, replacing `fork` and `wait` by their LIO counterparts renders the attack futile.

External timing covert channel We consider a simple external timing attack to our dating website in which the adversary a has access to a high-precision timer. An app $external$ colluding with a can use external timing to leak a target user t's interested-in relationship as follows:

- Authenticated adversary a issues requests containing the target user t: `GET /`$external$`?target=`t`&guess=`g
- The trusted container invokes $external$ with the request.
- App $external$ then proceeds to execute the following LIO code:

```
toLabeled ⊤ $ do
  v <- lookupDB t
  if g == v then sleep 5000 else return ()
return $ mkHtmlResp200 "done"
```

The attack is a component of the internal timing attack: given a target t and guess g, if the g was correct the thread sleeps; otherwise it does nothing. The attacker simply measures the response time – recognizing a delay as a correct guess.

Despite its simplicity, we also found this attack to be plausible. In 33 seconds, we recovered a database of 10 users. Addressing this attack we mitigated the app handler, as described in Section 2.5. The response time of an app is mitigated, taking into account the arrival of a request. Although we manged to recover 3 of the 10 user entries in 64 seconds—we found that recovering the remaining user entries was infeasible. Of course, the performance of well-behaved apps was unaffected.

2.9 Related Work

IFC security libraries The seminal work by Li and Zdancewic [Li and Zdancewic, 2006] presents an implementation of information-flow security as a library using a generalization of monads called Arrows [Hughes, 2000]. Following this line of work, Tsai et al. [Tsai et al., 2007] further consider side-effects and concurrency. Different from our approach, Tsai et al. provide termination-insensitive non-interference under a cooperative scheduler and no synchronization primitives. Russo et al. [Russo et al., 2008] eliminate the need for Arrows by showing an IFC security library based solely on monads. Their library leverages Haskell's type-system to statically enforce non-interference. Jaskelioff and Russo [Jaskelioff and Russo,

2011] propose a library that enforces non-interference by executing the program as many times as security levels, which is known as secure multi-execution [Devriese and Piessens, 2010]. Recently, Stefan et al. propose the use of the monad `LIO` to track information-flow dynamically [Stefan et al., 2011]. Morgenstern et al. [Morgenstern and Licata, 2010] encoded an authorization- and IFC-aware programming language in Agda. Their encoding, however, does not consider computations with side-effects. Devriese and Piessens [Devriese and Piessens, 2011] used monad transformers and parametrized monads [Atkey, 2006] to enforce non-interference, both dynamically and statically. None of the approaches mentioned above deals with the termination channel. Moreover, none of them (except from Tsai et al.) handle concurrency.

Internal timing covert channel There are several approaches to deal with the internal timing covert channel. The work in [Smith and Volpano, 1998, Volpano and Smith, 1999, Smith, 2001, 2003] relies on the non-realistic primitive `protect(c)` which, by definition, hides the timing behaviour of `c`. Our approach, on the other hand, relies on the fork primitive and the semantics for mutable locations. Assuming a scenario where it is possible to modify the scheduler, the work in [Russo and Sabelfeld, 2006a, Barthe et al., 2007] propose a novel interaction between threads and the scheduler that is able to implement a generalized version of `protect(c)`. A series of work [Zdancewic and Myers, 2003, Huisman et al., 2006, Terauchi, 2008] prevents internal timing leaks by avoiding any races on public data. Boudol and Castellani [Boudol and Castellani, 2001, 2002] avoid internal timing leaks by disallowing public events after branching on secret data. The authors consider a fixed number of threads and no synchronization primitives. Russo and Sabelfeld [Russo and Sabelfeld, 2006b] show how to remove internal timing leaks under a cooperative scheduling by manipulating yield commands. The termination channel is intrinsically present under cooperative scheduling, i.e., there is no way to decouple executions between threads. The work by Russo et al. [Russo et al., 2006] is the closest one to our approach to internal timing leaks. In that work, the authors introduce a code transformation, from a sequential program into a concurrent one, that spawns threads to execute branches and loops whose conditionals depend on secret values. The idea of spawning threads when computations use secrets is similar to ours, but it is used in a quite different context. Firstly, Russo et al. apply their technique for a simple sequential while-language, while we consider concurrent programs with synchronization primitives. Secondly, and different from our work, their approach does not consider leaks due to termination, i.e., the transformation guarantess termination-insensitive non-interference. Finally, incurring high synchronization costs, the code transformation introduces synchronization between spawned threads in order to preserve the semantics of the original sequential program. The transformation might change the termi-

nating behavior of programs in order to preserve security. Our proposal, on the other hand, guarantees that the semantics of the program is the one that the programmer writes in the code.

Termination and external covert channels There are several language-based mechanisms to tackle the termination and external timing channels. Volpano [Volpano and Smith, 1997] describes a type-system that removes the termination channel by forbidding loops whose conditional depend on secrets. The work by Hedin and Sands [Hedin and Sands, 2005] avoids the termination and external timing covert channels for sequential Java bytecode by disallowing outputs after branching on secrets. Similarly, `LIO` computations do not allow public outputs after observing secret data. However, the programmer can spawn new threads to perform such computations and thus allowing the rest of the system to still perform public outputs. Agat [Agat, 2000] describes a code transformation that removes external timing leaks by padding programs with dummy computations. The termination channel is closed by disallowing loops on secrets. One drawback of Agat's transformation is that if there is an if-then-else, whose guard depends on secret data, and only one of its branches is non-terminating, then the transformed program becomes non-terminating. This approach has been adapted for languages with concurrency [Sabelfeld and Sands, 2000, Sabelfeld, 2001, Sabelfeld and Mantel, 2002]. Moreover, the transformation has been rephrased as a unification problem [Köpf and Mantel, 2006] as well as being implemented using transactions [Barthe et al., 2006]. While targeting sequential programs, secure multi-execution [Devriese and Piessens, 2010] removes both the termination and external timing channel. However, the latter is only closed if there are as many CPUs (or cores) as security levels being considered by the technique. We refer the reader to [Kashyap et al., 2011] for a more detailed description of possible enforcements for timing- and termination-sensitive non-interference. Recently, Zhang et al. [Zhang et al., 2012] propose a language-based mitigation approach for a simple while-language extended with a `mitigate` primitive. Their work relies on static annotations to provide information about the underlying hardware. Compared to their work, our functional approach is more general and can be extended to address other covert channels (e.g., storage). However, their attack model is more powerful in considering the effects of hardware (e.g., cache). Nevertheless, we find their work to be complimentary: our system can leverage static annotations and the Xenon "no-fill" mode to address attacks relying on underlying hardware.

2.10 Summary

Many information flow control systems allow applications to sequence code with publicly visible side-effects after code computing over private

data. Unfortunately, such sequencing leaks private data through termination channels (which affect whether the public side-effects ever happen), internal timing channels (which affect the order of publicly visible side-effects), and external timing channels (which affect the response time of visible side-effects). Such leaks are far worse in the presence of concurrency, particularly when untrusted code can spawn new threads.

We demonstrate that such sequencing can be avoided by introducing additional concurrency when public values must reference the results of computations over private data. We implemented this idea in an existing Haskell information flow library, LIO. In addition, we show how our library is amenable to mitigating external timing attacks by quantizing the appearance of externally visible side-effects. To evaluate our ideas, we prototyped the core of a dating web site showing that our interfaces are practical and our implementation does indeed mitigate these covert channels.

Listing 6 Semantics for non-standard expressions.

$$
\begin{aligned}
E \quad ::= \quad & \ldots \mid \texttt{label}\ E\ e \mid \texttt{unlabel}\ E \mid \texttt{out}\ E\ e \mid \texttt{out}\ l\ E \\
& \mid \texttt{forkLIO}\ E\ e \mid \texttt{newLMVar}\ E\ e \mid \texttt{takeLMVar}\ E \\
& \mid \texttt{putLMVar}\ E\ e \mid \texttt{labelOfLMVar}\ E
\end{aligned}
$$

(LAB)
$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}}{\langle \Sigma, \langle \sigma, E[\texttt{label}\ l\ e] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)] \rangle \rangle}$$

(UNLAB)
$$\frac{l' = \sigma.\texttt{lbl} \sqcup l \qquad l' \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{lbl} \mapsto l']}{\langle \Sigma, \langle \sigma, E[\texttt{unlabel}\ (\texttt{Lb}\ l\ e)] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma', E[\texttt{return}\ e] \rangle \rangle}$$

(OUTPUT)
$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\alpha_l \mapsto \Sigma.\alpha_l \triangleright \texttt{out}(v)]}{\langle \Sigma, \langle \sigma, E[\texttt{out}\ l\ v] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma, E[\texttt{return}\ ()] \rangle \rangle}$$

(LFORK)
$$\frac{\begin{array}{c}\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]] \\ \alpha = e\ \texttt{>>=}\ \lambda x.\texttt{putLMVar}\ m\ x \qquad m\ \textit{fresh}\end{array}}{\langle \Sigma, \langle \sigma, E[\texttt{forkLIO}\ l\ e] \rangle \rangle \overset{\texttt{fork}(e)}{\longrightarrow} \langle \Sigma', \langle \sigma, E[\texttt{return}\ (\texttt{R}\ m)] \rangle \rangle}$$

(LWAIT)
$$\langle \Sigma, \langle \sigma, E[\texttt{waitLIO}\ (\texttt{R}\ m)] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{takeLMVar}\ m] \rangle \rangle$$

(NLMVAR)
$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]] \qquad m\ \textit{fresh}}{\langle \Sigma, \langle \sigma, E[\texttt{newLMVar}\ l\ e] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma, E[\texttt{return}\ m] \rangle \rangle}$$

(TLMVAR)
$$\frac{\begin{array}{c}\Sigma.\phi(m) = \texttt{Lb}\ l\ e \qquad \sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \\ \sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l] \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]]\end{array}}{\langle \Sigma, \langle \sigma, E[\texttt{takeLMVar}\ m] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma', E[\texttt{return}\ e] \rangle \rangle}$$

(PLMVAR)
$$\frac{\begin{array}{c}\Sigma.\phi(m) = \texttt{Lb}\ l\ \boxdot \qquad \sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \\ \sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l] \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]]\end{array}}{\langle \Sigma, \langle \sigma, E[\texttt{putLMVar}\ m\ e] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma', E[\texttt{return}\ ()] \rangle \rangle}$$

(GLABR)
$$\frac{e = \Sigma.\phi(m)}{\langle \Sigma, \langle \sigma, E[\texttt{labelOfLMVar}\ m] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{labelOf}\ e] \rangle \rangle}$$

Listing 7 Semantics for threadpools.

$$\frac{\langle \Sigma, t\rangle \longrightarrow \langle \Sigma', t'\rangle}{\langle \Sigma, t \triangleleft t_s\rangle \hookrightarrow \langle \Sigma', t_s \triangleright t'\rangle}\ (\textsc{step}) \qquad \frac{\langle \Sigma, t\rangle \not\longrightarrow}{\langle \Sigma, t \triangleleft t_s\rangle \hookrightarrow \langle \Sigma, t_s \triangleright t\rangle}\ (\textsc{no-step})$$

$$\frac{\langle \Sigma, t\rangle \xrightarrow{\texttt{fork}(e)} \langle \Sigma', \langle \sigma, e'\rangle\rangle \qquad t_{\text{new}} = \langle \sigma, e\rangle}{\langle \Sigma, t \triangleleft t_s\rangle \hookrightarrow \langle \Sigma', t_s \triangleright \langle \sigma, e'\rangle \triangleright t_{\text{new}}\rangle}\ (\textsc{fork})$$

$$\frac{l = \sigma.\texttt{lbl} \qquad \Sigma' = \Sigma[\alpha_l \mapsto \Sigma.\alpha_l \triangleright \texttt{exit}(v)]}{\langle \Sigma, \langle \sigma, v\rangle \triangleleft t_s\rangle \hookrightarrow \langle \Sigma', t_s\rangle}\ (\textsc{exit})$$

Listing 8 Erasure function.

$$\varepsilon_L(\langle \Sigma, t_s\rangle) = \langle \varepsilon_L(\Sigma), \text{filter } (\lambda\langle \sigma, e\rangle.e \notin \bullet)\ (\text{map } \varepsilon_L\ t_s)\rangle$$

$$\varepsilon_L(\langle \sigma, e\rangle) = \begin{cases} \langle \sigma, \bullet\rangle & \sigma.\texttt{lbl} \notin L \\ \langle \sigma, \varepsilon_L(e)\rangle & \text{otherwise} \end{cases}$$

$$\varepsilon_L(\Sigma) = \Sigma[\phi \mapsto \varepsilon_L(\Sigma.\phi)][\alpha_l \mapsto \varepsilon_L(\alpha_l)]_{l \in \text{Labels}}$$

$$\varepsilon_L(\alpha_l) = \begin{cases} \epsilon & l \notin L \\ \text{map } \varepsilon_L\ \alpha_l & \text{otherwise} \end{cases}$$

$$\varepsilon_L(\phi) = \{(x, \varepsilon_L(\phi(x))) : x \in \text{dom}(\phi)\}$$

$$\varepsilon_L(\texttt{Lb}\ l\ e) = \begin{cases} \texttt{Lb}\ l\ \bullet & l \notin L \\ \texttt{Lb}\ l\ \varepsilon_L(e) & \text{otherwise} \end{cases}$$

In the rest of the cases, ε_L is homomorphic.

BIBLIOGRAPHY

J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, January 2000.

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. of the 13th ESORICS*. Springer-Verlag, 2008.

Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM CCS*, CCS '10. ACM, 2010.

Robert Atkey. Parameterised notions of computation. In *Workshop on mathematically structured functional programming, ed. Conor McBride and Tarmo Uustalu. Electronic Workshops in Computing, British Computer Society*, pages 31–45, 2006.

G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *Proc. European Symposium on Research in Computer Security (ESORICS)*, pages 2–18, September 2007.

Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153, May 2006.

Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07. ACM, 2007.

Boudol and Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*. Springer-Verlag, July 2001.

Gérard Boudol and Ilaria Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1), June 2002.

D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*, pages 1–1. USENIX Association, 2003.

D. Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10(4), 1997.

Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

Dominique Devriese and Frank Piessens. Information flow enforcement in monadic libraries. In *Proc. of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM, 2011.

Mattias Felleisen. The theory and practice of first-class prompts. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, pages 180–190. ACM, 1988.

Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and communications security*, CCS '00. ACM, 2000.

Helena Handschuh and Howard M. Heys. A timing attack on RC5. In *Proc. of the Selected Areas in Cryptography*. Springer-Verlag, 1999.

D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Proc. of the 2011 Marktoberdorf Summer School*. IOS Press, 2011.

Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. *Elec. Notes Theor. Comput. Sci.*, 141, 2005.

J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, July 2006.

M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2011.

Simon P. Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996.

V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE, 2011.

Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. of the 16th CRYPTO*. Springer-Verlag, 1996.

B. Köpf and H. Mantel. Eliminating implicit information leaks by transformational typing and unification. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'05)*, volume 3866 of *LNCS*. Springer-Verlag, July 2006.

Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *In Proceedings of the 22nd ACM Symposium on Principles of Programming Languages. ACMPress*, 1995.

Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently typed programming. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.

A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.

A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 177–189, July 2006a.

A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (PSI)*, LNCS. Springer-Verlag, June 2006b.

A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Proc. of Asian Computing Science Conference (ASIAN)*, LNCS. Springer-Verlag, December 2006.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN Symposium on Haskell (HASKELL)*, pages 13–24. ACM Press, September 2008.

A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, September 2002.

A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 200–214, July 2000.

V. Simonet. The Flow Caml system. Software release at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 3–13, 2003.

G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, June 2001.

G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, January 1998.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

Tachio Terauchi. A type system for observational determinism. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 287–300. IEEE Computer Society, 2008.

T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. IEEE Computer Security Foundations Symposium (CSF)*, July 2007.

D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3), November 1999.

Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE workshop on Computer Security Foundations*, CSFW '97. IEEE Computer Society, 1997.

Wing H. Wong. Timing attacks on RSA: revealing your secrets through the fourth dimension. *Crossroads*, 11, May 2005.

S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 29–43, June 2003.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proc. of the 18th ACM CCS*. ACM, 2011.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proc. of PLDI*. ACM, 2012.

A Detailed proofs

In this section, we provide more details about the proofs for the results in Section 2.7 as well as some auxiliary lemmas.

The following lemmas are necessary to prove that $\longrightarrow_L^*$ and $\hookrightarrow_L^*$ are deterministic.

Proposition 3 (Determinacy of $\longrightarrow$). *If $\langle \Sigma, t_s \rangle \longrightarrow \langle \Sigma', t_s' \rangle$ and $\langle \Sigma, t_s \rangle \longrightarrow \langle \Sigma'', t_s'' \rangle$, then $\langle \Sigma', t_s' \rangle = \langle \Sigma'', t_s'' \rangle$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step.

Proposition 1 (Determinacy of $\longrightarrow_L$). *If $\langle \Sigma, t \rangle \longrightarrow_L \langle \Sigma', t' \rangle$ and $\langle \Sigma, t \rangle \longrightarrow_L \langle \Sigma'', t'' \rangle$, then $\langle \Sigma', t' \rangle = \langle \Sigma'', t'' \rangle$.*

Proof. By Proposition 3 and definition of ε_L.

Proposition 4 (Determinacy of $\hookrightarrow$). *If $\langle \Sigma, t_s \rangle \hookrightarrow \langle \Sigma', t_s' \rangle$ and $\langle \Sigma, t_s \rangle \hookrightarrow \langle \Sigma'', t_s'' \rangle$, then $\langle \Sigma', t_s' \rangle = \langle \Sigma'', t_s'' \rangle$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step, and using Proposition 3.

Proposition 2 (Determinacy of $\hookrightarrow_L$). *If $\langle \Sigma, t_s \rangle \hookrightarrow_L \langle \Sigma', t_s' \rangle$ and $\langle \Sigma, t_s \rangle \hookrightarrow_L \langle \Sigma'', t_s'' \rangle$, then $\langle \Sigma', t_s' \rangle = \langle \Sigma'', t_s'' \rangle$.*

Proof. By Proposition 4 and the definition of ε_L.

The following proposition shows that the erasure function is homomorphic to the application of evaluation contexts and substitution, and that it is idempotent.

Proposition 5 (Properties of erasure function).

1. $\varepsilon_L(E[e]) = \varepsilon_L(E)[\varepsilon_L(e)]$
2. $\varepsilon_L([e_2/x]e_1) = [\varepsilon_L(e_2)/x]\varepsilon_L(e_1)$
3. $\varepsilon_L(\varepsilon_L(e)) = \varepsilon_L(e)$
4. $\varepsilon_L(\varepsilon_L(E)) = \varepsilon_L(E)$
5. $\varepsilon_L(\varepsilon_L(\Sigma)) = \varepsilon_L(\Sigma)$
6. $\varepsilon_L(\varepsilon_L(\langle \sigma, e \rangle)) = \varepsilon_L(\langle \sigma, e \rangle)$
7. $\varepsilon_L(\varepsilon_L(t_s)) = \varepsilon_L(t_s)$
8. $\varepsilon_L(\varepsilon_L(\langle\!\langle \Sigma, t_s \rangle\!\rangle)) = \varepsilon_L(\langle\!\langle \Sigma, t_s \rangle\!\rangle)$

Proof. All follow from the definition of the erasure function ε_L, and by induction on expressions and evaluation contexts.

Most of the reduction rules in Listing 6 will change the runtime environment. In addition, these transformations usually depend on a given expression, e.g. $\Sigma \mapsto \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto e]]$ can be seen as a function of e. We will represent these runtime transformations as functions $h : e \times \Sigma \to \Sigma$, where e is the set of expressions and Σ is the set of runtime environments. We will also write $h_e : \Sigma \to \Sigma$ for the partial application of h to an expression e. We extend this notation to transformations of stores and output channels.

We say that a transformation $f : e \times A \to A$ is *L-independent* if the secrets introduced in structure A by the application of f_e cannot be observed by an attacker at level L, i.e.

$$\varepsilon_L \circ f_e = \varepsilon_L \circ f_{\varepsilon_L(e)} \circ \varepsilon_L.$$

The next lemma is useful in proving that a given environment transformation is L-independent, by showing that its corresponding store and output channel transformations are L-independent.

Lemma 2. *Let h_e be a transformation for runtime environments that depends on an expression e, given as $h_e(\Sigma) = \Sigma[\phi \mapsto f_e(\Sigma.\phi)][\alpha_l \mapsto g_l^e(\Sigma.\alpha_l)]$ and thus uniquely determined by functions f_e and g_l^e for every label l and expression e. If f and g_l are all L-independent, then h is L-independent.*

Proof.

$$\begin{aligned}
&\varepsilon_L(h_{\varepsilon_L(e)}(\varepsilon_L(\Sigma))) \\
&= \varepsilon_L(\varepsilon_L(\Sigma)[\phi \mapsto f_{\varepsilon_L(e)}(\varepsilon_L(\Sigma).\phi)] \\
&\qquad [\alpha_l \mapsto g_l^{\varepsilon_L(e)}(\varepsilon_L(\Sigma).\alpha_l)]) \\
&= \varepsilon_L(\varepsilon_L(\Sigma)[\phi \mapsto f_{\varepsilon_L(e)}(\varepsilon_L(\Sigma.\phi))] \\
&\qquad [\alpha_l \mapsto g_l^{\varepsilon_L(e)}(\varepsilon_L(\Sigma.\alpha_l))]) \\
&= \varepsilon_L(\varepsilon_L(\Sigma))[\phi \mapsto \varepsilon_L(f_{\varepsilon_L(e)}(\varepsilon_L(\Sigma.\phi)))] \\
&\qquad [\alpha_l \mapsto \varepsilon_L(g_l^{\varepsilon_L(e)}(\varepsilon_L(\Sigma.\alpha_l)))] \\
&= \varepsilon_L(\Sigma)[\phi \mapsto \varepsilon_L(f_e(\Sigma.\phi))][\alpha_l \mapsto \varepsilon_L(g_l^e(\Sigma.\alpha_l))] \\
&= \varepsilon_L(\Sigma)[\phi \mapsto f_e(\Sigma.\phi)][\alpha_l \mapsto g_l^e(\Sigma.\alpha_l)] \\
&= \varepsilon_L(h_e(\Sigma))
\end{aligned}$$

The next lemma shows that the environment transformations in the reduction rules are all L-independent.

Lemma 3. *All runtime transformations h_e in the reduction rules in Listing 6 are L-independent.*

Proof. There are two cases to consider: modifications to the store (ϕ), which only update the contents of one reference, or appending a value to an output channel.

- **Case $h_e(\Sigma) = \Sigma[\phi \mapsto f_e(\Sigma.\phi)]$, with $f_e(\phi) = \phi[m \mapsto \texttt{Lb}\ l\ e]$.** By Lemma 2, we only have to prove that f is L-independent. We consider two cases:

- $l \sqsubseteq L$:
  $$\begin{aligned}&\varepsilon_L(f_{\varepsilon_L(e)}(\varepsilon_L(\phi)))\\ &= \varepsilon_L(\varepsilon_L(\phi)[m \mapsto \texttt{Lb}\ l\ \varepsilon_L(e)])\\ &= \varepsilon_L(\varepsilon_L(\phi[m \mapsto \texttt{Lb}\ l\ e]))\\ &= \varepsilon_L(f_e(\phi))\end{aligned}$$
- $l \not\sqsubseteq L$:
  $$\begin{aligned}&\varepsilon_L(f_{\varepsilon_L(e)}(\varepsilon_L(\phi)))\\ &= \varepsilon_L(\varepsilon_L(\phi)[m \mapsto \texttt{Lb}\ l\ \varepsilon_L(e)])\\ &= \varepsilon_L(\varepsilon_L(\phi))[m \mapsto \texttt{Lb}\ l\ \bullet]\\ &= \varepsilon_L(\phi)[m \mapsto \texttt{Lb}\ l\ \bullet]\\ &= \varepsilon_L(f_e(\phi))\end{aligned}$$

▸ **Case** $h_e(\Sigma) = \Sigma[\alpha_l \mapsto g_l^e(\Sigma.\alpha_l)]$ **with** $g_l^e(\alpha) = \alpha \triangleright e$. By Lemma 2, we only have to prove that g_l is L-independent.
$$\begin{aligned}&\varepsilon_L(g_l^{\varepsilon_L(e)}(\varepsilon_L(\alpha)))\\ &= \varepsilon_L(\varepsilon_L(\alpha) \triangleright \varepsilon_L(e))\\ &= \varepsilon_L(\varepsilon_L(\alpha \triangleright e))\\ &= \varepsilon_L(g_l^e(\alpha))\end{aligned}$$

▸ The rest of the cases are similar.

The following lemma establishes a simulation between $\longrightarrow$ and $\longrightarrow_L$ when reducing the body of a thread whose current label is below or equal to level L.

Lemma 4 (Single-step simulation for public computations). *If* $\langle \Sigma, \langle \sigma, t\rangle\rangle \longrightarrow \langle \Sigma', t'\rangle$ *with* $\sigma.\texttt{lbl} \sqsubseteq L$, *then* $\varepsilon_L(\langle \Sigma, \langle \sigma, t\rangle\rangle) \longrightarrow_L \varepsilon_L(\langle \Sigma', t'\rangle)$.

$$\begin{array}{ccc}\langle \Sigma, \langle \sigma, e\rangle\rangle & \longrightarrow & \langle \Sigma^1, t'\rangle \\ \Big\downarrow{\scriptstyle \varepsilon_L} & & \Big\downarrow{\scriptstyle \varepsilon_L} \\ \varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle\rangle) & \xrightarrow[L]{} & \varepsilon_L(\langle \Sigma^1, t'\rangle)\end{array}$$

Proof. The proof is by case analysis on the rule used to derive $\langle \Sigma, \langle \sigma, t\rangle\rangle \longrightarrow \langle \Sigma', t'\rangle$. As shown in Lemma 3, all environment modifications are consistent with the simulation: erasing secret data and then modifying the environment with erased data is equivalent to modifying the environment and then erasing the secrets.

▸ Case $t = E[\texttt{forkLIO}\ l\ e]$
$$\begin{aligned}&\varepsilon_L(\langle \Sigma, \langle \sigma, E[\texttt{forkLIO}\ l\ e]\rangle\rangle)\\ &= \langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(E)[\texttt{forkLIO}\ l\ \varepsilon_L(e)]\rangle\rangle\\ &\xrightarrow[L]{\text{fork}(\alpha)} \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \langle \sigma, \varepsilon_L(E)[\texttt{return ()}]\rangle\rangle)\\ &= \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \varepsilon_L(\langle \sigma, E[\texttt{return ()}]\rangle)\rangle)\\ &\quad \text{(by Lemma 3, } \varepsilon_L(\Sigma^1) = \varepsilon_L(\Sigma'))\\ &= \varepsilon_L(\langle \Sigma', \langle \sigma, E[\texttt{return ()}]\rangle\rangle)\end{aligned}$$

- Case $t = E[\texttt{out}\ l\ e]$

$$\begin{aligned}
&\varepsilon_L(\langle\!\langle \Sigma, \langle \sigma, E[\texttt{out}\ l\ e]\rangle\rangle\!\rangle)\\
&= \langle\!\langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(E)[\texttt{out}\ l\ \varepsilon_L(e)]\rangle\rangle\!\rangle\\
&\longrightarrow_L \varepsilon_L(\langle\!\langle \varepsilon_L(\Sigma^1), \langle \sigma, \varepsilon_L(E)[\texttt{return}\ ()]\rangle\rangle\!\rangle)\\
&= \varepsilon_L(\langle\!\langle \varepsilon_L(\Sigma^1), \varepsilon_L(\langle \sigma, E[\texttt{return}\ ()]\rangle)\rangle\!\rangle)\\
&= \varepsilon_L(\langle\!\langle \Sigma', \langle \sigma, E[\texttt{return}\ ()]\rangle\rangle\!\rangle)
\end{aligned}$$

- Case $t = E[\texttt{takeLMVar}\ m]$

$$\begin{aligned}
&\varepsilon_L(\langle\!\langle \Sigma, \langle \sigma, E[\texttt{takeLMVar}\ m]\rangle\rangle\!\rangle)\\
&= \langle\!\langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(E)[\texttt{takeLMVar}\ m]\rangle\rangle\!\rangle\\
&\longrightarrow_L \varepsilon_L(\langle\!\langle \varepsilon_L(\Sigma^1), \langle \sigma', \varepsilon_L(E)[\texttt{return}\ \varepsilon_L(e)]\rangle\rangle\!\rangle)
\end{aligned}$$

  Note that now $\sigma'.\texttt{lbl} = l$. We consider two cases:
  - $l \sqsubseteq L$:

$$\begin{aligned}
&\varepsilon_L(\langle\!\langle \varepsilon_L(\Sigma^1), \langle \sigma', \varepsilon_L(E)[\texttt{return}\ \varepsilon_L(e)]\rangle\rangle\!\rangle)\\
&= \varepsilon_L(\langle\!\langle \Sigma', \langle \sigma', E[\texttt{return}\ e]\rangle\rangle\!\rangle)
\end{aligned}$$

  - $l \not\sqsubseteq L$:

$$\begin{aligned}
&\varepsilon_L(\langle\!\langle \varepsilon_L(\Sigma^1), \langle \sigma', \varepsilon_L(E)[\texttt{return}\ \varepsilon_L(e)]\rangle\rangle\!\rangle)\\
&= \langle\!\langle \varepsilon_L(\Sigma'), \langle \sigma', \bullet\rangle\rangle\!\rangle\\
&= \varepsilon_L(\langle\!\langle \Sigma', \langle \sigma', E[\texttt{return}\ e]\rangle\rangle\!\rangle)
\end{aligned}$$

  In both cases, it follows that $\varepsilon_L(\Sigma^1) = \varepsilon_L(\Sigma')$ by Lemma 3.
- Trivially reduces to the $t = E[\texttt{takeLMVar}\ m]$ case.
- Case $t = E[\texttt{newLMVar}\ l\ e]$.

$$\begin{aligned}
&\varepsilon_L(\langle\!\langle \Sigma, \langle \sigma, E[\texttt{newLMVar}\ l\ e]\rangle\rangle\!\rangle)\\
&= \langle\!\langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(E)[\texttt{newLMVar}\ l\ e]\rangle\rangle\!\rangle\\
&\longrightarrow_L \varepsilon_L(\langle\!\langle \Sigma^1, \langle \sigma, \varepsilon_L(E)[\texttt{return}\ m]\rangle\rangle\!\rangle)\\
&= \varepsilon_L(\langle\!\langle \Sigma^1, \varepsilon_L(\langle \sigma, E[\texttt{return}\ m]\rangle)\rangle\!\rangle)\\
&= \varepsilon_L(\langle\!\langle \Sigma', \langle \sigma, E[\texttt{return}\ m]\rangle\rangle\!\rangle)
\end{aligned}$$

- Case $t = E[\texttt{putLMVar}\ m\ e]$.

$$\begin{aligned}
&\varepsilon_L(\langle\!\langle \Sigma, \langle \sigma, E[\texttt{putLMVar}\ m\ e]\rangle\rangle\!\rangle)\\
&= \langle\!\langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(E)[\texttt{putLMVar}\ m\ \varepsilon_L(e)]\rangle\rangle\!\rangle\\
&\longrightarrow_L \varepsilon_L(\langle\!\langle \Sigma^1, \langle \sigma, \varepsilon_L(E)[\texttt{return}\ ()]\rangle\rangle\!\rangle)\\
&= \varepsilon_L(\langle\!\langle \Sigma^1, \varepsilon_L(\langle \sigma, E[\texttt{return}\ ()]\rangle)\rangle\!\rangle)\\
&= \varepsilon_L(\langle\!\langle \Sigma', \langle \sigma, E[\texttt{return}\ ()]\rangle\rangle\!\rangle)
\end{aligned}$$

- The rest of the cases are similar.

The following lemma establishes a simulation between $\hookrightarrow$ and $\hookrightarrow_L$ when reducing the body of a thread whose current label is below or equal to level L.

Lemma 5 (Single-step simulation for public computations). *If* $\langle\!\langle \Sigma, \langle \sigma, t\rangle \triangleleft t_s\rangle\!\rangle \hookrightarrow \langle\!\langle \Sigma', t'_s\rangle\!\rangle$ *with* $\sigma.\texttt{lbl} \sqsubseteq L$*, then* $\varepsilon_L(\langle\!\langle \Sigma, \langle \sigma, t\rangle \triangleleft t_s\rangle\!\rangle) \hookrightarrow_L \varepsilon_L(\langle\!\langle \Sigma', t'_s\rangle\!\rangle)$.

$$\begin{array}{ccc}
\langle\!\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle\!\rangle & \hookrightarrow & \langle\!\langle \Sigma', t'_s\rangle\!\rangle \\
\Big\downarrow \varepsilon_L & & \Big\downarrow \varepsilon_L \\
\varepsilon_L(\langle\!\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle\!\rangle) & \hookrightarrow_L & \varepsilon_L(\langle\!\langle \Sigma', t'_s\rangle\!\rangle)
\end{array}$$

Proof. The proof is by case analysis on the rule used to derive $\langle \Sigma, \langle \sigma, t\rangle \triangleleft t_s\rangle \hookrightarrow \langle \Sigma', t'_s\rangle$.

- Case (STEP). By Lemma 4, we know that $\varepsilon_L(\langle \Sigma, t\rangle) \longrightarrow_L \varepsilon_L(\langle \Sigma', t'\rangle)$, so $\langle \varepsilon_L(\Sigma), \varepsilon_L(t) \triangleleft \varepsilon_L(t_s)\rangle \hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma'), \varepsilon_L(t_s) \triangleright \varepsilon_L(t')\rangle)$.
$$\begin{aligned} & \varepsilon_L(\langle \Sigma, t \triangleleft t_s\rangle) \\ &= \langle \varepsilon_L(\Sigma), \varepsilon_L(t) \triangleleft \varepsilon_L(t_s)\rangle \\ &\hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma'), \varepsilon_L(t_s) \triangleright \varepsilon_L(t')\rangle) \\ &= \varepsilon_L(\langle \Sigma', t_s \triangleright t'\rangle) \end{aligned}$$
- Case (NO-STEP).
$$\begin{aligned} & \varepsilon_L(\langle \Sigma, t \triangleleft t_s\rangle) \\ &= \langle \varepsilon_L(\Sigma), \varepsilon_L(t) \triangleleft \varepsilon_L(t_s)\rangle \\ &\hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma), \varepsilon_L(t_s) \triangleright \varepsilon_L(t)\rangle) \\ &= \varepsilon_L(\langle \Sigma, t_s \triangleright t\rangle) \end{aligned}$$
- Case (FORK).
$$\begin{aligned} & \varepsilon_L(\langle \Sigma, \langle \sigma, t\rangle \triangleleft t_s\rangle) \\ &= \langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(t)\rangle \triangleleft \varepsilon_L(t_s)\rangle \\ &\hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma'), \varepsilon_L(t_s) \triangleright \langle \sigma, \varepsilon_L(t')\rangle \triangleright t_{\text{new}}\rangle) \\ &= \langle \varepsilon_L(\Sigma'), \varepsilon_L(t_s \triangleright \langle \sigma, t'\rangle \triangleright t_{\text{new}})\rangle \\ &= \varepsilon_L(\langle \Sigma', t_s \triangleright \langle \sigma, t'\rangle \triangleright t_{\text{new}}\rangle) \end{aligned}$$
- Case (EXIT).
$$\begin{aligned} & \varepsilon_L(\langle \Sigma, \langle \sigma, v\rangle \triangleleft t_s\rangle) \\ &= \langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(v)\rangle \triangleleft \varepsilon_L(t_s)\rangle \\ &\hookrightarrow_L \varepsilon_L(\langle \Sigma^1, \varepsilon_L(t_s)\rangle) \\ &= \varepsilon_L(\langle \Sigma', t_s\rangle) \end{aligned}$$

We can also show that initial and final configurations for any reduction steps taken from a thread above L are equal when erased.

Lemma 6. *If* $\langle \Sigma, \langle \sigma, e\rangle\rangle \longrightarrow \langle \Sigma^1, t'\rangle$ *with* $\sigma.\mathtt{lbl} \not\sqsubseteq L$, *then* $\varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle\rangle) = \varepsilon_L(\langle \Sigma^1, t'\rangle)$, *i.e.,*

$$\begin{array}{ccc} \langle \Sigma, \langle \sigma, e\rangle\rangle & \longrightarrow & \langle \Sigma^1, t'\rangle \\ \downarrow{\scriptstyle \varepsilon_L} & & \downarrow{\scriptstyle \varepsilon_L} \\ \varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle\rangle) & = & \varepsilon_L(\langle \Sigma^1, t'\rangle) \end{array}$$

Proof. Since $\varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle\rangle) = \langle \varepsilon_L(\Sigma), \langle \sigma, \bullet\rangle\rangle$, we only have to show that $\varepsilon_L(\Sigma) = \varepsilon_L(\Sigma^1)$, where Σ^1 is the modified environment after performing the reduction step. The proof is similar to L-independence for the simulation lemma: for an arbitrary environment transformation h_e, we have to prove that $\varepsilon_L \circ h_e = \varepsilon_L$.

- **Case** $h_e(\Sigma) = \Sigma[\phi \mapsto f_e(\Sigma.\phi)]$, **with** $f_e(\phi) = \phi[m \mapsto \mathtt{Lb}\ l\ e]$. We prove that $\varepsilon_L \circ f_e = \varepsilon_L$.

$$
\begin{aligned}
&\varepsilon_L(f_e(\phi)) \\
&= \varepsilon_L(\phi[m \mapsto \texttt{Lb}\ l\ e]) \\
&= \varepsilon_L(\phi[m \mapsto \texttt{Lb}\ l\ \bullet]) \\
&= \varepsilon_L(\phi)
\end{aligned}
$$

- **Case** $h_e(\Sigma) = \Sigma[\alpha_l \mapsto g_l^e(\Sigma.\alpha_l)]$ **with** $g_l^e(\alpha) = \alpha \triangleright e$. Analogous.

Lemma 7. *If* $\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle \hookrightarrow \langle \Sigma^1, t_s'\rangle$ *with* $\sigma.\texttt{lbl} \not\sqsubseteq L$, *then* $\varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle) = \varepsilon_L(\langle \Sigma^1, t_s'\rangle)$, *i.e.,*

$$
\begin{array}{ccc}
\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle & \hookrightarrow & \langle \Sigma^1, t_s'\rangle \\
\Big\downarrow{\scriptstyle \varepsilon_L} & & \Big\downarrow{\scriptstyle \varepsilon_L} \\
\varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle) & = & \varepsilon_L(\langle \Sigma^1, t_s'\rangle)
\end{array}
$$

Proof. We illustrate the proof in the case of rule (Step). Let $\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle \longrightarrow \langle \Sigma^1, t_s \triangleright \langle \sigma, e'\rangle\rangle$, then

$$
\begin{aligned}
&\varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle) \\
&= \langle \varepsilon_L(\Sigma), \varepsilon_L(t_s)\rangle \\
&= \langle \varepsilon_L(\Sigma^1), \varepsilon_L(t_s)\rangle \\
&= \varepsilon_L(\langle \Sigma^1, t_s \triangleright \langle \sigma, e'\rangle\rangle)
\end{aligned}
$$

The other cases are similar.

The next lemma establishes a simulation between $\hookrightarrow^*$ and $\hookrightarrow_L^*$.

Lemma 1 (Many-step simulation). *Given a well-typed thread pool* t_s *(with no* `Lb`, $()^{L\mathit{IO}}$, $\boxdot$, R, *and* $\bullet$*), an attacker at level* L, *and a runtime environment* Σ, *if* $\langle \Sigma, t_s\rangle \hookrightarrow^* \langle \Sigma', t_s'\rangle$, *then* $\varepsilon_L(\langle \Sigma, t_s\rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma', t_s'\rangle)$.

Proof. In order to prove this result, we rely on properties of the erasure function, such as the fact that it is idempotent and homomorphic to the application of evaluation contexts and substitution.

The proof is by induction on the derivation of $\langle \Sigma, t_s\rangle \hookrightarrow^* \langle \Sigma', t_s'\rangle$. We consider a thread queue of the form $\langle \sigma, e\rangle \triangleleft r_s$, and suppose that $\langle \Sigma, \langle \sigma, e\rangle \triangleleft r_s\rangle \hookrightarrow \langle \Sigma^1, r_s'\rangle$ and $\langle \Sigma^1, r_s'\rangle \hookrightarrow^* \langle \Sigma', t_s'\rangle$ (otherwise the reduction is not making any progress, and the result is trivial).

- If $\sigma.\texttt{lbl} \sqsubseteq L$, the result follows by Lemma 5 and the induction hypothesis.
- If $\sigma.\texttt{lbl} \not\sqsubseteq L$, the result follows by Lemma 7 and the induction hypothesis.

We can now prove the non-interference theorem.

Theorem 2 (Termination-sensitive non-interference). *Given a computation* e *(with no* `Lb`*,* $()^{LIO}$*,* $\boxdot$*,* `R`*, and* $\bullet$*) where* $\Gamma \vdash e : \texttt{Labeled}\,\ell\,\tau \to \texttt{LIO}\,\ell\,(\texttt{Labeled}\,\ell\,\tau')$*, an attacker at level* L*, an initial securiy context* σ*, and runtime environments* Σ_1 *and* Σ_2 *where* $\Sigma_1.\phi = \Sigma_2.\phi = \varnothing$ *and* $\Sigma_1.\alpha_k = \Sigma_2.\alpha_k = \epsilon$ *for all levels* k*, then*

$$\begin{aligned}&\forall e_1 e_2.(\Gamma \vdash e_i : \texttt{Labeled}\,\ell\,\tau)_{i=1,2} \wedge\ e_1 \approx_L e_2\\ &\qquad \wedge\ \langle \Sigma_1, \langle \sigma, e\ e_1\rangle\rangle \hookrightarrow^* \langle \Sigma_1', t_s^1\rangle\\ &\Rightarrow \exists \Sigma_2' t_s^2.\langle \Sigma_2, \langle \sigma, e\ e_2\rangle\rangle \hookrightarrow^* \langle \Sigma_2', t_s^2\rangle \wedge\ \langle \Sigma_1', t_s^1\rangle \approx_L \langle \Sigma_2', t_s^2\rangle\end{aligned}$$

Proof. Take $\langle \Sigma_1, \langle \sigma, e\ e_1\rangle\rangle \hookrightarrow^* \langle \Sigma_1', t_s^1\rangle$ and apply Lemma 1 to get $\varepsilon_L(\langle \Sigma_1, \langle \sigma, e\ e_1\rangle\rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_1', t_s^1\rangle)$. We know this reduction only includes public ($\sqsubseteq L$) steps, so the number of steps is lower than or equal to the number of steps in the first reduction.

We can always find a reduction starting from $\varepsilon_L(\langle \Sigma_2, \langle \sigma, e\ e_2\rangle\rangle)$ with the same number of steps as $\varepsilon_L(\langle \Sigma_1, \langle \sigma, e\ e_1\rangle\rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_1', t_s^1\rangle)$, so by the Determinacy Lemma we have $\varepsilon_L(\langle \Sigma_2, \langle \sigma, e\ e_2\rangle\rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_2', t_s^2\rangle)$. By Lemma 1 again, we get $\langle \Sigma_2, \langle \sigma, e\ e_2\rangle\rangle \hookrightarrow^* \langle \Sigma_2', t_s^2\rangle$ and therefore $\langle \Sigma_1', t_s^1\rangle \approx_L \langle \Sigma_2', t_s^2\rangle$.

B Semantics and typing rules

Listings 9 and 10 show the missing typing rules for the calculus. Similarly, Listing 11 shows the reduction rules that were not included in Section 2.6.

Listing 9 Typing rules for values.

$$\frac{}{\vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{Bool}} \qquad \frac{}{\vdash () : ()} \qquad \frac{}{\vdash l : \ell} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \texttt{fix}\ e : \tau}$$

C Application: Mitigating attack on RSA

As in [Askarov et al., 2010], to highlight the effectiveness of our mitigator implementation, we re-implement the timing attack on the OpenSSL 0.9.7 RSA implementation as originally presented in [Brumley and Boneh, 2003]. Compared to the previous dating-website scenario, in which a malicious app deliberately delayed computations, the covert timing channel in this

case is present due to the non-trivial operations performed in a decryption. Hence, an attacker can recover an RSA key by repeatedly requesting the RSA oracle, which may be a web server using SSL, to decrypt different ciphertext messages.

Following [Brumley and Boneh, 2003], one can reveal the secret key indirectly, by recovering q and exposing the factorization of RSA modulus $N = pq$, for $q < p$. To do so, the attack proceeds as follows. Firstly, it guesses an initial value for q, named g, that is between $2^{\log_2 N/2}$ and $2^{\log_2 N/2-1}$, and plots the decryption times (in nanoseconds) of all the most significant 2-3 bits. The expected peak in the plot graph corresponds to our first approximation of q. Assuming that the most significant $i-1$ bits of q have been already recovered, we recover the ith bit according to:

- Set the $i-1$ most significant bits (MSB) of g_i to the $i-1$ recovered MSB of q, leaving the remaining bits unset.
- Let g_{hi} being the same as g_i but with the ith bit set.
- Measure the time to decrypt g_i, written t_1.
- Measure the time to decrypt g_{hi}, written t_2.
- Compute $\Delta = \mid t_2 - t_1 \mid$. If Δ is large, bit i of q is unset, otherwise it is set.

As in [Brumley and Boneh, 2003, Askarov et al., 2010], we overcome noise due to the operating system being a multi-user environment by repeating the decryption for g_i and g_{hi} numerous times (in our experiments, 7) and taking the median time difference. Additionally, to build a strong indicator for the bits of q, we take the time difference of decrypting a neighborhood of values $g_i, \ldots, g_i + n$ and the corresponding neighborhood of high values $g_{hi}, \ldots, g_{hi} + n$; in our experiments $n = 600$.

To evaluate our Haskell mitigator implementation with the RSA attack, we extended the HsOpenSSL package with bindings for the C OpenSSL RSA encryption and decryption functions. On a laptop with a Intel Core i7 2620M (2.7GHz) processor with 8GB of RAM, we built our extended Haskell OpenSSL library with GHC 7.2.1, linking it against the C OpenSSL 0.9.7 library. The attack against a "toy" 512-bit key is shown Figure 1. We only carried out the attack against the 256 MSBs as Coppersmith's algorithm can be used to recover the rest in an efficient manner [Coppersmith, 1997]. As the figure shows, there is a clear distinction between when the bits of q are 0 and 1. Finally, applying the fast-doubling time mitigator with an initial quantum of 500 microseconds, we bound the key leakage as shown by the results of Figure 2.

D Evaluation: Overhead of a fork

To analyze the performance penalty in using `forkLIO` and `waitLIO` as opposed to `toLabeled` we micro-benchmarked the two approaches. As expected, Figure 3, the performance overhead of forking is unnoticeable.

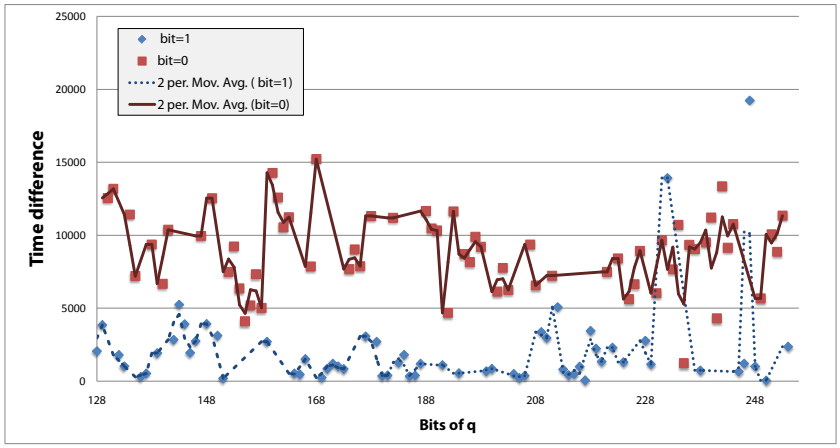

Fig. 1. Unmitigated RSA attack. Time difference is in nanoseconds.

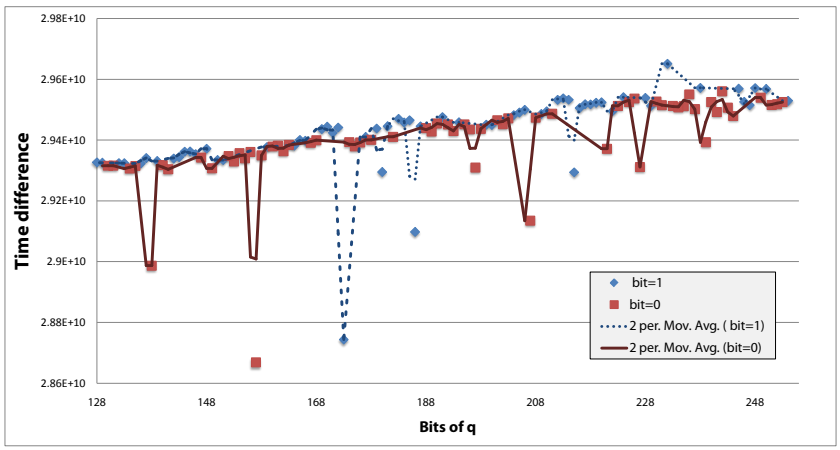

Fig. 2. Mitigated RSA attack. Time difference is in nanoseconds.

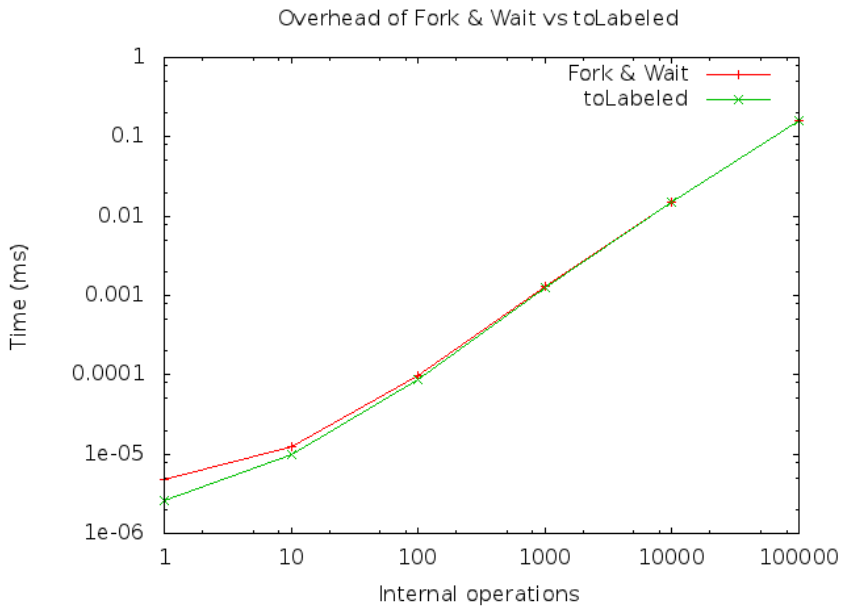

Fig. 3. Execution time in milliseconds for performing a `forkLIO` and `waitLIO` or `toLabeled`. The x-axis specifies the number of operations performed inside the `forkLIO` or `toLabeled`.

Listing 10 Typing rules for expressions.

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e : \texttt{LIO}\ \ell\ \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{LIO}\ \ell\ \tau_1 \qquad \Gamma \vdash e_2 : \tau_1 \to \texttt{LIO}\ \ell\ \tau_2}{\Gamma \vdash e_1\ \texttt{>>=}\ e_2 : \texttt{LIO}\ \ell\ \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{label}\ e_1\ e_2 : \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau)} \qquad \frac{\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau}{\Gamma \vdash \texttt{unlabel}\ e : \texttt{LIO}\ \ell\ \tau}$$

$$\frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \texttt{LIO}\ \ell\ \tau}{\Gamma \vdash \texttt{forkLIO}\ e_1\ e_2 : \texttt{LIO}\ \ell\ (\texttt{Result}\ \ell\ \tau)} \qquad \frac{\Gamma \vdash e : \texttt{Result}\ \ell\ \tau}{\Gamma \vdash \texttt{waitLIO}\ e : \texttt{LIO}\ \ell\ \tau}$$

$$\frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{out}\ e_1\ e_2 : \texttt{LIO}\ \ell\ ()} \qquad \frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{newLMVar}\ e_1\ e_2 : \texttt{LIO}\ \ell\ (\texttt{LMVar}\ \ell\ \tau)}$$

$$\frac{\Gamma \vdash e : \texttt{LMVar}\ \ell\ \tau}{\Gamma \vdash \texttt{takeLMVar}\ e : \texttt{LIO}\ \ell\ \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{LMVar}\ \ell\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{putLMVar}\ e_1\ e_2 : \texttt{LIO}\ \ell\ ()}$$

$$\frac{\Gamma \vdash e : \ell}{\vdash \texttt{lowerClr}\ e : \texttt{LIO}\ \ell\ ()} \qquad \vdash \texttt{getLabel} : \texttt{LIO}\ \ell\ \ell$$

$$\vdash \texttt{getClearance} : \texttt{LIO}\ \ell\ \ell \qquad \frac{\Gamma \vdash e : \texttt{Lb}\ \ell\ \tau}{\Gamma \vdash \texttt{labelOf}\ e : \ell}$$

$$\frac{\Gamma \vdash m : \texttt{LMVar}\ \ell\ \tau}{\Gamma \vdash \texttt{labelOfLMVar}\ m : \ell}$$

Listing 11 Semantics for standard constructs.

$$
\begin{aligned}
E \quad ::= \; & [\cdot] \mid E\ e \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid \texttt{return}\ E \mid E\ \texttt{>>=}\ e \\
& \mid \texttt{lowerClr}\ E \mid \texttt{labelOf}\ E \mid \ldots
\end{aligned}
$$

$$\langle \Sigma, \langle \sigma, E[(\lambda x.e_1)\ e_2] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[[e_2/x]e_1] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{fix}\ e] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[e\ (\texttt{fix}\ e)] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{if true then}\ e_1\ \texttt{else}\ e_2] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[e_1] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{if false then}\ e_1\ \texttt{else}\ e_2] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[e_2] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{let}\ x = e_1\ \texttt{in}\ e_2] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[[e_1/x]e_2] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{return}\ v] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[(v)^{\texttt{LIO}}] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[(v)^{\texttt{LIO}}\ \texttt{>>=}\ e_2] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[e_2\ v] \rangle \rangle$$

$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}}{\langle \Sigma, \langle \sigma, E[\texttt{label}\ l\ e] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)] \rangle \rangle}$$

$$\frac{l' = \sigma.\texttt{lbl} \sqcup l \qquad l' \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{lbl} \mapsto l']}{\langle \Sigma, \langle \sigma, E[\texttt{unlabel}\ (\texttt{Lb}\ l\ e)] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma', E[\texttt{return}\ e] \rangle \rangle}$$

$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{clr} \mapsto l]}{\langle \Sigma, \langle \sigma, E[\texttt{lowerClr}\ l] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma', E[\texttt{return}\ ()] \rangle \rangle}$$

$$\langle \Sigma, \langle \sigma, E[\texttt{getLabel}] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{return}\ \sigma.\texttt{lbl}] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{getClearance}] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{return}\ \sigma.\texttt{clr}] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{labelOf}\ (\texttt{Lb}\ l\ e)] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[l] \rangle \rangle$$

CHAPTER
THREE

ELIMINATING CACHE-BASED TIMING ATTACKS WITH INSTRUCTION-BASED SCHEDULING

Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy
David Terei, Alejandro Russo, David Mazières

Abstract. Information flow control allows untrusted code to access sensitive and trustworthy information without leaking this information. However, the presence of covert channels subverts this security mechanism, allowing processes to communicate information in violation of IFC policies. In this paper, we show that concurrent deterministic IFC systems that use time-based scheduling are vulnerable to a cache-based internal timing channel. We demonstrate this vulnerability with a concrete attack on Hails, one particular IFC web framework. To eliminate this internal timing channel, we implement instruction-based scheduling, a new kind of scheduler that is indifferent to timing perturbations from underlying hardware components, such as the cache, TLB, and CPU buses. We show this scheduler is secure against cache-based internal timing attacks for applications using a single CPU. To show the feasibility of instruction-based scheduling, we have implemented a version of Hails that uses the CPU retired-instruction counters available on commodity Intel and AMD hardware. We show that instruction-based scheduling does not impose significant performance penalties. Additionally, we formally prove that our modifications to Hails' underlying IFC system preserve non-interference in the presence of caches.

3.1 Introduction

The rise of extensible web applications, like the Facebook Platform, is spurring interest in information flow control (IFC) [Myers and Liskov, 1997, Sabelfeld and Myers, 2003]. Popular platforms like Facebook give approved apps full access to users' sensitive data, including the ability to violate security policies set by users. In contrast, IFC allows websites to run untrusted, third-party apps that operate on sensitive user data [Krohn et al., 2007a, Giffin et al., 2012], ensuring they abide by security policies in a mandatory fashion.

Recently, Hails [Giffin et al., 2012], a web-platform framework built atop the LIO IFC system [Stefan et al., 2011, 2012], has been used to implement websites that integrate third-party untrusted apps. For example, the code-hosting website GitStar.com built with Hails uses untrusted apps to deliver core features, including a code viewer and wiki. GitStar relies on LIO's IFC mechanism to enforce robust privacy policies on user data and code.

LIO, like other IFC systems, ensures that untrusted code does not write data that may have been influenced by sensitive sources to public sinks. For example, an untrusted address-book app is allowed to compute over Alice's friends list and display a stylized version of the list to Alice, but it cannot leak any information about her friends to arbitrary end-points. The flexibility of IFC makes it particularly suitable for the web, where access control lists often prove either too permissive or too restrictive.

However, a key limitation of IFC is the presence of *covert channels*, i.e., "channels" not intended for communication that nevertheless allow code to subvert security policies and share information [Lampson, 1973]. A great deal of research has identified and analyzed covert channels [Millen, 1999]. In this work, we focus on the *internal timing covert channel*, which occurs when sensitive data is used to manipulate the timing behavior of threads so that other threads can observe the order in which shared public resources are used [Smith and Volpano, 1998, Volpano and Smith, 1999]. Though we do not believe our solution to the internal timing covert channel affects (either positively or negatively) other timing channels, such as the external timing covert channel, which is derived from measuring external events [Agat, 2000, Hedin and Sands, 2005, Barthe et al., 2006] (e.g., wall-clock), addressing these channels is beyond our present scope.

LIO eliminates the internal timing covert channel by restricting how programmers write code. Programmers are required to explicitly decouple computations that manipulate sensitive data from those that can write to public resources, eliminating covert channels *by construction*. However, decoupling only works when *all* shared resources are modeled. LIO only considers shared resources that are expressible by the programming language, e.g., shared-variables, file descriptors, semaphores, channels, etc. Implicit operating system and hardware state can still be exploited to alter

the timing behavior of threads, and thus leak information. Reexamining LIO, we found that the underlying CPU cache can be used to introduce an internal timing covert channel that leaks sensitive data. A trivial attack can leak data at 0.75 bits/s and, despite the low bandwidth, we were able to leak all the collaborators on a private GitStar.com project in less than a minute.

Several countermeasures to cache-based attacks have previously been considered, primarily in the context of cryptosystems following the work of Kocher [Kocher, 1996] (see Section 3.8). Unfortunately, many of the techniques are not designed for IFC scenarios. For example, modifying an algorithm implementation, as in the case of AES [Bonneau and Mironov, 2006], does not naturally generalize to arbitrary untrusted code. Similarly, flushing or disabling the cache when switching protection domains, as suggested in [Barthe et al., 2012, Zhang et al., 2012], is prohibitively expensive in systems like Hails, where context switches occur hundreds of times per second. Finally, relying on specialized hardware, such as partitioned caches [Page, 2005], which isolate the effects of one partition from code using a different partition, restricts the deployability and scalability of the solution; partitioned caches are not readily available and often cannot be partitioned to an arbitrary security lattice.

This paper describes a countermeasure for cache-based attacks when execution is confined to a single CPU. Our method generalizes to arbitrary code, imposes minimal performance overhead, scales to an arbitrary security lattice, and leverages hardware features already present in modern CPUs. Specifically, we present an instruction-based scheduler that eliminates internal timing channels in concurrent programs that time-slice a single CPU and contend for the same cache, TLB, bus, and other hardware facilities. We implement the scheduler for the LIO IFC system and demonstrate that, under realistic restrictions, our scheduler eliminates such attacks in Hails web applications.

Our contributions are as follows.

- We implement a cache-based internal timing attack for LIO.
- We close the cache-based covert channel by scheduling user-level threads on a single CPU core based on the number of instructions they execute (as opposed to the amount of time they execute). Our scheduler can be used to implement other concurrent IFC systems which implicitly assume instruction-level scheduling (e.g., [Smith and Volpano, 1998, Honda et al., 2000, Zdancewic and Myers, 2003, Huisman et al., 2006, Russo and Sabelfeld, 2006a]).
- We implement our instruction-based scheduler as part of the Glasgow Haskell Compiler (GHC) runtime system, atop which LIO and Hails are built. We use CPU performance counters, prevalent on most modern CPUs, to pre-empt threads according to the number of retired in-

structions. The measured impact on performance, when compared to time-based scheduling, is negligible.
We believe these techniques to be applicable to operating systems that enforce IFC, including [Zeldovich et al., 2006, Krohn et al., 2007b, Murray et al., 2013], though at a higher cost in performance for application code that is highly optimized for locality (see Section 3.5).

- We augment the LIO [Stefan et al., 2012] semantics to model the cache and formally prove that instruction-based scheduling removes leaks due to caches.

The paper is organized as follows. Section 3.2 discusses cache-based attacks and existing countermeasures. In Section 3.3 presents our instruction-based scheduling solution. Section 3.4 describes our modifications to GHC's runtime, while Section 3.5 analyses their performance impact. Formal guarantees and discussions of our approach are detailed in Sections 3.6 and 3.7. We describe related work in Section 3.8 and conclude in Section 3.9.

3.2 Cache Attacks and Countermeasures

The severity of information leakage attacks through the CPU hardware cache has been widely considered by the cryptographic community (e.g. [Percival, 2005, Osvik et al., 2006]). Unlike crypto work, where attackers extract sensitive information through the execution of a fixed crypto algorithm, we consider a scenario in which the attacker provides arbitrary code in a concurrent IFC system. In our scenario, the adversary is a developer that implements a Hails app that interfaces with user-sensitive data using LIO libraries.

We found that, knowing only the cache size of the underlying CPU, we can easily build an app that exploits the shared cache to carry out an internal timing attack that leaks sensitive data at 0.75 bits/s. Several IFC systems, including [Honda et al., 2000, Smith and Volpano, 1998, Zdancewic and Myers, 2003, Huisman et al., 2006, Russo and Sabelfeld, 2006a, Stefan et al., 2012], model internal timing attacks and address them by ensuring that the outcome of a race to a public resource does not depend on secret data. Unfortunately, these systems only account for resources explicitly modeled at the programming language level and not underlying OS or hardware state, such as the CPU cache or TLB. Hence, even though the semantics of these systems rely on instruction-based scheduling (usually to simplify expressing reduction rules), real-world implementations use time-based scheduling for which the formal guarantees do not hold. The instruction-based scheduler proposed in this work can be used to make the assumptions of such concurrent IFC systems match the situation in practice. In the remainder of this section, we show the internal timing attack that leverages the hardware cache. We also discuss several existing countermeasures that could be employed by Hails.

```
1.                                  lowArray := new Array[M];
2.                                  fillArray (lowArray)
```

thread 1	thread 2	thread 3
1. if secret	1. for i in [1.. n]	1. for i in [1.. n+m]
2. then highArray := new Array[M]	2. skip	2. skip
3. fillArray (highArray)	3. readArray(lowArray)	3. outputLow(0)
4. else skip	4. outputLow(1)	

Fig. 1. A simple cache attack.

3.2.1 Example cache attack

We mount an internal timing attack by influencing the scheduling behavior of threads through the cache. Consider the code shown in Figure 1. The attack leaks the secret boolean value secret in thread 1 by affecting when thread 2 writes to the public channel relative to thread 3.

The program starts (lines 1–2) by creating and initializing a public array lowArray whose size M corresponds to the cache size; fillArray simply sets every element of the array to 0 (this will place the array in the cache). The program then spawns three threads that run concurrently. Assuming a round-robin time-based scheduler, the execution of the attack proceeds as illustrated in Figure 2, where secret is set to true (top) and false (bottom), respectively.

- Depending on the secret value secret, thread 1 either performs a no-operation (skip on line 4), leaving the cache intact, or evicts lowArray from the cache (lines 2–3) by creating and initializing a new (non-public) array highArray.
- We assume that thread 1 takes less than n steps to complete its execution—a number that can be determined experimentally; in Figure 2, n is four. Hence, to allow all the effects on the cache due to thread 1 to settle, thread 2 delays its computation by n steps (lines 1–2). Subsequently, the thread reads every element of the public array lowArray (line 3), and finally writes 1 to a public output channel (line 4). Crucial to carrying out the attack, the duration of thread 2's reads (line 3) depends on the state of the cache: if the cache was modified by thread 1, i.e., secret is true, thread 2 needs to wait for all the public data to be retrieved from memory (as opposed to the cache) before producing an output. This requires evicting highArray from the cache and fetching lowArray, a process that takes a non-negligible amount of time. However, if the cache was not touched by thread 1, i.e., secret is false, thread 2 will get few cache misses and thus produce its output with no delay.
- We assume that thread 2 takes less than m, where m<n, steps to complete reading lowArray (line 3) when the reads hit the cache, i.e., lowArray was not replaced by highArray. Like n, this metric can be determined experimentally; in Figure 2, m is three. Using this, thread 3 simply delays its computation by n+m steps (lines 1–2) and then writes 0 to

a public output channel (line 3). The role of thread 3 is solely to serve as a baseline for thread 2's output: producing its output before thread 2 when the latter is filling the cache, i.e., secret is true; conversely, it produces an output after thread 2 if thread 1 did not touch the cache, i.e., secret is false.

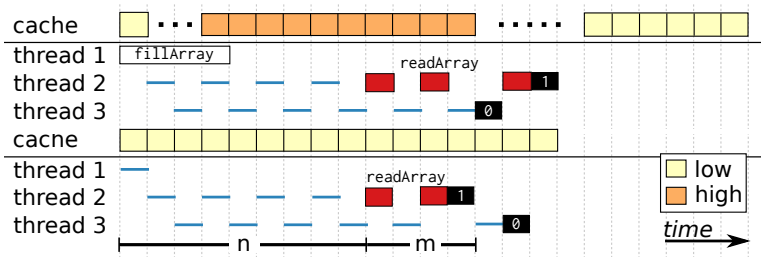

Fig. 2. Execution of the cache attack with secret true (top) and false (bottom).

We remark that the race between thread 2 and thread 3 to write to a shared public channel, influenced by the cache state, is precisely what facilitates the attack. We described how to leak a single bit, but the attack can easily be magnified by wrapping it in a loop. Note also that we have assumed the attacker has complete control of the cache—i.e., the cache is not affected by other code running in parallel. However, the attack is still plausible under weaker assumptions so long as the attacker deals with the additional noise, as exemplified by the timing attacks on AES [Osvik et al., 2006].

3.2.2 Existing countermeasures

The internal timing attack arises as a result of cache effects influencing thread-scheduling behavior. Hence, one series of countermeasures addresses the problem through low-level CPU features that provide better control of the cache.

Flushing the cache Naively, we can flush the cache on every context switch. In the context of Figure 1, this guarantees that, when thread 2 executes the readArray instruction, its duration is not affected by thread 1 evicting lowArray from the cache—the cache will *always* be flushed on a context switch, hence thread 3 will always write to the output channel first.

No-fill cache mode Several architectures, including Intel's Xeon and Pentium 4, support a cache *no-fill* mode [Intel, 2012]. In this mode, read/write

hits access the cache; misses, however, read from and write to memory directly, leaving the cache unchanged. As considered by Zhang et al. [Zhang et al., 2012], we can execute all threads that operate on non-public data in this mode. This approach guarantees that sensitive data cannot affect the cache. Unfortunately, threads operating on non-public data and relying on the cache will suffer from performance degradation.

Partitioned cache Another approach is to partition the cache according to the number of security levels, as suggested in [Zhang et al., 2012]. Using this architecture, a thread computing on secret data only accesses the secret partition, while a thread computing on public data only access the public one. This approach effectively corresponds to giving each differently-labeled thread access to its own cache and, as a result, the scheduling behavior of public threads cannot be affected by evicting data from the cache.

Unfortunately, none of the aforementioned solutions can be used in systems built with Hails (e.g., GitStar). Flushing the cache is prohibitively expensive for preemptive systems that perform a context switch hundreds of times per second—the impact on performance would gravely reduce usability. The no-fill mode solution is well suited for systems wherein the majority of the threads operate on public data. In such cases, only threads operating on sensitive data will incur a performance penalty. However, in the context of Hails, the solution is only slightly less expensive than flushing the cache. Hails threads handle HTTP requests that operate on individual (non-public) user data, hence most threads will not be using the cache. Another consequence of threads handling differently-labeled data is that partitioned caches can only be used in a limited way (see Section 3.8). Specifically, to address internal timing attacks, it is required that we partition the cache according to the number of security levels in the lattice. Given that most existing approaches can only partition caches up to 16-ways at the OS level [Lin et al., 2008], and fewer at the hardware level, an alternative scalable approach is necessary. Moreover, neither flushing nor partitioning the cache can handle timing perturbations arising from other pieces of hardware such as the TLB, buses, etc.

3.3 Instruction-based Scheduling

As the example in Figure 2 shows, races to acquire public resources are affected by the cache state, which in turn might be affected by secret values. It is important to highlight that the number of instructions executed in a given quantum of time might vary depending on the state of the cache. It is precisely this variability that reintroduces dangerous races into systems. However, the actual set of instructions executed is not affected by the cache. Hence, we propose scheduling threads according to the number of instructions they execute, rather than the amount of time they consume.

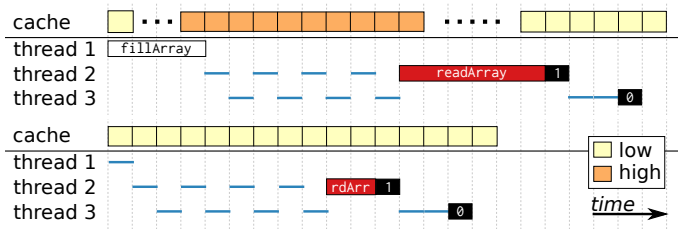

Fig. 3. Execution of cache attack program of Figure 1 with secret set to true (top) and false (bottom). In both executions, we highlight that the threads execute one "instruction" at a time in a round-robin fashion. The concurrent threads take the same amount of time to complete execution as in Figure 2. However, since we use instructions to context switch threads, the interleaving between thread 2 or 3 is not influenced by the actions in thread 1, and thus the internal timing attack does not arise—the threads' output order cannot encode sensitive data.

The point at which a thread produces an output (or any other visible operation) is determined according to the number of instructions it has executed, a measurement unaffected by the amount of time it takes to perform a read/write from memory.

Consider the code in Figure 1 executing atop an instruction-based scheduler. An illustration of this is shown in Figure 3. For simplicity of exposition, the instruction granularity is at the level of commands (skip, readArray, etc.) and therefore context switches are triggered after one command gets executed. (In Section 3.4, we describe a more practical and realistic instruction-based scheduler.) Observe that the amount of time it takes to execute an instruction has not changed from the time-based scheduler of Figure 2. For example, readArray still takes 6 units of time when secret is true, and 2 when it is false. Unlike Figure 2, however, the interleaving between thread 2 and thread 3 did not change depending on the state of the cache (which did change according to secret). Therefore, a race to write to the public channel between thread 2 and thread 3 cannot be caused by the secret, through the cache. The second thread always executes $n+1 = 5$ instructions before writing 1 to the public channel, while the third thread always executes $n+m+1 = 8$ instructions before writing 0.

Our proposed countermeasure, the implementation of which is detailed in Section 3.4, eliminates the cache-based internal timing attacks without sacrificing scalability and with a minor performance impact. With instruction-based scheduling, we do not require flushing of the cache. In this manner, applications can safely utilize the cache to retain most of their performance without giving up system security, and unlike current partitioned caches, we can scale up to consider arbitrarily complex lattices.

3.4 Implementation

We implemented an instruction-based scheduler for LIO. In this section, we describe this implementation and detail some key design features we believe to be useful when modifying concurrent IFC systems to address cache-based timing attacks.

3.4.1 LIO and Haskell

LIO is a Haskell library that exposes concurrency to programmers in the form of "green," lightweight threads. Each LIO thread is a *native* Haskell thread that has an associated security level (label) which is used to track and control the flow of information to/from the thread. LIO relies on Haskell libraries for creating new threads and the runtime system for managing them.

In general, M lightweight Haskell threads may concurrently execute on N OS threads. (It is common, however, for multiple Haskell threads to execute on a single OS thread, i.e., a many-to-one mapping.) The Haskell runtime, as implemented by the GHC system, uses a round-robin scheduler to context switch between concurrently executing threads. Specifically, the scheduler is invoked whenever a thread blocks/terminates or a timer signal alarm is received. The timer is used to guarantee that the scheduler is periodically executed, allowing the runtime to implement preemptive scheduling.

3.4.2 Instruction-based scheduler

As previously mentioned, timing-based schedulers render systems, such as LIO, vulnerable to cache-based internal timing attacks. We implement our instruction-based scheduler as a drop-in replacement for the existing GHC scheduler, using the number of retired instructions to trigger a context switch.

Specifically, we use performance monitoring units (PMUs) present in almost all recent Intel [Intel, 2012] and AMD [AMD, 2008] CPUs. PMUs expose hardware performance counters that are typically used by developers to optimize code—they provide metrics such as the number of cache misses, instructions executed per cycle, branch mispredictions, etc. Importantly, PMUs also provide a means for counting the number of retired instructions.

Using the perfmon2 [Eranian, 2006] Linux monitoring interface and helper user-level library libpfm4, we modified the GHC runtime to configure the underlying PMU to count the number of retired instructions the Haskell process is executing. Specifically, with perfmon2 we set a data performance counter register to $2^{64} - n$, which the CPU increments upon retiring an

instruction.[1] Once the counter overflows, i.e., n instructions have been retired, perfmon2 is sent a hardware interrupt. In our implementation, we configured perfmon2 to handle the interrupt by delivering a signal to the GHC runtime.

If threads share no resources, upon receiving a signal, the executing Haskell thread can immediately save its state and jump to the scheduler. However, preempting a thread which is operating on a shared memory space can be dangerous, as the thread may have left memory in an inconsistent state. (This is the case for many language runtimes, not solely GHC's.) To avoid this, GHC produces code that contains *safe points* where threads may yield. Hence, a signal does not cause an immediate preemption. Instead, the signal handler simply sets a flag indicating the arrival of a signal; at the next safe point, the thread "cooperatively" yields to the scheduler.

To ensure liveness, we must guarantee that given any point in execution, a safe point is reached in n instructions. Though GHC already inserts many safe points as a means of invoking the garbage collector (via the scheduler), tight loops that do not perform any allocation are known to hang execution [GHC, 2005]. Addressing this eight-year old bug, which would otherwise be a security concern in LIO, we modified the compiler to insert safe points on function entry points. This modification, integrated in the mainline GHC, has almost no effect on performance and only a 7% bloat in average binary size.

3.4.3 Handling IO

Threads yield at safe points in their execution paths as a result of a retired instruction signal. However, there are circumstances in which threads would like to explicitly yield prior to the reception of a retired instruction signal. In particular, when a thread performs a blocking operation, it immediately yields to the scheduler, registering itself to wake up when the operation completes. Thus, any IO action is a yield which allows the thread to give up the rest of its scheduling quantum.

While yields are not intrinsically unsafe, it is not safe to allow the leftover scheduling quantum to be passed on to the next thread. Thus, after running any asynchronous IO action, the runtime must reset the retired instruction counter. Hence, whenever a thread enters the scheduler loop due to being blocked, we reset the retired instruction counter.

3.5 Performance Evaluation

We evaluated the performance of instruction-based scheduling against existing time-based approaches using the nofib benchmark suite [Partain,

[1] Though the bit-width of the hardware counters vary (they are typically 40-bits wide) perfmon2 internally manages a 64-bit counter.

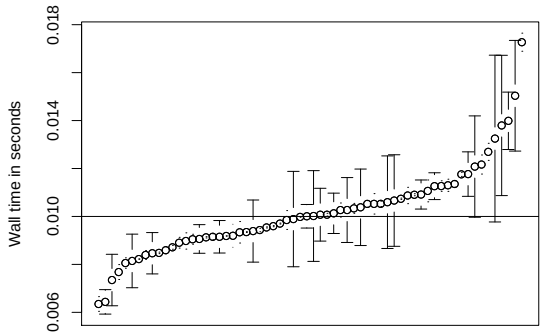

Fig. 4. Mean time between timer signal and retired-instruction signal. Each point represents a program from nofib, which have been sorted on the x-axis by their mean time.

1992]. nofib is the standard benchmarking suite used for measuring the performance of Haskell implementations.

In our experimental setup, we used the latest development version of GHC (the Git master branch as of November 6, 2012). The measurements were taken on the same hardware as Hails [Giffin et al., 2012]: a machine with two dual-core Intel Xeon E5620 (2.4GHz) processors, and 48GB of RAM.

We first needed to find an instruction budget—number of instructions to retire before triggering the scheduler. We found a poorly chosen instruction budget could increase runtime by 100%. To determine a good parameter, we measured the mean time between retired-instruction signals with an initially guessed instruction budget parameter. We then adjusted the parameter so the median test program had a 10 millisecond mean time-slice (the default quantum size in vanilla GHC with time-based scheduling) and verified our final choice by re-running the measurements. For our specific setup, an instruction budget of approximately 37,100,000 retired-instructions corresponded to a 10 millisecond time quantum. We plot the mean and standard deviation across all nofib applications with the final tuning parameter in Figure 4. We found that most programs receive a signal within 2 milliseconds of when they would have normally received the signal using the standard time-based scheduler. While the instruction budget parameter will vary across machines, it is relatively simple to bootstrap this parameter by performing these measurements at startup and tuning the budget accordingly.

Next, we compared the performance of Haskell's timer-based scheduler with our instruction-based scheduler. We used a subset of the nofib benchmark suite called the real benchmark, which consists of "real world programs", as opposed to synthetic benchmarks (however, results for the whole nofib suite are comparable). Figure 5 shows the run time of these

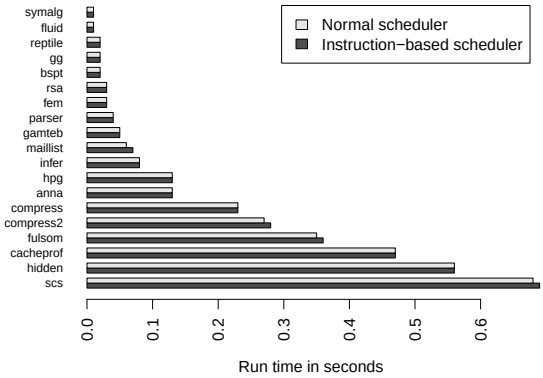

Fig. 5. Change to run time from instruction-based scheduling

programs with both scheduling approaches. With an optimized instruction budget parameter, instruction-based scheduling has no impact to the runtime of the majority of nofib applications and results in only a very slight increase in runtime for others (about 1%).

This result may seem surprising: instruction-based scheduling purposely punishes threads with good data locality, so one might expect a more substantial performance impact. We hypothesize that this is the case due to two reasons. First, with preemptive scheduling, we are already inducing cache misses when we switch from running one thread to another—instruction-based scheduling only perturbs when these preempts occur, and as seen in Figure 4, these perturbations are very minor. Second, modern L2 caches are quite large, meaning that hardware is more forgiving of poor data locality—an effect that has been measured in the behavior of stock lazy functional programs [Ahmad and DeYoung, 2009].

3.6 Cache-aware semantics

In this section we recall relevant design aspects of LIO [Stefan et al., 2012] and extend the original formalization to consider how caches affect the timing behavior of programs. Importantly, we formalize instruction-based scheduling and show how it removes cache-based internal timing covert channels.

3.6.1 LIO Overview

At a high level, LIO provides the LIO monad, which is used in place of IO. Wrapping standard Haskell libraries, LIO exports a collection of functions that untrusted code may use to access the filesystem, network, shared

variables, etc. Unlike the standard libraries, which usually return IO actions, these functions return actions in the LIO monad, thus allowing LIO to perform label checks before executing a potentially unsafe action.

Internally, the LIO monad keeps track of a *current label*, L_{cur}. The current label is effectively a ceiling over the labels of all data that the current computation may depend on. This label eliminates the need to label individual definitions and bindings: symbols in scope are (conceptually) labeled with L_{cur}.[2] Hence, when a computation C, with current label L_C, observes an object labeled L_O, C's label is raised to the least upper bound or *join* of the two labels, written $L_C \sqcup L_O$. Importantly the current label governs where the current computation can write, what labels may be used when creating new channels or threads, etc. For example, after reading O, the computation should not be able to write to a channel K if L_C is more restricting than L_K—this would potentially leak sensitive information (about O) into a less sensitive channel.

Note that an LIO computation can only execute a sub-computation on sensitive data by either raising its current label or forking a new thread in which to execute this sub-computation. In the former case, raising the current label prevents writing to less sensitive endpoints. In the latter case, to observe the result (or timing and termination behavior) of the sub-computation the thread must wait for the forked thread to finish, which first raises the current label. A consequence of this design is that differently-labeled computations are decoupled, which, as mentioned in Section 3.1, is key to eliminating the internal timing covert channel.

In the next subsection, we will outline the semantics for a cache-aware, time-based scheduler where the cache attack described in Section 3.2 is possible. Moreover, we show that we can easily adapt this semantics to model the new LIO instruction-based scheduler.

3.6.2 Cache-aware semantics

We model the underlying CPU cache as an abstract memory shared among all running threads, which we will denote with the symbol ζ. Every step of the sequential execution relation will affect ζ according to the current instruction being executed, the runtime environment, and the existing state of the cache. As in [Stefan et al., 2012], each LIO thread has a thread-local runtime environment σ, which contains the current label $\sigma.\mathrm{lbl}$. The global environment Σ, common to all threads, holds references to shared resources.

In addition, we explicitly model the number of machine cycles taken by a single execution step as a result of the cache. Specifically, the transition

[2] As described in [Stefan et al., 2011], LIO does, however, allow programmers to heterogeneously label data they consider sensitive.

$$\text{(STEP)}\quad \frac{\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \longrightarrow_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'} \qquad q > 0}{\langle \Sigma, \zeta, q, \langle \sigma, e\rangle \triangleleft t_s\rangle \hookrightarrow \langle \Sigma', \zeta', q-k, \langle \sigma', e'\rangle \triangleleft t_s\rangle}$$

$$\text{(PREEMPT)}\quad \frac{q \leq 0}{\langle \Sigma, \zeta, q, t \triangleleft t_s\rangle \hookrightarrow \langle \Sigma', \zeta, q_i, t_s \triangleright t\rangle}$$

Fig. 6. Semantics for threadpools under round-robin time-based scheduling

$\zeta \rightharpoonup_k^{(\Sigma,\sigma,e)} \zeta'$ captures the parameters that influence the cache (Σ, σ, and e) as well as the number of cycles k it takes for the cache to be updated.

A *cache-aware* evaluation step is obtained by merging the reduction rule of LIO with our formalization of CPU cache as given below:

$$\frac{\langle \Sigma, \langle \sigma, e\rangle\rangle \xrightarrow{\gamma} \langle \Sigma', \langle \sigma', e'\rangle\rangle \qquad \zeta \rightharpoonup_k^{(\Sigma,\sigma,e)} \zeta' \qquad k \geq 1}{\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \xrightarrow{\gamma}_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'}}$$

We read $\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \xrightarrow{\gamma}_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'}$ as "the configuration $\langle \Sigma, \langle \sigma, e\rangle\rangle$ reduces to
$\langle \Sigma', \langle \sigma', e'\rangle\rangle$ in one step, but k machine cycles, producing event γ and modifying the cache from ζ to ζ'." As in LIO [Stefan et al., 2012], the relation $\langle \Sigma, \langle \sigma, e\rangle\rangle \xrightarrow{\gamma} \langle \Sigma', \langle \sigma', e'\rangle\rangle$ represents a single execution step from thread expression e, under the run-time environments Σ and σ, to thread expression e' and run-time environments Σ' and σ'. Events are used to communicate information between the threads and the scheduler, e.g., when spawning new threads.

Figure 6 shows the most important rules of our time-based scheduler in the presence of cache effects. We elide the rest of the rules for brevity. The relation $\hookrightarrow$ represents a single evaluation step for the program threadpool, in contrast with $\longrightarrow$ which is only for a single thread. Configurations are of the form $\langle \Sigma, \zeta, q, t_s\rangle$, where q is the number of cycles available in the current time slice and t_s is a queue of thread configurations of the form $\langle \sigma, e\rangle$. We use a standard deque-like interface with operations $\triangleleft$ and $\triangleright$ for front and back insertion, respectively, i.e., $\langle \sigma, e\rangle \triangleleft t_s$ denotes a threadpool in which the first thread is $\langle \sigma, e\rangle$ while $t_s \triangleright \langle \sigma, e\rangle$ indicates that $\langle \sigma, e\rangle$ is the last one.

As in LIO, threads are scheduled in a round-robin fashion. Our scheduler relies on the number of cycles that each step takes; we respectively write q_i and q as the initial and remaining number of cycles assigned to a thread in each quantum. In rule (STEP), the number of cycles k that the current instruction takes is reflected in the scheduling quantum. Consequently, threads that compute on data that is not present in the cache will take more cycles, i.e., have a higher k, so they will run "slower" because they are

allowed to perform fewer reduction steps in the remaining time slice. In practice, this permits attacks, such as that in Figure 1, where the interleaving of the threads can be affected by sensitive data. Rule (PREEMPT) is used when the thread has exhausted its cycle budget, triggering a context switch by moving the current thread to the end of the queue.

We can adapt this semantics to reflect the behavior of the new instruction-based scheduler. To this end, we replace the number of cycles q with an instruction budget; we write b_i for the initial instruction budget and b for the current budget. Crucially, we change rule (STEP) into rule (STEP-CA), given by

$$\text{(STEP-CA)}\quad \frac{\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \longrightarrow_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'} \qquad b > 0}{\langle \Sigma, \zeta, b, \langle \sigma, e\rangle \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta', b-1, \langle \sigma', e'\rangle \lhd t_s\rangle}.$$

Rule (STEP-CA) executes a sequential instruction in the current thread, provided the instruction budget is not empty ($b > 0$), and updates the cache accordingly ($\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \longrightarrow_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'}$). It is important to remark that the effects of the underlying cache ζ, as indicated by k, are intentionally ignored by the scheduler. This subtle detail captures the essence of removing the cache-based internal timing channel. (Our formalization of a time-based scheduler does not ignore k and thus is vulnerable.) Similarly, rule (PREEMPT) turns into rule (PREEMPT-CA), where q and q_i are respectively replaced with b and b_i to reflect the fact that there is an instruction budget instead of a cycle count. The rest of the rules can be adapted in a straightforward manner. Our rules have the invariant that the instruction budget gets decremented by one when a thread executes one instruction.

By changing the cache-aware semantics in this way, we obtain a generalized semantics for LIO. In fact, the previous semantics for LIO [Stefan et al., 2012], is a special case, with $b_i = 1$, i.e., the threads perform only one reduction step before a context-switch happens. In addition, it is easy to extend our previous termination-sensitive non-interference result to the instruction-based semantics. The security guarantees of our approach are stated below.

Theorem 1 (Termination-sensitive non-interference). *Given a program function f, an attacker that observes data at level L, and a pair of inputs e_1 and e_2 indistinguishable to the attacker, then for every reduction sequence starting from $f(e_1)$ there is a corresponding reduction sequence starting from $f(e_2)$ such that both sequences reach indistinguishable configurations.*

Proof Sketch: Our proof relies on the *term erasure technique* as used in [Li and Zdancewic, 2010, Russo et al., 2008, Stefan et al., 2011], and follows in a similar fashion to that of [Stefan et al., 2012]. More details can be found in Appendix A.

3.7 Limitations

This section discusses some limitations of our current implementation, the significance of these limitations, and how the limitations can be addressed.

Nondeterminism in the hardware counters While the retired-instruction counter should be deterministic, in most hardware implementations there is some degree of nondeterminism. For example, on most x86 processors the instruction counter adds an extra instruction every time a hardware interrupt occurs [Weaver and McKee, 2008]. This anomaly could be exploited to affect the behavior of an instruction-based scheduler, causing it to trigger a signal early. However, this is only a problem if a high thread is able to cause a large number of hardware interrupts in the underlying operating system. In the Hails framework, attackers can trigger interrupts by forcing a server to frequently receive HTTP responses, i.e., trigger a hardware interrupt from the network interface card. Hails, however, provides mechanisms to mitigate the effects of external events, using the techniques of [Askarov et al., 2010, Zhang et al., 2011], that can reduce the frequency of such operations. Nevertheless, the feasibility of such attacks is not directly clear and left as future work.

Scheduler and garbage collector instruction counts For performance reasons, we do not reset the retired-instruction counter prior to re-entering user code. This means that instruction counts include the instructions executed from when the previous thread received the signal, to when the previous thread yields, to when the next thread is scheduled. While this suggests that thread are not completely isolated, we think that this interaction is extremely difficult to exploit. This is because the number of instructions it takes for the scheduler to schedule a new thread is essentially fixed, and the "time to yield" for any code is highly dependent on the compiler, which we assume is not under the control of an adversary.

Parallelism Unfortunately, we cannot simply run instruction-based scheduling on multiple cores. Threads running in parallel will be able to race to public resources. Under normal conditions, such races can be still influenced by the state of the (L3) cache. Some parallelism is, however, possible. For instance, we can extend the instruction-based scheduler to parallelize regions of code that do not share state or have side effects (e.g., synchronization operations or writes to channels). To this end, when a thread wishes to perform a side effect, it is required that all the other threads lagging behind (as per retired-instruction count) first complete the execution of their side effects. Hence, an implementation would rely on a synchronization barrier whenever a side-effecting computation is executed; at the barrier, the execution of all the side effects is done in a pre-determined order. Although we believe that this "optimization" is viable, we have not implemented it,

since it requires major modifications to the GHC runtime system and the performance gains due to parallelism requiring such strict synchronization barriers are not clear. We leave this investigation to future work.

Even without built-in parallelism, we believe that instruction-based scheduling represents a viable and deployable solution when considering modern web applications and data-centers. In particular, when an application is distributed over multiple machines, these machines do not share a processor cache and thus can safely run the application concurrently. Attacks which involve making these two machines access shared external resources can be mitigated in the same fashion as external timing attacks [Askarov et al., 2010, Zhang et al., 2011, 2012, Stefan et al., 2012]. Load-balancing an application in this manner is already a well-established technique for deploying applications.

3.8 Related work

Impact of cache on cryptosystems Kocher [Kocher, 1996] was one of the first to consider the security implications of memory access-time in implementations of cryptographic primitives and systems. Since then, several attacks (e.g., [Percival, 2005, Osvik et al., 2006]) against popular systems have successfully extracted secret keys by using the cache as a covert channel. As a countermeasure, several authors propose partitioning the cache (e.g., [Page, 2005]). Until recently, partitioned caches have been of limited application in dynamic information flow control systems due to the small number of partitions available. The recent Vantage cache partition scheme of Sanchez and Kozyrakis [Sanchez and Kozyrakis, 2011], however, offers tens to hundreds of configurable partitions and high performance. As hardware is not yet available with Vantage, it is hard to evaluate its effectiveness for our problem domain. However, we expect it to be mostly complimentary to our instruction-based scheduler. Specifically, a partitioned cache can be used to safely run threads in parallel, each group of threads using instruction-based schedulers. Other countermeasures (e.g., [Osvik et al., 2006]) are primarily implementation-specific, and, while applicable to cryptographic primitives, they do not easily generalize to arbitrary code.

Language-based information-flow security Several works (e.g., [Honda et al., 2000]) consider systems that satisfy the notion of *possibilistic non-interference* [Smith and Volpano, 1998], which states that a concurrent program is secure iff the possible observable events do not depend on sensitive data. An alternative notion, *probabilistic non-interference*, considers a concurrent program secure iff the probability distribution over observable events is not affected by sensitive data [Volpano and Smith, 1999]. Zdancewic and Myers introduce *observational low-determinism* [Zdancewic and Myers, 2003], which intuitively states that the observable behavior of

concurrent systems must be deterministic. After this seminal work, several authors improve on each other's definitions on low-determinism (e.g., [Huisman et al., 2006]). Other IFC systems rely on deterministic semantics and a determined class of runtime schedulers (e.g., [Russo and Sabelfeld, 2006a]).

The lines of work mentioned above assume that the execution of a single step is performed in a single unit of time, corresponding to an instruction, and show that races to publicly-observable events cannot be influenced by secret data. Unfortunately, the presence of the cache breaks the correspondence between an instruction and a single unit of time, making cache attacks viable. Instruction-based scheduling could be seen as a necessary component in making the previous concurrent IFC approaches practical.

Agat [Agat, 2000] presents a code transformation for sequential programs such that both code paths of a branch have the same memory access pattern. This eliminates timing covert channels, even those relying on the cache. This transformation has been adapted by several authors (e.g., [Sabelfeld and Sands, 2000]). This approach, however, focuses on avoiding attacks relying on the data cache, while leaving the instruction cache unattended.

Russo and Sabelfeld [Russo and Sabelfeld, 2006b] consider non-interference for concurrent systems under cooperative and deterministic scheduling. An implementation of such a system was presented by Tsai et al. in [Tsai et al., 2007]. This approach eliminates internal timing leaks, including those relying on the cache, by restricting the use of yields. Cooperative schedulers are intrinsically vulnerable to attacks that use termination as a covert channel. In contrast, our solution is able to safely preempt non-terminating computations while guaranteeing termination-sensitive non-interference.

Secure multi-execution [Devriese and Piessens, 2010] preserves confidentiality of data by executing the same sequential program several times, one for each security level. In this scenario, the cache-based covert channel can only be removed in specific configurations [Kashyap et al., 2011]. Zhang et al. [Zhang et al., 2012] provide a method to mitigate external events when their timing behavior could be affected by the underlying hardware. This solution is directly applicable to our system when considering external events. Similar to our work, they consider an abstract model of the hardware machine state which includes a description of time. However, their semantics focus on sequential programs, wherein attacks due to the cache arise in the form of externally visible events.

Hedin and Sands [Hedin and Sands, 2005] present a type-system for preventing external timing attacks for bytecode. Their semantics is augmented to incorporate history, which enables the modeling of cache effects. We proceed in a similar manner when extending the original LIO semantics [Stefan et al., 2012] to consider caches.

System security In order to achieve strong isolation, Barthe et al. [Barthe et al., 2012] present a model of virtualization which flushes the cache upon

switching between guest operating systems. Different from our scenario, flushing the cache in such scenarios is common and does not impact the already-costly context-switch.

Allowing some information leakage, Köpft et al. [Köpf et al., 2012] combines abstract interpretation and quantitative information-flow to analyze leakage bounds for cache attacks. Kim et al. [Kim et al., 2012] propose StealthMem, a system level protection against cache attacks. StealthMem allows programs to allocate memory which does not get evicted from the cache. In fact, this approach could be seen as a software-level partition of the cache. StealthMem is capable of enforcing confidentiality for a stronger attacker model than ours, i.e., they consider programs with access to wall-clock and perhaps running on multi-cores. As other works on partition caches, StealthMem does not scale to scenarios with arbitrarily complex security lattices.

Performance monitoring counters The use of PMUs for tasks other than performance monitoring is a relatively recent one. Vogl and Ekert [Vogl and Eckert, 2012] also use PMUs, but for monitoring applications running within a virtual machine, allowing instruction level monitoring of all or specific instructions. While the mechanism is the same, our goals are different: we merely seek to replace interrupts generated by a clock-based timer with interrupts generated by hardware counters; their work introduces new interrupts that trigger vmexits. This causes a considerable slowdown, while we achieve no major performance impact.

3.9 Conclusion

Cache-based internal timing attacks constitute a practical set of attacks. We present instruction-based scheduling as a solution to remove such attacks. Different from simply flushing the cache on a context switch or partitioning the cache, this new class of schedulers also removes timing perturbations introduced by other components of the underlying hardware (e.g., the TLB, CPU buses, etc.). To demonstrate the applicability of our solution, we implemented a scheduler using the CPU retired-instruction counters available on commodity Intel and AMD hardware. We integrated the scheduler into the Hails IFC web framework, replacing the timing-based scheduler. This integration was, in part, possible because of the scheduler's negligible performance impact and, in part, due to our formal guarantees. Specifically, by generalizing previous results, we proved that instruction-based scheduling for LIO preserves confidentiality and integrity of data, i.e., termination-sensitive non-interference. Finally, we remark that our design, implementation, and proof are not limited to LIO; we believe that instruction-based scheduling is applicable to other concurrent deterministic IFC systems where cache-based timing attacks could be a concern.

Acknowledgments

This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by multiple gifts from Google, and by the Swedish research agency VR and STINT. Deian Stefan is supported by the DoD through the NDSEG Fellowship Program.

References

J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, January 2000.

Arbob Ahmad and Henry DeYoung. Cache performance of lazy functional programs on current hardware. Technical report, CMU, December 2009.

AMD. BIOS and kernel developer's guide for AMD family 11h processors, July 2008.

Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proc. of the 17th ACM CCS*. ACM, 2010.

G. Barthe, G. Betarte, J.D. Campo, and C. Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE Computer Society, june 2012.

Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153, May 2006.

J. Bonneau and I. Mironov. Cache-collision timing attacks against AES. *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 201–215, 2006.

D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. of the 2006 Ottawa Linux Symposium*, pages 269–288. Citeseer, 2006.

GHC. Infinite loops can hang Concurrent Haskell. http://hackage.haskell.org/trac/ghc/ticket/367, 2005.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, October 2012.

Daniel Hedin and David Sands. Timing aware information flow security for a JavaCard-like bytecode. *Elec. Notes Theor. Comput. Sci.*, 141, 2005.

Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *Proc. of the 9th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2000.

M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Sec. Foundations Workshop*, July 2006.

Intel. Intel 64 and IA-32 architectures software developer's manual, August 2012.

V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE, 2011.

Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12. USENIX Association, 2012.

Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. of the 16th CRYPTO*. Springer-Verlag, 1996.

Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *Proc. of the Intl. conference on Computer Aided Verification*. Springer-Verlag, 2012.

Maxwell Krohn, Alex Yip, Micah Brodsky, Robert Morris, and Michael Walfish. A World Wide Web Without Walls. In *6th ACM Workshop on Hot Topics in Networking (Hotnets)*, Nov. 2007a.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st Symp. on Operating Systems Principles*, October 2007b.

Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. of the Intl. Symposium on High Performance Computer Architecture*. IEEE, 2008.

Jonathan Millen. 20 years of covert channel modeling and analysis. In *IEEE Symp. on Security and Privacy*, 1999.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: from general purpose to a proof of information flow enforcement. In *Proceedings of the34th IEEE Symp. on Security and Privacy*, 2013.

Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.

Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, CT-RSA'06. Springer-Verlag, 2006.

Dan Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint Archive*, 2005, 2005.

Will Partain. The nofib benchmark suite of Haskell programs. *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, 1992.

Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 177–189, July 2006a.

A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (PSI)*, LNCS. Springer-Verlag, June 2006b.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM Press, September 2008.

A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 200–214, July 2000.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *International Symposium on Computer Architecture*. ACM IEEE, 2011.

G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, January 1998.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the 17th ACM SIGPLAN International Conference on Functional Programming*, Sep. 2012.

T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. IEEE Computer Sec. Foundations Symposium*, July 2007.

Sebastian Vogl and Claudia Eckert. Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture. *Proceedings of the 2012 European Workshop on System Security EuroSec'12*, 2012.

D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3), November 1999.

Vincent M. Weaver and Sally A. McKee. Can hardware performance counters be trusted? *Workload Characterization.*, 08, 2008. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4636099.

S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 29–43, June 2003.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proc. of the 18th ACM CCS*. ACM, 2011.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proc. of PLDI*. ACM, 2012.

A Formalization of LIO with instruction-based scheduling

LIO is formalized as a simply typed Curry-style call-by-name λ-calculus with some extensions. Figure 7 defines the formal syntax for the language. Syntactic categories v, e, and τ represent values, expressions, and types, respectively.

The values in the calculus have their usual meaning for typed λ-calculi. Symbol m represents LMVars. Special syntax nodes are added to this category: $\mathrm{Lb}\ v\ e$, $(e)^{\mathrm{LIO}}$, $\mathrm{R}\ m$, and $\boxdot$. Node $\mathrm{Lb}\ v\ e$ denotes the run-time representation of a labeled value. Similarly, node $(e)^{\mathrm{LIO}}$ denotes the run-time result of a monadic LIO computation. Node $\boxdot$ denotes the run-time representation of an empty LMVar. Node $\mathrm{R}\ m$ is the run-time representation of a Result, implemented as a LMVar, that is used to access the result produced by spawned computations.

$$
\begin{array}{rl}
\text{Label:} & l \\
\text{LMVar:} & m \\
\text{Value:} & v ::= \mathrm{true} \mid \mathrm{false} \mid () \mid l \mid m \mid x \mid \lambda x.e \\
 & \quad \mid \mathrm{fix}\ e \mid \mathrm{Lb}\ l\ e \mid (e)^{\mathrm{LIO}} \mid \boxdot \mid \mathrm{R}\ m \\
\text{Expression:} & e ::= v \mid \bullet \mid e\ e \mid \mathrm{if}\ e\ \mathrm{then}\ e\ \mathrm{else}\ e \\
 & \quad \mid \mathrm{let}\ x = e\ \mathrm{in}\ e \mid \mathrm{return}\ e \mid e \gg\!= e \\
 & \quad \mid \mathrm{label}\ e\ e \mid \mathrm{unlabel}\ e \mid \mathrm{getLabel} \\
 & \quad \mid \mathrm{labelOf}\ e \mid \mathrm{lFork}\ e\ e \mid \mathrm{lWait}\ e \\
 & \quad \mid \mathrm{newLMVar}\ e\ e \mid \mathrm{takeLMVar}\ e \\
 & \quad \mid \mathrm{putLMVar}\ e\ e \mid \mathrm{labelOfLMVar}\ e \\
\text{Type:} & \tau ::= \mathrm{Bool} \mid () \mid \tau \to \tau \mid \ell \mid \mathrm{Labeled}\ \ell\ \tau \\
 & \quad \mid \texttt{Result}\ \ell\ \tau \mid \mathrm{LMVar}\ \ell\ \tau \mid \mathrm{LIO}\ \ell\ \tau
\end{array}
$$

Fig. 7. Syntax for values, expressions, and types.

Expressions are composed of values (v), the special node $\bullet$, representing an erased term, function applications ($e\ e$), conditional branches ($\mathrm{if}\ e\ \mathrm{then}\ e\ \mathrm{else}\ e$), and local definitions ($\mathrm{let}\ x = e\ \mathrm{in}\ e$). Additionally, expressions may involve operations related to monadic computations in the LIO monad. More precisely, $\mathrm{return}\ e$ and $e \gg\!= e$ represent the monadic return and bind operations. Monadic operations related to the manipulation of labeled values inside the LIO monad are given by label and unlabel. Expression $\mathrm{unlabel}\ e$ acquires the content of the labeled value e while in an LIO computation. Expression $\mathrm{label}\ e_1\ e_2$ creates a labeled value, with label e_1, of the result obtained by evaluating the LIO computation e_2. Expression

lFork e_1 e_2 spawns a thread that computes e_2 and returns a handle with label e_1. Expression lWait e inspects the value returned by the spawned computation whose result is accessed by the handle e. Creating, reading, and writing labeled MVars are respectively captured by expressions newLMVar, takeLMVar, and putLMVar.

We consider standard types for Booleans (Bool), unit (()), and function ($\tau \to \tau$) values. Type ℓ describes security labels. Type `Result` $\ell\ \tau$ denotes handles used to access labeled results produced by spawned computations, where the results are of type τ and labeled with labels of type ℓ. Type LMVar $\ell\ \tau$ describes labeled MVars, with labels of type ℓ and storing values of type τ. Type LIO $\ell\ \tau$ represents monadic LIO computations, with a result type τ and the security labels of type ℓ.

As in [Stefan et al., 2012], we consider that each thread has a thread-local runtime environment σ, which contains the current label σ.lbl and the current clearance σ.clr. The global environment Σ, common to every thread, holds the global memory store ϕ, which is a mapping from LMVar names to Lb nodes.

The relation $\langle \Sigma, \langle \sigma, e \rangle \rangle \xrightarrow{\gamma} \langle \Sigma', \langle \sigma', e' \rangle \rangle$ represents a single execution step from thread e, under the run-time environments Σ and σ, to thread e' and run-time environments Σ' and σ'. (This relation does not account for the effects of the cache.) We say that e reduces to e' in one step. Symbol γ ranges over the *internal* events triggered by threads. We utilize internal events to communicate between the threads and the scheduler, e.g., when spawning new threads.

We show the most relevant rules for $\xrightarrow{\gamma}$ in Figure 8. Rule (LAB) generates a labeled value if and only if the label is between the current label and clearance of the LIO computation. Rule (UNLAB) requires that, when the content of a labeled value is "retrieved" and used in a LIO computation, the current label is raised ($\sigma' = \sigma[\text{lbl} \mapsto l']$, where $l' = \sigma.\text{lbl} \sqcup l$), thus capturing the fact that the remaining computation might depend on e. Rule (LFORK) allows for the creation of a thread and generates the internal event fork(e'), where e' is the computation to spawn. The rule allocates a new LMVar in order to store the result produced by the spawned thread ($e \gg\!= \lambda x.\text{putLMVar}\ m\ x$). Using that LMVar, the rule provides a handle to access to the thread's result (return (R m)). Rule (LWAIT) simply uses the LMVar for the handle. Rule (NLMVAR) describes the creation of a new LMVar with a label bounded by the current label and clearance ($\sigma.\text{lbl} \sqsubseteq l \sqsubseteq \sigma.\text{clr}$). Rule (TLMVAR) raises the current label ($\sigma' = \sigma[\text{lbl} \mapsto \sigma.\text{lbl} \sqcup l]$) when emptying ($\Sigma.\phi[m \mapsto \text{Lb}\ l\ \boxdot]$) its content ($\Sigma.\phi(m) = \text{Lb}\ l\ e$). Similarly, considering the security level l of a LMVar, rule (PLMVAR) raises the current label ($\sigma' = \sigma[\text{lbl} \mapsto \sigma.\text{lbl} \sqcup l]$) when filling ($\Sigma.\phi[m \mapsto \text{Lb}\ l\ e]$) its content ($\Sigma.\phi(m) = \text{Lb}\ l\ \boxdot$). Note that both takeLMVar and putLMVar observe if the LMVar is empty in order to proceed to modify its content. Precisely, takeLMVar and putLMVar perform a read and a write of the

$$\text{(Lab)}\quad \frac{\sigma.\mathrm{lbl} \sqsubseteq l \sqsubseteq \sigma.\mathrm{clr}}{\langle \Sigma, \langle \sigma, E[\mathrm{label}\ l\ e]\rangle\rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\mathrm{return}\ (\mathrm{Lb}\ l\ e)]\rangle\rangle}$$

$$\text{(unLab)}\quad \frac{l' = \sigma.\mathrm{lbl} \sqcup l \qquad l' \sqsubseteq \sigma.\mathrm{clr} \qquad \sigma' = \sigma[\mathrm{lbl} \mapsto l']}{\langle \Sigma, \langle \sigma, E[\mathrm{unlabel}\ (\mathrm{Lb}\ l\ e)]\rangle\rangle \longrightarrow \langle \Sigma, \langle \sigma', E[\mathrm{return}\ e]\rangle\rangle}$$

$$\text{(lFork)}\quad \frac{\sigma.\mathrm{lbl} \sqsubseteq l \sqsubseteq \sigma.\mathrm{clr} \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \mathrm{Lb}\ l\ \boxdot]] \qquad e' = e \gg\!= \lambda x.\mathrm{putLMVar}\ m\ x \qquad m\ \mathrm{fresh}}{\langle \Sigma, \langle \sigma, E[\mathrm{lFork}\ l\ e]\rangle\rangle \xrightarrow{\mathrm{fork}(e')} \langle \Sigma', \langle \sigma, E[\mathrm{return}\ (\mathrm{R}\ m)]\rangle\rangle}$$

$$\text{(lWait)}\quad \langle \Sigma, \langle \sigma, E[\mathrm{lWait}\ (\mathrm{R}\ m)]\rangle\rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\mathrm{takeLMVar}\ m]\rangle\rangle$$

$$\text{(nLMVar)}\quad \frac{\sigma.\mathrm{lbl} \sqsubseteq l \sqsubseteq \sigma.\mathrm{clr} \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \mathrm{Lb}\ l\ e]] \qquad m\ \textit{fresh}}{\langle \Sigma, \langle \sigma, E[\mathrm{newLMVar}\ l\ e]\rangle\rangle \longrightarrow \langle \Sigma', \langle \sigma, E[\mathrm{return}\ m]\rangle\rangle}$$

$$\text{(tLMVar)}\quad \frac{\Sigma.\phi(m) = \mathrm{Lb}\ l\ e \qquad e \neq \boxdot \qquad \sigma.\mathrm{lbl} \sqsubseteq l \sqsubseteq \sigma.\mathrm{clr} \qquad \sigma' = \sigma[\mathrm{lbl} \mapsto \sigma.\mathrm{lbl} \sqcup l] \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \mathrm{Lb}\ l\ \boxdot]]}{\langle \Sigma, \langle \sigma, E[\mathrm{takeLMVar}\ m]\rangle\rangle \longrightarrow \langle \Sigma', \langle \sigma', E[\mathrm{return}\ e]\rangle\rangle}$$

$$\text{(pLMVar)}\quad \frac{\Sigma.\phi(m) = \mathrm{Lb}\ l\ \boxdot \qquad \sigma.\mathrm{lbl} \sqsubseteq l \sqsubseteq \sigma.\mathrm{clr} \qquad \sigma' = \sigma[\mathrm{lbl} \mapsto \sigma.\mathrm{lbl} \sqcup l] \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \mathrm{Lb}\ l\ e]]}{\langle \Sigma, \langle \sigma, E[\mathrm{putLMVar}\ m\ e]\rangle\rangle \longrightarrow \langle \Sigma', \langle \sigma', E[\mathrm{return}\ ()]\rangle\rangle}$$

Fig. 8. Semantics for expressions.

mutable location. Operations on LMVar are *bi-directional* and consequently the rules (tLMVar), and (pLMVar) require not only that the label of the mentioned LMVar be between the current label and current clearance of the thread ($\sigma.\mathrm{lbl} \sqsubseteq l \sqsubseteq \sigma.\mathrm{clr}$), but that the current label be raised appropriately.

A.1 Cache-aware semantics using instruction-based scheduling

Figure 9 presents cache-aware reduction rules for concurrent execution using instruction-based scheduling. The configurations for this relation are very similar to the ones for time-based scheduling in Figure 6 except that we use an instruction budget b rather than a time quantum q. We write b_i for the initial budget for threads.

The main difference between these semantics and the time-based ones is the cache-aware transition rule (Step-CA). In this rule, the number of cycles

k that the current instruction takes is ignored by the scheduler, counting as one instruction regardless of the time its execution took.

$$\text{(STEP-CA)}\quad \frac{\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \longrightarrow_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'} \qquad q > 0}{\langle \Sigma, \zeta, b, \langle \sigma, e\rangle \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta', b-1, \langle \sigma', e'\rangle \lhd t_s\rangle}$$

$$\text{(PREEMPT-CA)}\quad \frac{q \le 0}{\langle \Sigma, \zeta, b, t \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta, b_i, t_s \rhd t\rangle} \qquad \text{(NO-STEP-CA)}\quad \frac{\langle \Sigma, t\rangle_\zeta \not\longrightarrow \qquad t = \langle \sigma, e\rangle \qquad e \neq v}{\langle \Sigma, \zeta, b, t \lhd t_s\rangle \hookrightarrow \langle \Sigma, \zeta, b_i, t_s \rhd t\rangle}$$

$$\text{(FORK-CA)}\quad \frac{\langle \Sigma, t\rangle_\zeta \xrightarrow{\text{fork}(e)}_k \langle \Sigma', \langle \sigma, e'\rangle\rangle_{\zeta'} \qquad t_{\text{new}} = \langle \sigma, e\rangle \qquad q > 0}{\langle \Sigma, \zeta, b, t \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta', b-1, \langle \sigma, e'\rangle \lhd t_s \rhd t_{\text{new}}\rangle}$$

$$\text{(EXIT-CA)}\quad \frac{\langle \Sigma, t\rangle_\zeta \longrightarrow_k \langle \Sigma', \langle \sigma, v\rangle\rangle_{\zeta'} \qquad b > 0}{\langle \Sigma, \zeta, b, t \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta', b_i, t_s\rangle}$$

Fig. 9. Semantics for threadpools under round-robin instruction-based scheduling

A.2 Security guarantees

In this section, we show that LIO computations satisfy termination-sensitive non-interference. As in [Li and Zdancewic, 2010, Russo et al., 2008, Stefan et al., 2011], we prove this property by using the *term erasure* technique. The erasure function ε_L rewrites data at security levels that the attacker cannot observe into the syntax node $\bullet$.

The function ε_L is defined in such a way that $\varepsilon_L(e)$ contains no information above level L, i.e., the function ε_L replaces all the information more sensitive than L in e with a hole ($\bullet$). In most of the cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon_L(e_1\ e_2) = \varepsilon_L(e_1)\ \varepsilon_L(e_2)$). For run expressions, the erasure function is mapped into all threads; all threads with a current label above L are removed from the pool (filter $(\lambda\langle \sigma, e\rangle.e \not\equiv \bullet)$ (map $\varepsilon_L\ t_s$), where $\equiv$ denotes syntactic equivalence). The computation performed in a certain sequential configuration is erased if the current label is above L. For runtime environments and stores, we map the erasure function into their components. Similarly, a labeled value is erased if the label assigned to it is above L.

Following the definition of the erasure function, we introduce a new evaluation relation $\longrightarrow_L$ as follows:

$$\frac{\langle \Sigma, \langle \sigma, t\rangle\rangle \longrightarrow \langle \Sigma', \langle \sigma', t\rangle\rangle}{\langle \Sigma, \langle \sigma, t\rangle\rangle \longrightarrow_L \varepsilon_L(\langle \Sigma', \langle \sigma', t'\rangle\rangle)}$$

The relation $\longrightarrow_L$ guarantees that confidential data, i.e., data not below level L, is erased as soon as it is created. We write $\longrightarrow_L^*$ for the reflexive and transitive closure of $\longrightarrow_L$.

In order to prove non-interference, we will establish a simulation relation between $\longrightarrow^*$ and $\longrightarrow_L^*$ through the erasure function: erasing all secret data and then taking evaluation steps in $\longrightarrow_L$ is equivalent to taking steps in $\longrightarrow$ first, and then erasing all secret values in the resulting configuration. Note that this relation would not hold if information from some level above L was being leaked by the program. In the rest of this section, we only consider well-typed terms to ensure there are no stuck configurations.

We start by showing that the evaluation relation $\longrightarrow_L$ is deterministic

Proposition 1 (Determinacy of $\longrightarrow_L$). *If $\langle \Sigma, t\rangle_\zeta \longrightarrow_L \langle \Sigma', t'\rangle_{\zeta'}$ and $\langle \Sigma, t\rangle_\zeta \longrightarrow_L \langle \Sigma'', t''\rangle_{\zeta''}$, then $\langle \Sigma', t'\rangle_{\zeta'} = \langle \Sigma'', t''\rangle_{\zeta''}$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step.

The next lemma establishes a simulation between $\hookrightarrow^*$ and $\hookrightarrow_L^*$.

Lemma 1 (Many-step simulation). *If $\langle \Sigma, \zeta, b, t_s\rangle \hookrightarrow^* \langle \Sigma', \zeta', b', t_s'\rangle$, then $\varepsilon_L(\langle \Sigma, \zeta, b, t_s\rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma', \zeta', b', t_s'\rangle)$.*

Proof. In order to prove this result, we rely on properties of the erasure function, such as the fact that it is idempotent and homomorphic to the application of evaluation contexts and substitution. We show that the result holds by case analysis on the rule used to derive $\langle \Sigma, t_s\rangle \hookrightarrow^* \langle \Sigma', t_s'\rangle$, and considering different cases for threads whose current label is below (or not) level L.

The L-equivalence relation $\approx_L$ is an equivalence relation between configurations (and their parts), defined as the equivalence kernel of the erasure function ε_L: $\langle \Sigma, \zeta, b, t_s\rangle \approx_L \langle \Sigma', \zeta', b', r_s\rangle$ iff $\varepsilon_L(\langle \Sigma, \zeta, b, t_s\rangle) = \varepsilon_L(\langle \Sigma', \zeta', b', r_s\rangle)$. If two configurations are L-equivalent, they agree on all data below or at level L, i.e., they cannot be distinguished by an attacker at level L. Note that two queues are L-equivalent iff the threads with current label no higher than L are pairwise L-equivalent in the order that they appear in the queue.

The next theorem shows the non-interference property. It essentially states that if we take two executions of a program with two L-equivalent inputs, then for every intermediate step of the computation of the first run, there is a corresponding step in the computation of the second run which results in an L-equivalent configuration.

Theorem 2 (Termination-sensitive non-interference). *Given a computation* e *(with no* Lb, $()^{\mathrm{LIO}}$, $\boxdot$, R, *and* $\bullet$*) where* $\Gamma \vdash e : \mathrm{Labeled}\ \ell\ \tau \rightarrow$ $\mathrm{LIO}\ \ell\ (\mathrm{Labeled}\ \ell\ \tau')$*, an attacker at level* L*, an initial securiy context* σ*, runtime environments* Σ_1 *and* Σ_2 *where* $\Sigma_1.\phi = \Sigma_2.\phi = \varnothing$*, and initial cache states* ζ_1 *and* ζ_2*, then*

$$\begin{aligned} &\forall e_1 e_2.(\Gamma \vdash e_i : \mathrm{Labeled}\ \ell\ \tau)_{i=1,2} \wedge\ e_1 \approx_L e_2 \\ &\qquad \wedge \langle \Sigma_1, \zeta_1, b_i, \langle \sigma, e\ e_1 \rangle \rangle \hookrightarrow^* \langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle \\ \Rightarrow\ &\exists \Sigma_2' \zeta_2' b_2' t_s^2 . \langle \Sigma_2, \zeta_2, b_i, \langle \sigma, e\ e_2 \rangle \rangle \hookrightarrow^* \langle \Sigma_2', \zeta_2', b_2', t_s^2 \rangle \\ &\qquad \wedge \langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle \approx_L \langle \Sigma_2', \zeta_2', b_2', t_s^2 \rangle \end{aligned}$$

Proof. Take $\langle \Sigma_1, \zeta_1, b_i, \langle \sigma, e\ e_1 \rangle \rangle \hookrightarrow^* \langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle$ and apply Lemma 1 to get $\varepsilon_L(\langle \Sigma_1, \zeta_1, b_i, \langle \sigma, e\ e_1 \rangle \rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle)$. We know this reduction only includes public ($\sqsubseteq L$) steps, so the number of steps is lower than or equal to the number of steps in the first reduction.

We can always find a reduction starting from $\varepsilon_L(\langle \Sigma_2, \zeta_2, b_i, \langle \sigma, e\ e_2 \rangle \rangle)$ with the same number of steps as $\varepsilon_L(\langle \Sigma_1, \zeta_1, b_i, \langle \sigma, e\ e_1 \rangle \rangle) \hookrightarrow_L^*$ $\varepsilon_L(\langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle)$, so by the Determinacy Lemma we have $\varepsilon_L(\langle \Sigma_2, \zeta_2, b_i, \langle \sigma, e\ e_2 \rangle \rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_2', \zeta_2', b_2', t_s^2 \rangle)$. By Lemma 1 again, we get $\langle \Sigma_2, \zeta_2, b_i, \langle \sigma, e\ e_2 \rangle \rangle \hookrightarrow^* \langle \Sigma_2', \zeta_2', b_2', t_s^2 \rangle$ and therefore $\langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle \approx_L$ $\langle \Sigma_2', \zeta_2', b_2', t_s^2 \rangle$.

CHAPTER
FOUR

A LIBRARY FOR REMOVING CACHE-BASED ATTACKS IN CONCURRENT INFORMATION FLOW SYSTEMS

Pablo Buiras, Amit Levy, Deian Stefan
Alejandro Russo, David Mazières

Abstract. Information-flow control (IFC) is a security mechanism conceived to allow untrusted code to manipulate sensitive data without compromising confidentiality. Unfortunately, untrusted code might exploit some covert channels in order to reveal information. In this paper, we focus on the LIO concurrent IFC system. By leveraging the effects of hardware caches (e.g., the CPU cache), LIO is susceptible to attacks that leak information through the *internal timing covert channel*. We present a *resumption*-based approach to address such attacks. Resumptions provide fine-grained control over the interleaving of thread computations at the library level. Specifically, we remove cache-based attacks by enforcing that every thread yield after executing an "instruction," i.e., atomic action. Importantly, our library allows for porting the full LIO library—our resumption approach handles local state and exceptions, both features present in LIO. To amend for performance degradations due to the library-level thread scheduling, we provide two novel primitives. First, we supply a primitive for securely executing pure code in parallel. Second, we provide developers a primitive for controlling the granularity of "instructions"; this allows developers to adjust the frequency of context switching to suit application demands.

4.1 Introduction

Popular website platforms, such as Facebook, run third-party applications (apps) to enhance the user experience. Unfortunately, in most of today's platforms, once an app is installed it is usually granted full or partial access to the user's sensitive data—the users have no guarantees that their data is not arbitrarily ex-filtrated once apps are granted access to it [Krohn et al., 2007]. As demonstrated by Hails [Giffin et al., 2012], information-flow control (IFC) addresses many of these limitations by restricting how sensitive data is disseminated. While promising, IFC systems are not impervious to attacks; the presence of *covert channels* allows attackers to leak sensitive information.

Covert channels are mediums not intended for communication, which nevertheless can be used to carry and, thus, reveal information [Lampson, 1973]. In this work, we focus on the *internal timing covert channel* [Smith and Volpano, 1998]. This channel emanates from the mere presence of concurrency and shared resources. A system is said to have an internal timing covert channel when an attacker, as to reveal sensitive data, can alter *the order of public events* by affecting the timing behavior of threads. To avoid such attacks, several authors propose decoupling computations manipulating sensitive data from those writing into public resources (e.g., [Boudol and Castellani, 2001, 2002, Pottier, 2002, Russo et al., 2006, Stefan et al., 2012a]).

Decoupling computations by security levels only works when all shared resources are modeled. Similar to most IFC systems, the concurrent IFC system LIO [Stefan et al., 2012a] only models shared resources at the programming language level and does not explicitly consider the effects of hardware. As shown in [Stefan et al., 2013], LIO threads can exploit the underlying CPU cache to leak information through the internal timing covert channel.

We propose using *resumptions* to model interleaved computations. (We refer the interested reader to [Harrison, 2004] for an excellent survey of resumptions.) A resumption is either a (computed) value or an atomic action which, when executed, returns a new resumption. By expressing thread computations as a series of resumptions, we can leverage resumptions for controlling concurrency. Specifically, we can interleave atomic actions, or "instructions," from different threads, effectively forcing each thread to yield at deterministic points. This ensures that scheduling is not influenced by underlying caches and thus cannot be used to leak secret data. We address the attacks on the recent version of LIO [Stefan et al., 2012a] by implementing a Haskell library which ports the LIO API to use resumptions. Since LIO threads possess local state and handle exceptions, we extend resumptions to account for these features.

In principle, it is possible to force deterministic interleaving by means other than resumptions; in [Stefan et al., 2013] we show an instruction-

based scheduler that achieves this goal. However, Haskell's monad abstraction allows us to to easily model resumptions as a library. This has two consequences. First, and different from [Stefan et al., 2013], it allows us to deploy a version of LIO that does not rely on changes to the Haskell compiler. Importantly, LIO's concurrency primitives can be modularly redefined, with little effort, to operate on resumptions. Second, by effectively implementing "instruction based-scheduling" at the level of library primitives, we can address cache attacks not covered by the approach described in [Stefan et al., 2013] (see Section 4.5).

In practice, a library-level interleaved model of computations imposes performance penalties. With this in mind, we provide primitives that allow developers to execute code in parallel, and means for securely controlling the granularity of atomic actions (which directly affects performance).

Although our approach addresses internal timing attacks in the presence of shared hardware, the library suffers from leaks that exploit the termination channel, i.e., programs can leak information by not terminating. However, this channel can only be exploited by brute-force attacks that leak data external to the program—an attacker cannot leak data within the program, as can be done with the internal timing covert channel.

4.2 Cache Attacks on Concurrent IFC Systems

Figure 1 shows an attack that leverages the timing effects of the underlying cache in order to leak information through the internal timing covert channel. In isolation, all three threads are secure. However, when executed concurrently, threads B and C race to write to a public, shared variable `l`. Importantly, the race outcome depends on the state of the secret variable `h`, by changing the contents of underlying CPU cache according to its value (e.g., by creating and traversing a large array as to fill the cache with new data).

The attack proceeds as follows. First, thread A fills the cache with the contents of a public array `lowArray`. Then, depending on the secret variable `h`, it evicts data from the cache (by filling it with arbitrary

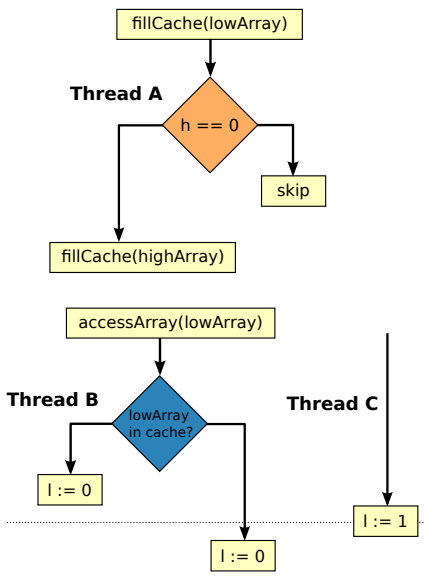

Fig. 1. Cache attack

data) or leaves it intact. Concurrently, public threads B and C delay execution long enough for A to finish. Subsequently, thread B accesses elements of the public array `lowArray`, and writes 0 to public variable `l`; if the array has been evicted from the cache (`h==0`), the amount of time it takes to perform the read, and thus the write to `l`, will be much longer than if the array is still in the cache. Hence, to leak the value of `h`, thread C simply needs to delay writing 1 to `l` long enough so that it is above the case where the cache is full (with the public array), but shorter than it take to refill the cache with the (public) array. Observing the contents of `l`, the attacker directly learns the value of `h`.

Appendix A shows the concrete code of the attack for LIO. This simple attack has previously been demonstrated in [Stefan et al., 2013], where confidential data from the GitStar system [Giffin et al., 2012], build atop LIO, was leaked. Such attacks are not limited to LIO or IFC systems; cache-based attacks against many system, including cryptographic primitives (e.g., RSA and AES), are well known [Wong, 2005, Osvik et al., 2006, Percival, 2005, Aciiçmez, 2007].

The next section details the use of resumptions in modeling concurrency at the programming language level by defining atomic steps, which are used as the thread scheduling quantum unit. By scheduling threads according to the number of executed atoms, the attack in Figure 1 is eliminated. As in [Stefan et al., 2013], this is the case because an atomic step runs till completion, regardless of the state of the cache. Hence, the timing behavior of thread B, which was previously leaked to thread C by the time of preemption, is no longer disclosed. Specifically, the scheduling of thread C's `l:=1` does not depend on the time it takes thread B to read the public array from the cache; rather it depends on the atomic actions, which do not depend on the cache state. In addition, our use of resumptions also eliminates attacks that exploit other timing perturbations produced by the underlying hardware, e.g., TLB misses, CPU bus contention, etc.

4.3 Modeling Concurrency with Resumptions

In pure functional languages, computations with side-effects are encoded as values of abstract data types called *monads* [Moggi, 1991]. We use the type $m\ a$ to denote computations that produce results of type a and may perform side-effects in monad m. Different side-effects are often handled by different monads. In Haskell, there are monads for performing inputs and outputs (monad IO), handling errors (monad $Error$), etc. The IFC system LIO simply exposes a monad, LIO, in which security checks are performed before any IO side-effecting action.

Resumptions are a simple approach to modeling interleaved computations of concurrent programs. A resumption, which has the form $res\ ::=$

```
data Thread m a where
  Done :: a → Thread m a
  Atom :: m (Thread m a)
       → Thread m a
  Fork :: Thread m ()
       → Thread m a
       → Thread m a
```

Fig. 2. Threads as Resumptions

```
sch :: [Thread m ()] → m ()
sch []                    = return ()
sch ((Done _) : thrds)    = sch thrds
sch ((Atom m) : thrds) =
  do res ← m; sch (thrds ++ [res])
sch ((Fork res res') : thrds) =
  sch ((res : thrds) ++ [res'])
```

Fig. 3. Simple round-robin scheduler

$x \mid \alpha \triangleright res$, is either a computed value x or an atomic action α followed by a new resumption res. Using this notion, we can break down a program that is composed of a series of instructions into a program that executes an atomic action and yields control to a scheduler by giving it its subsequent resumption. For example, program $P := i_1; i_2; i_3$, which performs three side-effecting instructions in sequence, can be written as $res_P := i_1; i_2 \triangleright i_3 \triangleright ()$, where $()$ is a value of a type with just one element, known as *unit*. Here, an atomic action α is any sequence of instructions. When executing res_P, instructions i_1 and i_2 execute atomically, after which it yields control back to the scheduler by supplying it the resumption $res'_P := i_3 \triangleright ()$. At this point, the scheduler may schedule atomic actions from other threads or execute res'_P to resume the execution of P. Suppose program $Q := j_1; j_2$, rewritten as $j_1 \triangleright j_2 \triangleright ()$, runs concurrently with P. Our concurrent execution of P and Q can be modeled with resumptions, under a round-robin scheduler, by writing it as $P||Q := i_1; i_2 \triangleright j_1 \triangleright i_3 \triangleright j_2 \triangleright ()$, where the return value of P (namely $()$) is discarded.

In other words, resumptions allow us to implement a scheduler that executes $i_1; i_2$, postponing the execution of i_3, and executing atomic actions from Q in the interim.

Implementing threads as resumptions As previously done in [Harrison, 2004, Harrison and Hook, 2005], Fig. 2 defines threads as resumptions at the programming language level. The thread type ($Thread\ m\ a$) is parametric in the resumption computation value type (a) and the monad in which atomic actions execute (m)[1]. (Symbol :: introduces type declarations and $\rightarrow$ denotes function types.) The definition has several value constructors for a thread. Constructor $Done$ captures computed values; a value $Done\ a$ represents the computed value a. Constructor $Atom$ captures a resumption of the form $\alpha \triangleright res$. Specifically, $Atom$ takes a monadic action of type $m\ (Thread\ m\ a)$, which denotes an atomic computation in monad m that returns a new resumption as a result. In other words, $Atom$ captures both the atomic action that is being executed (α) and the subsequent

[1] In our implementation, atomic actions α (as referred as in $\alpha \triangleright res$) are actions described by the monad m.

resumption (res). Finally, constructor $Fork$ captures the action of spawning new threads; value $Fork\ res\ res'$ encodes a computation wherein a new thread runs resumption res and the original thread continues as res'.[2] As in the standard Haskell libraries, we assume that a fork does not return the new thread's final value and thus the type of the new thread/resumption is simply $Thread\ m\ ()$.

Programming with resumptions Users do not build programs based on resumptions by directly using the constructors of $Thread\ m\ a$. Instead, they use the interface provided by Haskell monads:
$return :: a \to Thread\ m\ a$ and $(\gg\!=) :: Thread\ m\ a \to (a \to Thread\ m\ b) \to Thread\ m\ b$. The expression $return\ a$ creates a resumption which consists of the computed value a, i.e., it corresponds to $Done\ a$. The operator $(\gg\!=)$, called *bind*, is used to sequence atomic computations. Specifically, the expression $res \gg\!= f$ returns a resumption that consists of the execution of the atomic actions in res followed by the atomic actions obtained from applying f to the result produced by res. (The precise definition of $return$ and $\gg\!=$ can be found in Appendix B.) We sometimes use Haskell's **do**-notation to write such monadic computations. For example, the expression $res \gg\!= (\lambda a \to return\ (a+1))$, i.e., actions described by the resumption res followed by $return\ (a+1)$ where a is the result produced by res, is written as **do** $a \leftarrow res; return\ (a+1)$.

Scheduling computations We use round-robin to schedule atomic actions of different threads. Fig. 3 shows our scheduler implemented as a function from a list of threads into an interleaved computation in the monad m. The scheduler behaves as follows. If there is an empty list of resumptions, the scheduler, and thus the program, terminates. If the resumption at the head of the list is a computed value ($Done\ \_$), the scheduler removes it and continues scheduling the remaining threads (**sch** $thrds$). (Recall that we are primarily concerned with the side-effects produced by threads and not about their final values.) When the head of the list is an atomic step ($Atom\ m$), **sch** runs it ($res \leftarrow m$), takes the resulting resumption (res), and appends it to the end of the thread list (**sch** $(thrds +\!\!+ [res])$). Finally, when a thread is forked, i.e., the head of the list is a $Fork\ res\ res'$, the spawned resumption is placed at the front of the list ($res : thrds$). Observe that in both of the latter cases the scheduler is invoked recursively—hence we keep evaluating the program until there are no more threads to schedule. We note that although we choose a particular, simple scheduling approach, our results naturally extend for a wide class of deterministic schedulers [Russo and Sabelfeld, 2006a, Swierstra, 2008].

[2] Spawning threads could also be represented by a equivalent constructor $Fork' :: Thread\ m\ () \to Thread\ m\ a$, we choose $Fork$ for pedagogical reasons.

4.4 Extending Resumptions with State and Exceptions

LIO provides general programming language abstrations (e.g., state and exceptions), which our library must preserve to retain expressiveness. To this end, we extend the notion of resumptions and modify the scheduler to handle thread local state and exceptions.

Thread local state As described in [Stefan et al., 2011], the *LIO* monad keeps track of a *current label*, L_{cur}. This label is an upper bound on the labels of all data in lexical scope. When a computation C, with current label L_C, observes an object labeled L_O, C's label is raised to the least upper bound or *join* of the two labels, written $L_C \sqcup L_O$. Importantly, the current label governs where the current computation can write, what labels may be used when creating new channels or threads, etc. For example, after reading an object O, the computation should not be able to write to a channel K if L_O is more confidential than L_K—this would potentially leak sensitive information (about O) into a less sensitive channel. We write $L_C \sqsubseteq L_K$ when L_K at least as confidential as L_C and information is allowed to flow from the computation to the channel.

```
sch ((Atom m) : thrds) =
  do res ← m
     st  ← get
     sch (thrds ++ [put st ≻ res])
sch ((Fork res res') : thrds) =
  do st ← get
     sch ((res : thrds) ++ [put st ≻ res'])
```

Fig. 4. Context-switch of local state

Using our resumption definition of Section 4.3, we can model concurrent LIO programs as values of type $Thread\ LIO$. Unfortunately, such programs are overly restrictive—since LIO threads would be sharing a single current label—and do not allow for the implementation of many important applications. Instead, and as done in the concurrent version of LIO [Stefan et al., 2012a], we track the state of each thread, independently, by modifying resumptions, and the scheduler, with the ability to context-switch threads with state.

Figure 4 shows these changes to **sch**. The context-switching mechanism relies on the fact that monad m is a state monad, i.e., provides operations to retrieve (get) and set (put) its state. LIO is a state monad,[3] where the state contains (among other things) L_{cur}. Operation $(\succ) :: m\ b \to Thread\ m\ a \to Thread\ m\ a$ modifies a resumption in such a way that its first atomic

[3] For simplicity of exposition, we use get and set. However, LIO only provides such functions to trusted code. In fact, the monad LIO is not an instance of *MonadState* since this would allow untrusted code to arbitrarily modify the current label—a clear security violation.

step ($Atom$) is extended with $m\ b$ as the first action. Here, $Atom$ consists of executing the atomic step ($res \leftarrow m$), taking a snapshot of the state ($st \leftarrow get$), and restoring it when executing the thread again ($put\ st \succ res$). Similarly, the case for $Fork$ saves the state before creating the child thread and restores it when the parent thread executes again ($put\ st \succ res'$).

Exception handling As described in [Stefan et al., 2012b], LIO provides a secure way to throw and catch exceptions—a feature crucial to many real-world applications. Unfortunately, simply using LIO's *throw* and *catch* as atomic actions, as in the case of local state, results in non-standard behavior. In particular, in the interleaved computation produced by **sch**, an atomic action from a thread may throw an exception that would propagate outside the thread group and crash the program. Since we do not consider leaks due to termination, this does not impact security; however, it would have non-standard and restricted semantics. Hence, we first extend our scheduler to introduce a top-level *catch* for every spawned thread.

Besides such an extension, our approach still remains quite limiting. Specifically, LIO's *catch* is defined at the level of the monad *LIO*, i.e., it can only be used inside atomic steps. Therefore, catch-blocks are prevented from being extended beyond atomic actions. To address this limitation, we lift exception handling to work at the level of resumptions.

To this end, we consider a monad m that handles exceptions, i.e., a monad for which $throw :: e \to m\ a$ and $catch :: m\ a \to (e \to m\ a) \to m\ a$, where e is a type denoting exceptions, are accordingly defined. Function *throw* throws the exception supplied as an argument. Function *catch* runs the action supplied as the first argument ($m\ a$), and if an exception is thrown, then executes the handler ($e \to m\ a$) with the value of the exception passed as an argument. If no exceptions are raised, the result of the computation (of type a) is simply returned.

```
throw e = Atom (LIO.throw e)

catch (Done a) _ = Done a
catch (Atom a) handler =
  Atom (LIO.catch
    (do res ← a
        return (catch res handler))
    (λe → return (handler e)))
catch (Fork res res') handler =
  Fork res (catch res' handler)
```

Fig. 5. Exception handling for resumptions

Figure 5 shows the definition of exception handling for resumptions. Since *LIO* defines *throw* and *catch* [Stefan et al., 2012b], we qualify these underlying functions with *LIO* to distinguish them from our resumption-level *throw* and *catch*. When throwing an exception, the resumption simply executes an atomic step that throws the exception in *LIO* ($LIO.throw\ e$).

The definitions of *catch* for *Done* and *Fork* are self explanatory. The most interesting case for *catch* is when the resumption is an $Atom$. Here, *catch* applies $LIO.catch$ step by step to each atomic action in the sequence;

this is necessary because exceptions can only be caught in the *LIO* monad. As shown in Fig. 5, if no exception is thrown, we simply return the resumption produced by m. Conversely, if an exception is raised, $LIO.catch$ will trigger the exception handler which will return a resumption by applying the top-level $handler$ to the exception e. To clarify, consider catching an exception in the resumption $\alpha_1 \triangleright \alpha_2 \triangleright x$. Here, $catch$ executes α_1 as the first atomic step, and if no exception is raised, it executes α_2 as the next atomic step; on the other hand, if an exception is raised, the resumption $\alpha_2 \triangleright x$ is discarded and $catch$, instead, executes the resumption produced when applying the exception handler to the exception.

4.5 Performance Tuning

Unsurprisingly, interleaving computations at the library-level introduces performance degradation. To alleviate this, we provide primitives that allow developers to control the granularity of atomic steps—fine-grained atoms allow for more flexible programs, but also lead to more context switches and thus performance degradation (as we spend more time context switching). Additionally, we provide a primitive for the parallel execution of pure code. We describe these features—which do not affect our security guarantees—below.

Granularity of atomic steps To decrease the frequency of context switches, programmers can treat a complex set of atoms (which are composed using monadic bind) as a single atom using $singleAtom :: Thread\ m\ a \rightarrow Thread\ m\ a$. (See Appendix C.) This function takes a resumption and "compresses" all its atomic steps into one. Although $singleAtom$ may seem unsafe, e.g., because we do not restrict threads from adjust the granularity of atomic steps according to secrets, in Section 4.6 we show that this is not the case—it is the atomic execution of atoms, regardless of their granularity, that ensures security.

Parallelism As in [Stefan et al., 2013], we cannot run one scheduler **sch** per core to gain performance through parallelism. Threads running in parallel can still race to public resources, and thus vulnerable to internal timing attacks (that may, for example, rely on the L3 CPU cache). In principle, it is possible to securely parallelize arbitrary side-effecting computations if races (or their outcomes) to shared public resource are eliminated. Similar to *observational low-determinism* [Zdancewic and Myers, 2003], our library could allow parallel computations to compute on disjoint portions of the memory. However, whenever side-effecting computations follow parallel code, we would need to impose synchronization barriers to enforce that all side-effects are performed in a pre-determined order. It is precisely this order, and LIO's safe side-effecting primitives for shared-resources, that

hides the outcome of any potential dangerous parallel race. In this paper, we focus on executing pure code in parallel; we leave side-effecting code to future work.

Pure computations, by definition, cannot introduce races to shared resources since they do not produce side effects.[4] To consider such computations, we simply extend the definition of $Thread$ with a new constructor: $Parallel :: pure\ b \rightarrow (b \rightarrow Thread\ m\ a) \rightarrow Thread\ m\ a$. Here, $pure$ is a monad that characterizes pure expressions, providing the primitive $runPure :: pure\ b \rightarrow b$ to obtain the value denoted by the code given as argument. The monad $pure$ could be instantiated to Par, a monad that parallelizes pure computations in Haskell [Marlow et al., 2011], with $runPure$ set to $runPar$. In a resumption, $Parallel\ p\ f$ specifies that p is to be executed in a separate Haskell thread—potentially running on a different core than the interleaved computation. Once p produces a value x, f is applied to x to produce the next resumption to execute.

$$
\begin{array}{l}
\textbf{sch}\ (Parallel\ p\ f : thrds) = \\
\quad \textbf{do}\ res \leftarrow sync\ (\lambda v \rightarrow putMVar\ v\ (runPure\ p)) \\
\qquad\qquad\qquad\quad (\lambda v \rightarrow takeMVar\ v) \\
\qquad\qquad\qquad\quad f \\
\qquad \textbf{sch}\ (thrds +\!\!+ [res])
\end{array}
$$

Fig. 6. Scheduler for parallel computations

Figure 6 defines **sch** for pure computations, where interaction between resumptions and Haskell-threads gets regulated. The scheduler relies on well-established synchronization primitives called MVars [Jones et al., 1996]. A value of type $MVar$ is a mutable location that is either empty or contains a value. Function $putMVar$ fills the MVar with a value if it is empty and blocks otherwise. Dually, $takeMVar$ empties an MVar if it is full and returns the value; otherwise it blocks. Our scheduler implementation **sch** simply takes the resumption produced by the $sync$ function and schedules it at the end of the thread pool. Function $sync$, internally creates a fresh MVar v and spawns a new Haskell-thread to execute $putMVar\ v\ (runPure\ p)$. This action will store the result of the parallel computation in the provided MVar. Subsequently, $sync$ returns the resumption res, whose first atomic action is to read the parallel computation's result from the MVar ($takeMVar\ v$). At the time of reading, if a value is not yet ready, the atomic action will block the whole interleaved computation. However, once a value x is produced (in the separate thread), f is applied to it and the execution proceeds with the produced resumption ($f\ x$).

[4] In the case of Haskell, lazy evaluation may pose a challenge since whether or not a thunk has been evaluated is indeed an effect on a cache [Buiras and Russo, 2013]. Though our resumption-based approach handles this for the single-core case, handling this in general is part of our ongoing work.

$$\text{(Done)}\quad \frac{}{\langle \Sigma, \mathbf{sch}\ (\mathit{Done}\ x : t_s)\rangle \longrightarrow \langle \Sigma, \mathbf{sch}\ t_s\rangle}$$

$$\text{(Atom)}\quad \frac{\langle \Sigma[\texttt{lbl} \mapsto l], m\rangle \longrightarrow^{*} \langle \Sigma', (e)^{\texttt{LIO}}\rangle}{\langle \Sigma, \mathbf{sch}\ (\mathit{Atom}\ (\mathit{put}\ l \gg m) : t_s)\rangle \longrightarrow \langle \Sigma', \mathbf{sch}\ (t_s +\!\!+ [\mathit{put}\ \Sigma'.\texttt{lbl} \succ e])\rangle}$$

$$\text{(Fork)}\quad \frac{}{\langle \Sigma, \mathbf{sch}\ (\mathit{Fork}\ m_1\ m_2 : t_s)\rangle \longrightarrow \langle \Sigma, \mathbf{sch}\ ((m_1 : t_s) +\!\!+ [\mathit{put}\ \Sigma.\texttt{lbl} \succ m_2])\rangle}$$

Fig. 7. Semantics for **sch** expressions.

4.6 Soundness

In this section, we extend the previous formalization of LIO [Stefan et al., 2011] to model the semantics of our concurrency library. We present the syntax extensions that we require to model the behavior of the *Thread* monad:

Expression: $e ::= \ \ldots \mid \mathbf{sch}\, e_s \mid \mathit{Atom}\, e \mid \mathit{Done}\, e \mid \mathit{Fork}\, e\, e \mid \mathit{Parallel}\, e\, e$

where e_s is a list of expressions. For brevity, we omit a full presentation of the syntax and semantics, since we rely on previous results in order to prove the security property of our approach.

Expressions are the usual λ-calculus expressions with special syntax for monadic effects and LIO operations. The syntax node **sch** e_s denotes the scheduler running with the list of threads e_s as its thread pool. The nodes $\mathit{Atom}\ e$, $\mathit{Done}\ e$, $\mathit{Fork}\ e\ e$ and $\mathit{Parallel}\ e\ e$ correspond to the constructors of the *Thread* data type. In what follows, we will use metavariables x, m, p, t, v and f for different kinds of expressions, namely values, monadic computations, pure computations, threads, MVars and functions, respectively.

We consider a global environment Σ which contains the current label of the computation ($\Sigma.\texttt{lbl}$), and also represents the resources shared among all threads, such as mutable references. We start from the one-step reduction relation[5] $\langle \Sigma, e\rangle \longrightarrow \langle \Sigma', e'\rangle$, which has already been defined for LIO [Stefan et al., 2011]. This relation represents a single evaluation step from e to e', with Σ as the initial environment and Σ' as the final one. Presented as an extension to the $\longrightarrow$ relation, Figure 7 shows the reduction rules for concurrent execution using **sch**. The configurations for this relation are of the form $\langle \Sigma, \mathbf{sch}\ t_s\rangle$, where Σ is a runtime environment and t_s is a list of *Thread* computations. Note that the computation in an *Atom*

[5] As in [Stefan et al., 2012a], we consider a version of $\longrightarrow$ which does not include the operation *toLabeled*, since it is susceptible to internal timing attacks.

$$\text{(SEQ)}\quad \frac{\langle \Sigma, e\rangle \longrightarrow \langle \Sigma', e'\rangle \qquad P \Rightarrow P'}{\langle \Sigma, e \parallel P\rangle \hookrightarrow \langle \Sigma', e' \parallel P'\rangle}$$

$$\text{(PURE)}\quad \frac{P \Rightarrow P' \qquad v_s \text{ fresh MVar} \qquad s = \Sigma.\texttt{lbl}}{\begin{array}{c}\langle \Sigma, \mathbf{sch}\ (Parallel\ p\ f : t_s) \parallel P\rangle \hookrightarrow \\ \langle \Sigma, \mathbf{sch}\ (t_s +\!\!+ [Atom\ (takeMVar\ v_s \gg\!= f)]) \parallel P' \parallel (putMVar\ v_s\ (runPure\ p))_s\rangle\end{array}}$$

$$\text{(SYNC)}\quad \frac{P \Rightarrow P'}{\begin{array}{c}\langle \Sigma, \mathbf{sch}\ (Atom\ (takeMVar\ v_s \gg\!= f) : t_s) \parallel (putMVar\ v_s\ x)_s \parallel P\rangle \hookrightarrow \\ \langle \Sigma, \mathbf{sch}\ (f\ x : t_s) \parallel P'\rangle\end{array}}$$

Fig. 8. Semantics for **sch** expressions with parallel processes.

always begins with either $put\ l$ for some label l, or with $takeMVar\ v$ for some MVar v. Rules (DONE), (ATOM), and (FORK) basically behave like the corresponding equations in the definition of **sch** (see Figures 3 and 4). In rule (ATOM), the syntax node $(e)^{\text{LIO}}$ represents an LIO computation that has produced expression e as its result. Although **sch** applications should expand to their definitions, for brevity we show the unfolding of the resulting expressions into the next recursive call. This unfolding follows from repeated application of basic λ-calculus reductions.

Figure 8 extends relation $\longrightarrow$ into $\hookrightarrow$ to express pure parallel computations. The configurations for this relation are of the form $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle$, where P is an abstract process representing a pure computation that is performed in parallel. These abstract processes would be reified as native Haskell threads. The operator ($\parallel$), representing parallel process composition, is commutative and associative.

As described in the previous section, when a $Thread$ evaluates a $Parallel$ computation, a new native Haskell thread should be spawned in order to run it. Rule (PURE) captures this intuition. A fresh MVar v_s (where s is the current label) is used for synchronization between the parent and the spawned thread. A process is denoted by $putMVar\ v_s$ followed by a pure expression, and it is also tagged with the security level of the thread that spawned it.

Pure processes are evaluated in parallel with the main threads managed by **sch**. The relation $\Rightarrow$ nondeterministically evaluates one process in a parallel composition and is defined as follows.

$$\frac{runPure\ p \longrightarrow^{*} x}{(putMVar\ v_s\ (runPure\ p))_s \parallel P \Rightarrow (putMVar\ v_s\ x)_s \parallel P}$$

For simplicity, we consider the full evaluation of one process until it yields a value as just one step, since the computations involved are pure and therefore cannot leak data. Rule (SEQ) in Figure 8 represents steps where no parallel forking or synchronization is performed, so it executes one $\longrightarrow$ step alongside a $\Rightarrow$ step.

Rule (SYNC) models the synchronization barrier technique from Section 4.5. When an $Atom$ of the form $(takeMVar\ v_s \gg\!= f)$ is evaluated, execution blocks until the pure process with the corresponding MVar v_s completes its computation. After that, the process is removed and the scheduler resumes execution.

Security guarantees We show that programs written using our library satisfy termination-insensitive non-interference, i.e., an attacker at level L cannot distinguish the results of programs that run with indistinguishable inputs (see Appendix D for more details) . This result has been previously established for the sequential version of LIO [Stefan et al., 2011]. As in [Li and Zdancewic, 2010, Russo et al., 2008, Stefan et al., 2011], we prove this property by using the *term erasure* technique.

In this proof technique, we define function ε_L in such a way that $\varepsilon_L(e)$ contains only information below or equal to level L, i.e., the function ε_L replaces all the information more sensitive than L or incomparable to L in e with a hole ($\bullet$). We adapt the previous definition of ε_L to handle the new constructs in the library. In most of the cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon_L(e_1\ e_2) = \varepsilon_L(e_1)\ \varepsilon_L(e_2)$). For **sch** expressions, the erasure function is mapped into the list; all threads with a current label above L are removed from the pool ($filter\ (\not\equiv \bullet)\ (map\ \varepsilon_L\ t_s)$), where $\equiv$ denotes syntactic equivalence). Analogously, erasure for a parallel composition consists of removing all processes using an MVar tagged with a level not strictly below or equal to L. The computation performed in a certain $Atom$ is erased if the label is not strictly below or equal than L. This is given by

$$\varepsilon_L(Atom\ (put\ s \gg m)) = \begin{cases} \bullet & , \ s \not\sqsubseteq L \\ put\ s \gg \varepsilon_L\ (m) & , \ \text{otherwise} \end{cases}$$

A similar rule exists for expressions of the form $Atom\ (takeMVar\ v_s \gg\!= f)$. Note that this relies on the fact that an atom must be of the form $Atom\ (put\ s \gg m)$ or $Atom\ (takeMVar\ v_s \gg\!= f)$ by construction. For expressions of the form $Parallel\ p\ f$, erasure behaves homomorphically, i.e. $\varepsilon_L(Parallel\ p\ f) =$
$Parallel\ \varepsilon_L(p)\ (\varepsilon_L \circ f)$.

Following the definition of the erasure function, we introduce the evaluation relation $\hookrightarrow_L$ as follows: $\langle \Sigma, t \parallel P\rangle \hookrightarrow_L \varepsilon_L(\langle \Sigma', t' \parallel P'\rangle)$ if $\langle \Sigma, t \parallel P\rangle \hookrightarrow \langle \Sigma', t' \parallel P'\rangle$. The relation $\hookrightarrow_L$ guarantees that confidential data, i.e., data not below or equal-to level L, is erased as soon as it is created. We write $\hookrightarrow_L^*$ for the reflexive and transitive closure of $\hookrightarrow_L$.

In order to prove non-interference, we will establish a simulation relation between $\hookrightarrow^*$ and $\hookrightarrow_L^*$ through the erasure function: erasing all secret data and then taking evaluation steps in $\hookrightarrow_L$ is equivalent to taking steps in $\hookrightarrow$ first, and then erasing all secret values in the resulting configuration. In the rest of this section, we consider well-typed terms to avoid stuck configurations.

Proposition 1 (Many-step simulation). *If* $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle \hookrightarrow^* \langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle$*, then it holds that* $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle)$.

The L-equivalence relation $\approx_L$ is an equivalence relation between configurations and their parts, defined as the equivalence kernel of the erasure function ε_L: $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle \approx_L \langle \Sigma', \mathbf{sch}\ r_s \parallel Q\rangle$ iff $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle) = \varepsilon_L(\langle \Sigma', \mathbf{sch}\ r_s \parallel Q\rangle)$. If two configurations are L-equivalent, they agree on all data below or at level L, i.e., an attacker at level L is not able to distinguish them.

The next theorem shows the non-interference property. The configuration $\langle \Sigma, \mathbf{sch}\ []\rangle$ represents a final configuration, where the thread pool is empty and there are no more threads to run.

Theorem 1 (Termination-insensitive non-interference). *Given a computation* e*, inputs* e_1 *and* e_2*, an attacker at level* L*, runtime environments* Σ_1 *and* Σ_2*, then for all inputs* e_1, e_2 *such that* $e_1 \approx_L e_2$*, if* $\langle \Sigma_1, \mathbf{sch}\ [e\ e_1]\rangle \hookrightarrow^* \langle \Sigma'_1, \mathbf{sch}\ []\rangle$ *and* $\langle \Sigma_2, \mathbf{sch}\ [e\ e_2]\rangle \hookrightarrow^* \langle \Sigma'_2, \mathbf{sch}\ []\rangle$*, then* $\langle \Sigma'_1, \mathbf{sch}\ []\rangle \approx_L \langle \Sigma'_2, \mathbf{sch}\ []\rangle$.

This theorem essentially states that if we take two executions from configurations $\langle \Sigma_1, \mathbf{sch}\ [e\ e_1]\rangle$ and $\langle \Sigma_2, \mathbf{sch}\ [e\ e_2]\rangle$, which are indistinguishable to an attacker at level L ($e_1 \approx_L e_2$), then the final configurations for the executions $\langle \Sigma'_1, \mathbf{sch}\ []\rangle$ and $\langle \Sigma'_2, \mathbf{sch}\ []\rangle$ are also indistinguishable to the attacker ($\langle \Sigma'_1, \mathbf{sch}\ []\rangle \approx_L \langle \Sigma'_2, \mathbf{sch}\ []\rangle$). This result generalizes when constructors *Done*, *Atom*, and *Fork* involve exception handling (see Figure 5). The reason for this lies in the fact that *catch* and *throw* defer all exception handling to *LIO.throw* and *LIO.catch*, which have been proved secure in [Stefan et al., 2012b].

4.7 Case study: Classifying location data

We evaluated the trade-offs between performance, expressiveness and security through an LIO case study. We implemented an untrusted application that performs K-means clustering on sensitive user location data, in order to classify GPS-enabled cell phone into locations on a map, e.g., home, work, gym, etc. Importantly, this app is untrusted yet computes clusters for users without leaking their location (e.g., the fact that Alice frequents the

local chapter of the Rebel Alliance). K-means is a particularly interesting application for evaluating our scheduler as the classification phase is highly parallelizable—each data point can be evaluated independently.

We implemented and benchmarked three versions of this app: (i) A baseline implementation that does not use our scheduler and parallelizes the computation using Haskell's *Par* Monad [Marlow et al., 2011]. Since in this implementation, the scheduler is not modeled using resumptions, it leverages the parallelism features of *Par*. (ii) An implementation in the resumption based scheduler, but pinned to a single core (therefore not taking advantage of parallelizing pure computations). (iii) A parallel implementation using the resumption-based scheduler. This implementation expresses the exact same computation as the first one, but is not vulnerable to cache-based leaks, even in the face of parallel execution on multiple cores.

We ran each implementation against one month of randomly generated data, where data points are collected each minute (so, 43200 data points in total). All experiments were run ten times on a machine with two 4-core (with hyperthreading) 2.4Ghz Intel Xeon processors and 48GB of RAM. The secure, but non-parallel implementation using resumptions performed extremely poorly. With mean 204.55 seconds (standard deviation 7.19 seconds), it performed over eight times slower than the baseline at 17.17 seconds (standard deviation 1.16 seconds). This was expected since K-means is highly parallelizable. Conversely, the parallel implementation in the resumption based scheduler performed more comparably to the baseline, at 17.83 seconds (standard deviation 1.15 seconds).

To state any conclusive facts on the overhead introduce by our library, it is necessary to perform a more exhaustive analysis involving more than a single case study.

4.8 Related work

Cryptosystems Attacks exploiting the CPU cache have been considered by the cryptographic community [Kocher, 1996]. Our attacker model is weaker than the one typically considered in cryptosystems, i.e., attackers with access to a stopwatch. As a countermeasure, several authors propose partitioning the cache (e.g., [Page, 2005]), which often requires special hardware. Other countermeasures (e.g. [Osvik et al., 2006]) are mainly implementation-specific and, while applicable to cryptographic primitives, they do not easily generalize to arbitrary code (as required in our scenario).

Resumptions While CPS can be used to model concurrency in a functional setting [Claessen, 1999], resumptions are often simpler to reason about when considering security guarantees [Harrison, 2004, Harrison and Hook, 2005]. The closest related work is that of Harrison and Hook [Harrison

and Hook, 2005]; inspired by a secure multi-level operating system, the authors utilize resumptions to model interleaving and layered state monads to represent threads. Every layer corresponds to an individual thread, thereby providing a notion of local state. Since we do not require such generality, we simply adapt the scheduler to context-switch the local state underlying the *LIO* monad. We believe that authors overlooked the power of resumptions to deal with timing perturbations produced by the underlying hardware. In [Harrison, 2004], Harrison hints that resumptions could handle exceptions; in this work, we consummate his claim by describing precicely how to implement *throw* and *catch*.

Language-based IFC There has been a considerable amount of literature on applying programming languages techniques to address the internal timing covert channel (e.g. [Smith and Volpano, 1998, Volpano and Smith, 1999, Zdancewic and Myers, 2003, Russo and Sabelfeld, 2006a, Stefan et al., 2012a]). Many of these works assume that the execution of a single step, i.e., a reduction step in some transition system, is performed in a single unit of time. This assumption is often made so that security guarantees can be easily shown using programming language semantics. Unfortunately, the presence of the CPU cache (or other hardware shared state) breaks this correspondence, making cache attacks viable. Our resumption approach establishes a correspondence between atomic steps at the implementation-level and reduction step in a transition system. Previous approaches can leverage this technique when implementing systems, as to avoid the reappearance of the internal timing channel.

Agat [Agat, 2000] presents a code transformation for sequential programs such that both code paths of a branch have the same memory access pattern. This transformation has been adapted in different works (e.g., [Sabelfeld and Sands, 2000]). Agat's approach, however, focuses on avoiding attacks relying on the data cache, while leaving the instruction cache unattended.

Russo and Sabelfeld [Russo and Sabelfeld, 2006b] consider non-interference for concurrent while-like-programs under cooperative and deterministic scheduling. Similar to our work, this approach eliminates cache-attacks by restricting the use of yields. Differently, our library targets a richer programming languages, i.e., it supports parallelism, exceptions, and dynamically adjusting the granularity of atomic actions.

Secure multi-execution [Devriese and Piessens, 2010] preserves confidentiality of data by executing the same sequential program several times, one for each security level. In this scenario, cache-based attacks can only be removed in specific configurations [Kashyap et al., 2011] (e.g., when there are as many CPU cores as security levels).

Hedin and Sands [Hedin and Sands, 2005] present a type-system for preventing external timing attacks for bytecode. Their semantics is augmented to incorporate history, which enables the modeling of cache effects.

Zhang et al. [Zhang et al., 2012] provide a method for mitigating external events when their timing behavior could be affected by the underlying hardware. Their semantics focusses on sequential programs, wherein attacks due to the cache arise in the form of externally visible events. Their solution is directly applicable to our system when considering external events.

System security In order to achieve strong isolation, Barthe et al. [Barthe et al., 2012] present a model of virtualization which flushes the cache upon switching between guest operating systems. Flushing the cache in such scenarios is common and does not impact the already-costly context-switch. Although this technique addresses attacks that leverage the CPU cache, it does not address the case where a shared resource cannot be controlled (e.g., CPU bus).

Allowing some information leakage, Kopft et al. [Köpf et al., 2012] combines abstract interpretation and quantitative information-flow to analyze leakage bounds for cache attacks. Kim et al. [Kim et al., 2012] propose StealthMem, a system level protection against cache attacks. StealthMem allows programs to allocate memory that does not get evicted from the cache. StealthMem is capable of enforcing confidentiality for a stronger attacker model than ours, i.e., they consider programs with access to a stopwatch and running on multiple cores. However, we suspect that StealthMem is not adequate for scenarios with arbitrarily complex security lattices, wherein not flushing the cache would be overly restricting.

4.9 Conclusion

We present a library for LIO that leverages resumptions to expose concurrency. Our resumption-based approach and "instruction"- or atom-based scheduling removes internal timing leaks induced by timing perturbations of the underlying hardware. We extend the notion of resumptions to support state and exceptions and provide a scheduler which context-switches programs with such features. Though our approach eliminates internal-timing attacks that leverage hardware caches, library-level threading imposes considerable performance penalties. Addressing this, we provide programmers with a safe mean for controlling the context-switching frequency, i.e., allowing for the adjustment of the "size" of atomic actions. Moreover, we provide a primitive for spawning computations in parallel, a novel feature not previously available in IFC tools. We prove soundness of our approach and implement a simple case study to demonstrate its use. Our techniques can be adapted to other Haskell-like IFC systems beyond LIO. The library and case study can be found at [Buiras et al., 2013].

Acknowledgments We would like to thank Josef Svenningsson and our colleagues in the ProSec and Functional Programming group at Chalmers for useful comments. This work was supported by the Swedish research agency VR, STINT, the Barbro

Osher foundation, DARPA CRASH under contract #N66001-10-2-4088, and multiple gifts from Google. Deian Stefan is supported by the DoD through the NDSEG Fellowship Program.

BIBLIOGRAPHY

O. Aciiçmez. Yet another microarchitectural attack:: exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, CSAW '07. ACM, 2007.

J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Prog. Languages*, pages 40–53, Jan. 2000.

G. Barthe, G. Betarte, J. Campo, and C. Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, june 2012.

Boudol and Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*. Springer-Verlag, July 2001.

G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1), June 2002.

P. Buiras and A. Russo. Lazy programs leak secrets. In *the Pre-proceedings of the 18th Nordic Conference on Secure IT Systems (NordSec)*, October 2013.

P. Buiras, A. Levy, D. Stefan, A. Russo, and D. Mazières. A library for removing cache-based attacks in concurrent information flow systems: Extended version. `http://www.cse.chalmers.se/~buiras/resLIO.html`, 2013.

K. Claessen. A poor man's concurrency monad. *J. Funct. Program.*, May 1999.

D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, October 2012.

B. Harrison. Cheap (but functional) threads. *J. of Functional Programming*, 2004.

W. L. Harrison and J. Hook. Achieving information flow security through precise control of effects. In *Proc. IEEE Computer Sec. Foundations Workshop*. IEEE Computer Society, 2005.

D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. *Elec. Notes Theor. Comput. Sci.*, 141, 2005.

S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996.

V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE, 2011.

T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proc. of the USENIX Conference on Security Symposium*, Security'12. USENIX Association, 2012.

P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. of the 16th CRYPTO*. Springer-Verlag, 1996.

B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12. Springer-Verlag, 2012.

M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web Without Walls. In *6th ACM Workshop on Hot Topics in Networking (Hotnets)*, Atlanta, GA, November 2007.

B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

S. Marlow, R. Newton, and S. L. P. Jones. A monad for deterministic parallelism. In *Proc. ACM SIGPLAN Symposium on Haskell*, 2011.

E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, CT-RSA'06. Springer-Verlag, 2006.

D. Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint Archive*, 2005, 2005.

C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

F. Pottier. A simple view of type-secure information flow in the π-calculus. In *In Proc. of the 15th IEEE Computer Security Foundations Workshop*, 2002.

A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Sec. Foundations Workshop*, July 2006a.

A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (PSI)*, LNCS. Springer-Verlag, June 2006b.

A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Proc. of Asian Computing Science Conference*, LNCS. Springer-Verlag, Dec. 2006.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM Press, Sept. 2008.

A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Sec. Foundations Workshop*, July 2000.

G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Prog. Languages*, Jan. 1998.

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *The 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 201–213. ACM, September 2012a.

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012b.

D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proc. European Symp. on Research in Computer Security*, 2013.

W. Swierstra. *A Functional Specification of Effects*. PhD thesis, University of Nottingham, November 2008.

D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3), Nov. 1999.

W. H. Wong. Timing attacks on RSA: revealing your secrets through the fourth dimension. *Crossroads*, 11, May 2005.

S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Sec. Foundations Workshop*, June 2003.

D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proc. of PLDI*. ACM, 2012.

```
attack :: LMVar LH Int → LIORef LH [Int] → LIORef LH [Int] →
          Labeled LH Int → LIO LH Int
attack lmv lref href h
  = do b ← traverse lref
       when b (do    – Thread C
                  forkLIO (do x ← unlabel h
                              when (x ≡ 0) (do b ← traverse href
                                               when b (return ())))
                  threadDelay delay_C
                    – Thread A
                  forkLIO (do b ← traverse lref
                              when b (putLMVar lmv 1))
                    – Thread B
                  forkLIO (do threadDelay delay_B
                              putLMVar lmv 0)
                  return ()
       w ← takeLMVar lmv
       _ ← takeLMVar lmv
       return w
```

Fig. 9. Cache-attack that leaks one bit of a secret in *LIO*

A Cache-attack for *LIO*

```
guess hs = do let lL = [1 .. constant] :: [Int]
                  lH = [1 .. constant] :: [Int]
              lmv ← newEmptyLMVar L
              lref ← newLIORef L lL
              href ← newLIORef H lH
              mapM (attack lmv lref href) hs
```

Fig. 10. Magnification of the attack in Figure 9

Fig. 9 shows the cache-attack described in Section 4.2 for *LIO*. We assume the classic two-point lattice (of type *LH*) where security levels L and H denote public and secret data, respectively. Function *attack* takes a public shared *LMVar* (*lmv*), two references to lists of public (*lref*) and secret data (*href*), and a secret integer h. The goal of *attack* is to return a public integer equal to h. For simplicity, we use *threadDelay* n, which places a thread to sleep for n micro seconds, to exploit the race to *lmv*—using a loop would work equally well. In Fig. 9, parameter *delay_C* is set to

$$
\begin{aligned}
&return\ x = Done\ x\\
&Done\ x \gg\!= f = f\ x\\
&Atom\ m \gg\!= f = Atom\ (\mathbf{do}\ res \leftarrow m\\
&\qquad\qquad\qquad\qquad return\ (res \gg\!= f))\\
&Fork\ res\ res' \gg\!= f = Fork\ res\ (res' \gg\!= f)\\
&Parallel\ p\ g \gg\!= f = Parallel\ p\ (\lambda r \rightarrow g\ r \gg\!= f)
\end{aligned}
$$

Fig. 11. Defintions *return* and $\gg\!=$.

wait for thread C to finish running. Similarly, parameter $delay\_B$ imposes a delay on thread B before attempting to update lmv with 0. Variable w stores the first written value in lmv, which will coincide with the value of h.

Figure 10 shows the magnification of the attack for a list of secret integers (hs). Parameter *constant* determines the size of the lists with public and secret data, respectively. The magnification is simply to map function *attack* to the list of secrets. (The precise values of these parameters are machine-specific and experimentally determined.) Below we present the final component required for the attack:

$$
\begin{aligned}
&traverse\ ref =\\
&\quad \mathbf{do}\ ls \leftarrow readLIORef\ ref\\
&\qquad return\ ((ls \equiv [x \mid x \leftarrow ls]) \wedge (reverse\ (reverse\ ls) \equiv ls))
\end{aligned}
$$

B Monadic Operations for $(Thread\ m)$

Figure 11 shows the precise definition for *return* and $\gg\!=$. The interesting definitions are the ones related to bind. Computed values are represented by $Done$, so this is the only case when f is applied. The case for $Atom$ constructs a resumption consisting in the first atomic step in m $(res \leftarrow m)$ and returning a new resumption sequencing the subsequent atomic steps in m $(return\ (res \gg\!= f))$. In this case, the *do-notation* describes operations in the monad m (not $Thread\ m$). The definition of $Fork$ sequences the atomic actions found in the resumption res' $(res' \gg\!= f)$. Similarly, the case $Parallel\ p\ g$ sequences the atomic steps generated by $g\ r$ $(g\ r \gg\!= f)$, where r is the result of the spawned parallel computation.

C Granularity of Atomic Steps

Figure 12 shows the definition of function $singleAtom$. When applied, $singleAtom$ collapses the atomic steps found between constructors $Fork$ and $Parallel$. The cases for $Done$, $Fork$, and $Parallel$ are self-explanatory.

```
singleAtom :: Monad m ⇒ Thread m a → Thread m a
singleAtom (Done x)       = Done x
singleAtom (Atom m)       = Atom (m ≫= atomically)
  where
    atomically (Done x)        = return (Done x)
    atomically (Atom m')       = m' ≫= atomically
    atomically (Fork res res') = return (Fork res (singleAtom res'))
    atomically (Parallel p f)  = return (Parallel p (λr → singleAtom (f r)))
singleAtom (Fork res res') = Fork res (singleAtom res')
singleAtom (Parallel p f) = Parallel p (λr → singleAtom (f r))
```

Fig. 12. Collapsing atomic steps

The case for $Atom$ deserves some explanation. It only creates an $Atom$ ($Atom\ (m \gg\!= atomically)$), which first atomic step is performed by m, and the resulting resumption is given to the auxiliary function $atomically$. This function removes all the consecutive constructors $Atom$ ($atomically\ (Atom\ m') = m' \gg\!= atomically$).

D Soundness

We start by showing that the evaluation relations $\hookrightarrow$ and $\hookrightarrow_L$ are deterministic. Note that this is possible because we assume deterministic parallelism in our pure parallel computations. The following results rely on the previous determinacy results for sequential LIO.

Lemma 0 (Determinacy of $\hookrightarrow$). *If $\langle \Sigma, e\rangle \hookrightarrow \langle \Sigma', e'\rangle$ and $\langle \Sigma, e\rangle \hookrightarrow \langle \Sigma'', e''\rangle$, then $\langle \Sigma', e'\rangle = \langle \Sigma'', e''\rangle$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step.

Lemma 1 (Determinacy of $\hookrightarrow_L$). *If $\langle \Sigma, e\rangle \hookrightarrow_L \langle \Sigma', e'\rangle$ and $\langle \Sigma, e\rangle \hookrightarrow_L \langle \Sigma'', e''\rangle$, then $\langle \Sigma', e'\rangle = \langle \Sigma'', e''\rangle$.*

Proof. By Lemma 0 and definition of ε_L.

The following lemma establishes a simulation between $\hookrightarrow$ and $\hookrightarrow_L$ when reducing the body of a thread whose current label is below or equal to level L. In this result, we use the fact that the reduction $\longrightarrow$ from the original LIO formalization has been proved to have this property.

Lemma 2 (Single-step simulation for public computations).
If $\langle \Sigma, \mathbf{sch}\ (t : t_s) \parallel P\rangle \hookrightarrow \langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle$ with $\Sigma.\mathtt{lbl} \sqsubseteq L$, then $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ (t : t_s) \parallel P\rangle) \hookrightarrow_L \varepsilon_L(\langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle)$.

Proof. From previous results, we know that if m is a sequential LIO computation and $\langle \Sigma, m\rangle \longrightarrow \langle \Sigma', e\rangle$, then $\varepsilon_L(\langle \Sigma, m\rangle) \longrightarrow_L \varepsilon_L(\langle \Sigma', e\rangle)$.

- Case $t = Atom\ (put\ \Sigma.\texttt{lbl} \gg m)$:
  $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ (Atom\ (put\ l \gg m) : t_s) \parallel P\rangle)$
  $= \langle \varepsilon_L(\Sigma), \mathbf{sch}\ (Atom\ (put\ l \gg \varepsilon_L(m)) : \varepsilon_L(t_s)) \parallel \varepsilon_L(P)\rangle$
  $\hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \mathbf{sch}\ (\varepsilon_L(t_s) +\!\!+ [put\ \Sigma'.\texttt{lbl} \succ \varepsilon_L(e)]) \parallel \varepsilon_L(P')\rangle)$
  We know that $\varepsilon_L(\Sigma^1) = \varepsilon_L(\Sigma)$ from previous results, since LIO state transformations cannot introduce secrets observable by an attacker.
- Case $t = Parallel\ p\ f$:
  $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ (Parallel\ p\ f : t_s) \parallel P\rangle)$
  $= \langle \varepsilon_L(\Sigma), \mathbf{sch}\ (Parallel\ \varepsilon_L(p)\ (\varepsilon_L \circ f) : \varepsilon_L(t_s)) \parallel \varepsilon_L(P)\rangle$
  $\hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \mathbf{sch}\ (\varepsilon_L(t_s) +\!\!+ [Atom\ (readLMVar\ v_s \gg\!= \varepsilon_L \circ f)]) \parallel \varepsilon_L(P)$
  $\parallel putLMVar\ v_s\ (runPure\ \varepsilon_L(p))\rangle)$
  As before, we know that $\varepsilon_L(\Sigma^1) = \varepsilon_L(\Sigma)$, so the result follows directly from the properties of the erasure function.
- The other cases are similar.

We can also show that initial and final configurations for any reduction steps taken from a thread above L are equal when erased.

Lemma 3. *If* $\langle \Sigma, \mathbf{sch}\ (t : t_s) \parallel P\rangle \hookrightarrow \langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle$ *with* $\Sigma.\texttt{lbl} \not\sqsubseteq L$, *then*
$\varepsilon_L(\langle \Sigma, \mathbf{sch}\ (t : t_s) \parallel P\rangle) = \varepsilon_L(\langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle)$.

Proof. Since $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ (t : t_s) \parallel P\rangle) = \langle \varepsilon_L(\Sigma^1), \bullet\rangle$, we only have to show that $\varepsilon_L(\Sigma) = \varepsilon_L(\Sigma^1)$, where Σ^1 is the modified environment after performing the reduction step. The proof is similar to the corresponding lemma in the original version of LIO, since the possible environment modifications are the same.

We can now prove the many-step simulation lemma.

Proposition 1 (Many-step simulation). *If* $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle \hookrightarrow^*$ $\langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle$, *then it holds that*

$$\varepsilon_L(\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle).$$

Proof. The proof is by induction on the derivation of $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle \hookrightarrow^*$ $\langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle$. We consider a thread queue of the form $r : r_s$, and suppose that $\langle \Sigma, \mathbf{sch}\ (e : r_s) \parallel P\rangle \hookrightarrow \langle \Sigma^1, r'_s\rangle$ and $\langle \Sigma^1, r'_s\rangle \hookrightarrow^* \langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle$ (otherwise the reduction is not making any progress, and the result is trivial).

- If $\Sigma.\texttt{lbl} \sqsubseteq L$, the result follows by Lemma 2 and the induction hypothesis.
- If $\Sigma.\texttt{lbl} \not\sqsubseteq L$, the result follows by Lemma 3 and the induction hypothesis.

Finally, we prove the non-interference result, showing that two terminating runs that start with L-equivalent configurations must end in L-equivalent configurations.

Theorem 1 (Termination-insensitive non-interference). *Given a computation e, inputs e_1 and e_2, an attacker at level L, runtime environments Σ_1 and Σ_2, then for all inputs e_1, e_2 such that $e_1 \approx_L e_2$, if $\langle \Sigma_1, \mathbf{sch}\ [e\ e_1] \rangle \hookrightarrow^* \langle \Sigma_1', \mathbf{sch}\ [] \rangle$ and $\langle \Sigma_2, \mathbf{sch}\ [e\ e_2] \rangle \hookrightarrow^* \langle \Sigma_2', \mathbf{sch}\ [] \rangle$, then $\langle \Sigma_1', \mathbf{sch}\ [] \rangle \approx_L \langle \Sigma_2', \mathbf{sch}\ [] \rangle$.*

Proof. By definition of $\approx_L$, we have

$$\varepsilon_L(\langle \Sigma_1, \mathbf{sch}\ [e\ e_1] \rangle) = \varepsilon_L(\langle \Sigma_2, \mathbf{sch}\ [e\ e_2] \rangle).$$

Then, by the simulation lemma (Proposition 1), we have

$$\varepsilon_L(\langle \Sigma_1, \mathbf{sch}\ [e\ e_1] \rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_1', \mathbf{sch}\ [] \rangle)$$
$$\varepsilon_L(\langle \Sigma_2, \mathbf{sch}\ [e\ e_2] \rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_2', \mathbf{sch}\ [] \rangle)$$

Moreover, from the determinacy of $\hookrightarrow_L^*$ given in Lemma 1, the end configurations must be the same, *i.e.* $\varepsilon_L(\langle \Sigma_1', \mathbf{sch}\ [] \rangle) = \varepsilon_L(\langle \Sigma_2', \mathbf{sch}\ [] \rangle)$. Finally, by definition of $\approx_L$, we conclude $\langle \Sigma_1', \mathbf{sch}\ [] \rangle \approx_L \langle \Sigma_2', \mathbf{sch}\ [] \rangle$.

CHAPTER
FIVE

LAZY PROGRAMS LEAK SECRETS

Pablo Buiras, Alejandro Russo

Abstract. To preserve confidentiality, information-flow control restricts how untrusted code handles secret data. While promising, IFC systems are not perfect; they can still leak sensitive information via covert channels. In this work, we describe a novel exploit of *lazy evaluation* to reveal secrets in IFC systems. Specifically, we show that lazy evaluation might transport information through the *internal timing covert channel*, a channel present in systems with concurrency and shared resources. We illustrate our claim with an attack for LIO, a concurrent IFC system for Haskell. We propose a countermeasure based on restricting the implicit sharing caused by lazy evaluation.

5.1 Introduction

Information-flow control (IFC) permits untrusted code to safely operate on secret data. By tracking how data is disseminated inside programs, IFC can avoid leaking secrets into public channels—a policy known as non-interference [Goguen and Meseguer, 1982]. Despite being promising, IFC systems are not flawless; the presence of covert channels allows attackers to still leak sensitive information.

Covert channels arise when programming language features are misused to leak information [Lampson, 1973]. The tolerance to such channels is determined by their bandwidth and how easy it is to exploit them. For instance, the termination covert channel, which exploits divergence of programs, has a different bandwidth in systems with intermediate outputs than in batch processes [Askarov et al., 2008].

Lazy evaluation is the default evaluation strategy of the purely functional programming language Haskell. This evaluation strategy has two distinctive features which can be used together to reveal secrets. Firstly, since it is a form of *non-strict evaluation*, it delays the evaluation of function/constructor arguments and let-bound identifiers until their denoted values are needed. Secondly, when the evaluation of such expressions is required, their resulting value is stored (cached) for subsequent uses of the same expression, a feature known as *sharing* or *memoisation*. This is known as *call-by-need* semantics or simply lazy evaluation. In Haskell, a *thunk*, also known as a delayed computation, is a parameterless closure created to prevent the evaluation of an expression until it is required at a later time. The process of evaluating a thunk is known as *forcing*. While lazy evaluation does not affect the denotation of expressions with respect to non-strict semantics, it affects the timing behaviour of programs. For instance, if a function argument is used more than once in the body of a function, it is almost always faster to use lazy evaluation as opposed to call-by-name, since it avoids re-evaluating every occurrence of the argument.

From a security point of view, it is unclear what type of semantics (non-strict versus strict) is desirable in order to deal with covert channels. In sequential settings, Sabelfeld and Sands [Sabelfeld and Sands, 2001] suggest that a non-strict semantics might be intrinsically safer than a strict one. This observation is based on the ability to exploit the *termination covert channel*. Although it could avoid termination leaks, lazy evaluation can compromise security in other ways. For instance, Rafnsson et al. [Rafnsson et al., 2013] describe how to exploit the Java (lazy) class initialisation process to reveal secrets. Not surprinsingly, lazy evaluation might also reveal secrets through the *external timing covert channel*. This channel involves externally measuring the time used to complete operations that may depend on secret data.

More interestingly, and totally unexplored until this work, lazy evaluation might transport information through the *internal timing covert*

channel. This covert channel arises by the mere presence of concurrency and shared resources. Malicious code can exploit it by setting up threads to race for a public shared resource and, depending on the secret, affecting their timing behaviour to determine the winner. With lazy evaluation in place, thunks become shared resources and forcing their evaluation corresponds to affecting the threads' timing behaviour—subsequent evaluations of previously forced thunks take practically no time.

We present an attack for *LIO* [Stefan et al., 2012], a concurrent IFC system for Haskell, that leverages lazy evaluation to leak secrets. *LIO* presents countermeasures for internal timing leaks based on programming language level abstractions. Since *LIO* is embedded in Haskell as a library, lazy evaluation, as a feature that primarily affects pure values, is handled by the host language. Lazy evaluation is essentially built into Haskell's internals, hence there are no programming language-level mechanisms for inspecting or creating thunks that could be used to implement a countermeasure. Thunks for pure values are transparently injected into *LIO* computations, so the library could not be capable of explicitly considering whether they have been memoised at any given time.

This paper is organised as follows. Section 5.2 briefly recaps the basics of *LIO*. Section 5.3 presents the attack. Section 5.4 describes a possible countermeasure. Conclusions are drawn in Section 5.5.

5.2 *LIO*: a concurrent IFC system for Haskell

In purely functional languages, computations with side-effects are encoded as values of abstract data types called monads [Moggi, 1991]. In Haskell, there are monads for performing inputs and outputs (monad *IO*), handling errors (monad *Error*), etc. The IFC system *LIO* is simply another monad in which security checks are performed before side-effects are performed.

The *LIO* monad keeps track of a *current label*. This label is an upper bound on the labels of all data in lexical scope. When a computation C, with current label L_C, observes an object labelled L_O, C's label is raised to the least upper bound or *join* of the two labels, written $L_C \sqcup L_O$. Importantly, the current label governs where the current computation can write, what labels may be used when creating new channels or threads, etc. For example, after reading an object O, the computation should not be able to write to a channel K if L_O is more confidential than L_K—this would potentially leak sensitive information (about O) into a less sensitive channel.

Since the current label protects all the variables in scope, in practical programs we need a way of manipulating differently-labelled data without monotonically increasing the current label. For this purpose, *LIO* provides explicit references to labelled, immutable data through a parametric data type called *Labeled*. A locally accessible symbol can bind, for example,

```
attack :: LMVar LH Int → Labeled LH Int → LIO LH Int
attack lmv secret
  = do let thunk = [1 .. constant] :: [Int]
         – Thread C
       forkLIO (do s ← unlabel secret
                   when (s ≢ 0) (do n ← traverse thunk
                                    when (n > 0) (return ())))
       threadDelay delay_C
         – Thread A
       forkLIO (do n ← traverse thunk
                   when (n > 0) (putLMVar lmv 1))
         – Thread B
       forkLIO (do threadDelay delay_B
                   putLMVar lmv 0)
       w ← takeLMVar lmv
       _ ← takeLMVar lmv
       return w
```

Fig. 1: Attack exploiting lazy evaluation

a value of type *Labeled l Int* (for some label type *l*), which contains an *Int* protected by a label different from the current one. Function *unlabel* :: *Labeled l a* → *a*[1] brings the labelled value into the current lexical scope and updates the current label accordingly.

LIO also includes IFC-aware versions of well-established synchronisation primitives known as *MVar*s [Jones et al., 1996]. A value of type *LMVar* is a mutable location that is either empty or contains a value. Function *putLMVar*
fills the *LMVar* with a value if it is empty and blocks otherwise. Dually, *readLMVar* empties an *LMVar* if it is full and blocks otherwise.

5.3 A lazy attack for *LIO*

Figure 1 shows the attack for *LIO*. The code essentially implements an internal timing attack [Smith and Volpano, 1998] which leverages lazy evaluation to affect the timing behaviour of threads. We assume the classic two-point lattice (of type *LH*) where security levels *L* and *H* denote public and secret data, respectively, and the only disallowed flow is the one from *H* to *L*. Function *attack* takes a public, shared *LMVar lmv*, and a labelled boolean *secret* (encoded as an integer for simplicity). The goal of *attack* is

[1] Symbol :: introduces type declarations and → denotes function types.

to return a public integer equal to *secret*, thus exposing an *LIO* vulnerability. In isolation, all the threads are secure. When executed concurrently, however, *secret* gets leaked into *lmv*. For simplicity, we use *threadDelay* n, which causes a thread to sleep for n micro seconds, to exploit the race to *lmv*—if such an operation was not allowed, using a loop would work equally well.

The attack proceeds as follows. Threads A and B do not start running until thread C finishes. This effect can be easily achieved by adjusting the parameter $delay\_C$. The role of thread C is to force the evaluation of the list *thunk* when the value of *secret* is not zero ($s \not\equiv 0$). To that end, function *traverse* goes over *thunk*, returning one of its elements. Condition $n > 0$ always holds and it is only used to force Haskell to fully evaluate the closure returned by *traverse*. Threads A and B will eventually start racing. Thread A executes the command $traverse\ thunk$ before writing the constant 1 into *lmv* ($putLMVar\ lmv\ 1$). Thread B delays writing 0 into *lmv* ($putLMVar\ lmv\ 0$) by some (carefully chosen) time $delay\_B$. If $s \not\equiv 0$, *thunk* will have already been evaluated when thread A traverses its elements, thus taking less time than thread B's delay. As a result, value 1 is first written into *lmv*. Otherwise, thread B's delay is shorter than the time taken by thread A to force the evaluation of *thunk*. In this case, value 0 is first written into *lmv*. Variable w observes the first written value in *lmv*, which will coincide with the value of the secret. The precise values of parameters *constant*, $delay\_C$, and $delay\_B$ are machine-specific and experimentally determined.

The following code shows the magnification of the attack for a list of secret integers.

```
magnify :: [Labeled LH Int] → LIO LH [Int]
magnify ss = do lmv ← newEmptyLMVar L
                mapM (attack lmv) ss
```

Function *magnify* takes a list of secret values ss (of type $[Labeled\ LH\ Int]$). The magnification proceeds by creating the public $LMVar$ ($newEmptyLMVar\ L$) needed by the attack. Function $mapM$ sequentially applies function $attack\ lmv$ (i.e. the attack) to every element in ss and collects the results in a public list ($[Int]$).

Below, we present the final component required for the attack:

```
traverse :: [a] → LIO LH a
traverse xs = return (last xs)
```

This function simply returns the last element of the list given as argument.

The code for the attack can be downloaded from `http://www.cse.chalmers.se/~buiras/LazyAttack.tar.gz`.

5.4 Restricting sharing

We propose a countermeasure based on restricting the sharing feature of lazy evaluation. Specifically, we propose duplicating shared thunks when spawning new threads. In that manner, sharing gets restricted to the lexical scope of each thread. Thunks being forced in one thread will then not affect the timing behaviour of the others. To illustrate this point, consider the shared *thunk* from Figure 1. If this countermeasure was implemented, forcing the evaluation of *thunk* by thread C would not affect the time taken by thread A to evaluate $traverse\ thunk$, making the attack no longer possible. An important drawback of this approach is that there would be a performance penalty incurred by disabling sharing among threads. Benchmarking and evaluation would be necessary to determine the full extent of the overhead inherent in the technique. Presumably, programmers could restructure their programs to minimise the effect of this penalty.

As an optimisation, it is possible to only duplicate thunks denoting pure expressions. Thunks denoting side-effecting expressions can be shared across threads without jeopardising security. The reason for that relies on *LIO*'s ability to monitor side-effects. If a thread that depends on the secret forces the evaluation of side-effecting computations, the resulting side-effects are required to agree with the IFC policy. For instance, threads with secrets in lexical scope can only force thunks that perform no public side-effects; otherwise *LIO* will abort the execution in order to preserve confidentiality.

To implement our approach, we propose using **deepDup**, an operation introduced by Joachim Breitner [Breitner, 2012] to prevent sharing in Haskell. Essentially, **deepDup** takes a variable as its argument and creates a private copy of the whole heap reachable from it, effectively duplicating the argument thunk and disabling sharing between it and the original thunk. In his paper, Breitner shows how to extend Launchbury's natural semantics for lazy evaluation [Launchbury, 1993] with **deepDup**. The natural semantics is given by a relation $\Gamma : t \Downarrow \Delta : v$, which represents the fact that from the heap Γ we can reduce term t to the value v, producing a new heap Δ. It is the relation between Γ and Δ which captures heap modifications caused by memoisation. In this setting, the rule for **deepDup** is

$$\frac{\Gamma, x \mapsto e, x' \mapsto \hat{e}[y_1'/y_1 \ldots, y_n'/y_n], (y_i' \mapsto \mathbf{deepDup}\ y_i)_{i \in 1 \ldots n} : x' \Downarrow \Delta : z \qquad \mathrm{ufv}(e) = \{y_1, \ldots, y_n\} \qquad x', y_1', \ldots, y_n' \text{ fresh}}{\Gamma, x \mapsto e : \mathbf{deepDup}\ x \Downarrow \Delta : z}$$

where $\mathrm{ufv}(e)$ is the set of unguarded[2] free variables of e and $\hat{e}$ is e with all bound variables renamed to fresh variables in order to avoid variable

[2] Function $\mathrm{ufv}(e)$ is defined as the set of free variables that are not already marked for duplication, i.e. $\mathrm{ufv}(\mathbf{deepDup}\ x) = \emptyset$, and in the rest of the cases it is inductively defined as usual.

capture when applying substitutions. Note that **deepDup** x duplicates all the thunks reachable from x in a lazy manner: the free variables $y_1, \ldots, y_n$ are replaced with calls to **deepDup** for each variable, so these duplications will not be performed until those variables are actually evaluated. Laziness is necessary to properly handle cyclic data structures, since the duplication process would loop indefinitely if it were to eagerly copy all thunks for such structures. As explained below, this design decision has important consequences for security.

In practice, we would use this primitive every time we fork a new thread: we take the body of the new thread m_1 and the body of the parent thread m_2, and replace them with **deepDup** m_1 and **deepDup** m_2. Due to the lazy nature of the duplication performed by **deepDup**, it is necessary to duplicate both thunks, i.e., m_1 and m_2. Consider two threads A and B with current labels L and H, respectively, and suppose that they both have a pointer to a certain thunk x in the same scope. If we only duplicated the thunk in A (the public thread), thread B could evaluate parts of x depending on the secret, before they have been duplicated in thread A—recall that **deepDup** is lazy. This would cause the evaluation of the same parts of the duplicated version of x in A to go faster, thus conveying some information about the secret to thread A. In addition, note that it is not possible to determine in advance—at the time *forkLIO* is called—which thread will raise its current label to H. Therefore, we must take care to duplicate all further references to shared thunks every time a fork occurs.

As a possible optimisation, we advise designing a data dependency analysis capable of over-approximating which expressions are shared among threads. Once the list of expressions (and their scope) has been calculated, we would proceed to instrument the code, introducing instructions that duplicate only the truly shared thunks at runtime, as opposed to duplicating every pure thunk in the body of each thread. We believe that HERMIT [Farmer et al., 2012] is an appropriate tool to deploy such instrumentation as a code-to-code transformation.

5.5 Conclusions

We describe and implement a new way of leveraging lazy evaluation to leak secrets in *LIO*, a concurrent IFC system in Haskell. Beyond *LIO*, the attack points out a subtlety of IFC for programming languages with lazy semantics and concurrency. We propose a countermeasure based on duplicating thunks at the time of forking in order to restrict sharing among threads. For that, we propose to use the experimental Haskell package *ghc-dup*. This package provides operations that copy thunks in a lazy manner. Although convenient for preserving program semantics, such design decision has implications for security. To deal with that, our solution requires duplicating thunks for both the newly spawned thread and its parent. As future work,

we will implement the proposed countermeasure, prove soundness (non-interference), evaluate its applicability through different case studies, and introduce some optimisations to reduce the amount of duplicated thunks.

Acknowledgments We would like to thank Andrei Sabelfeld, David Sands, and the anonymous reviewers for useful comments. Special thanks to Edward Z. Yang, who mentioned the work by Joachim Breitner to us. This work was funded by the Swedish research agency VR, STINT, and the Barbro Osher foundation.

BIBLIOGRAPHY

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. of the European Symp. on Research in Computer Security (ESORICS)*. Springer-Verlag, 2008.

Joachim Breitner. dup – Explicit un-sharing in Haskell. *CoRR*, abs/1207.2017, 2012.

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs. In *Proc. ACM SIGPLAN Symposium on Haskell*, 2012.

Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

Simon P. Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proc. ACM Symp. on Principles of Prog. Languages*. ACM, 1996.

Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

J. Launchbury. A natural semantics for lazy evaluation. In *Proc. ACM Symp. on Principles of Prog. Languages*. ACM, 1993.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Willard Rafnsson, Keiko Nakata, and Andrei Sabelfeld. Securing class initialization in Java-like languages. *IEEE Transactions on Dependable and Secure Computing*, 10(1), January 2013.

Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher Order Symbol. Comput.*, 14(1), March 2001.

G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Prog. Languages*, January 1998.

Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. Addressing covert termination and timing channels

in concurrent information flow systems. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2012.

Part II

Policy facets

CHAPTER

SIX

DYNAMIC ENFORCEMENT OF DYNAMIC POLICIES

Pablo Buiras, Bart van Delft

Abstract. This paper presents SLIO, an information-flow control mechanism enforcing *dynamic policies*: security policies which change the relation between security levels while the system is running. SLIO builds on LIO, a floating-label information-flow control system embedded in Haskell that uses a runtime monitor to enforce security. We identify an implicit flow arising from the decision to change the policy based on sensitive information and introduce a corresponding check in the enforcement mechanism. We provide a formal security guarantee for SLIO, presented as a knowledge-based property, which specifies that observers can only learn information in accordance with the level ordering. Like LIO, SLIO is a generic enforcement mechanism, parametrised on the concrete instantiation of security labels and their policy change mechanism. To illustrate the applicability of our results, we implement well-known label models such as DLM, the Flowlocks framework, and DC labels in SLIO.

(a) Before. (b) After.

Fig. 1: Company's hierarchy before and after Alice leaves.

6.1 Introduction

Many computing systems, such as personal computers, mobile phones and web pages, allow for the installation or inclusion of third-party code. This introduces the risk that untrusted code, either by intention or programming errors, leaks confidential information or violates the integrity of data. Information-flow control (IFC) mechanisms aim to ensure that the information flows in a system abide by the desired security policy.

As a running example we consider an application responsible for combining and sharing employee files, where each file is labeled with the employee's name as security level. The application requires read and write access to all files, but the security policy dictates that information may only flow in accordance to the company's hierarchy. That is, an employee's file can only contain information from that employee, whereas information from files of all employees in a division may be written to the files of the divison manager. [1]

Many IFC mechanisms assume a *static* partial ordering ($\sqsubseteq$) between security levels (e.g. [Simonet, 2003, Li and Zdancewic, 2006, Russo et al., 2008]). Figure 1a shows such an ordering for a company where Alice is the chief executive, Bob and Carl are managers of two different divisions, and Dave is an employee in both divisions. A commonly enforced security poperty is the notion of *noninterference* [Goguen and Meseguer, 1982]: sensitive inputs may not influence outputs to less-sensitive locations. However, some mechanisms allow for occasional exceptions to this ordering in the form of declassification [Sabelfeld and Sands, 2005]. For example, a file of Alice's security level may be sent to the division managers in redacted form.

In this paper, we argue that the ordering between security levels can be much more *dynamic*, as others have argued before [Askarov and Chong, 2012, Broberg and Sands, 2010, Matos and Boudol, 2005]. That is, rather

[1] One could argue that this particular situation could instead be modelled using roles for manager, employee etc. rather than employee names. We use this model for the sake of simplicity in our examples.

than an occasional exception to the static ordering, the ordering might change much more drastically and permanently. As an example, we consider that Alice has accepted a position in a different company and leaves. Consequently, Bob and Dave are promoted and Alice's documents have become isolated from the rest of the company, as show in Figure 1b. To enforce such dynamic policies, we need mechanisms that can account for the addition, as well as the removal of allowed information flows *during* program execution.

Similar changes in the information flow policy might occur in subscription-based services (such as music streaming) where information is only available for the time that the user paid for. Other examples are applications where users can change the policy themselves (e.g. in a smart phone system), and situations where the inherent value of information changes over time (e.g. revealing cards at the end of a poker game [Askarov and Sabelfeld, 2005]). These examples show that it is natural for the value of information to change over time, motivating the need to support dynamic information-flow policies.

Enforcing dynamic policies brings new challenges when compared to IFC for static policies.

- Under the assumption of the static nature of a partial ordering it possible to approximate the security level of information. For example, a file containing information of both Bob and Carl could, in Figure 1a, be approximated with the security level Alice. Such approximations are incorrect when we need to account for the possibility that the ordering might change.
- With static orderings, IFC typically enforces noninterference [Cohen, 1977, Goguen and Meseguer, 1982]. In the dynamic setting, the ordering between levels might be different at the moment of (each) output, so a different security condition is necessary.
- As the decision to modify the label ordering might be affected by sensitive inputs to the system, this creates a new potential flow of information that needs to be controlled.

This paper investigates the field of dynamic policies on both a theoretical and practical level in the context of dynamic enforcement, i.e. enforcement that checks information flows at runtime. As a starting point we use LIO [Stefan et al., 2011], a dynamic information-flow control library for Haskell implemented as an embedded language augmented with runtime security checks. In its most general form, LIO is parametric on the policy specification language and label ordering to be used in the program; this ordering must be defined in advance and cannot change at runtime. The library provides a noninterference guarantee with respect to this ordering. LIO's genericity makes it a suitable framework for our exploration of dynamic policies and for implementing our results.

Contributions Concretely, our contributions are:

- We present a new enforcement mechanism for dynamic policies, called SLIO. SLIO is a strict generalisation of LIO which, by providing a modifiable state component, allows for the generic enforcement of dynamic policies. We preserve LIO's genericity and abstract away from a concrete choice for both policy specification language and ordering change mechanism. (§ 6.3)
- We identify a clear constraint that needs to be checked for each ordering change to prevent information flow leaks via sensitive modifications to the label ordering. (§ 6.4)
- The original noninterference guarantee from LIO does not generalise to a setting with dynamic policies. For this purpose we introduce a new knowledge-based security condition. Though based on existing work ([Askarov and Chong, 2012]), we introduce a novel extension to allow for the persistent relabeling (or: declassification) of information. (§ 6.5, § 6.6)
- To demonstrate the practicality of SLIO, we implement multiple instances for various policy specification languages, including the Decentralized Label Model (DLM) [Myers and Liskov, 1998], Disjunction Category Labels [Stefan et al., 2012b] and Paralocks [Broberg and Sands, 2010]. (§ 6.7)

Before turning to the details of these contributions, we summarise the essentials of LIO.

6.2 LIO

Labeled IO, or LIO [Stefan et al., 2011], is a Haskell library that dynamically enforces information flow control, providing *termination-insensitive* noninterference [Askarov et al., 2008] for sequential programs[2].

LIO leverages Haskell's monadic encapsulation of side-effects to provide security. A *monad* [Moggi, 1989] is an abstract data type that can be used to structure effectful computations in purely functional languages. Monads specify how to build and bind computations together in sequence, and typically provide distinguished operations that model specific side-effects. Different monads are often used to model different kinds of effectful computations. In Haskell, monads are used to express all effects, including exception handling, nondeterminism, and input/output.

LIO leverages monads to precisely control what (side-effecting) operations the programmer is allowed to perform at any given time. An LIO program is a computation in the *LIO* monad, composed from simpler

[2] A concurrent version of LIO exists [Stefan et al., 2012a], but this work is largely orthogonal to the work presented here.

monadic terms using the fundamental monadic combinators **return** and $(\gg\!=)$ (read as "bind").

The operation **return** x produces a computation which returns the value denoted by x. Function $(\gg\!=)$ is used to sequence LIO computations. Specifically, $t \gg\!= (\lambda x.t')$ takes the result produced by t and applies function $\lambda x.t'$ to it. (This operator allows computation t' to depend on the value produced by t.) We sometimes use Haskell's **do**-notation to write such monadic computations. For example, the program $t \gg\!= \lambda x.\mathbf{return}\ (x+1)$, which adds 1 to the value produced by t, can be written as shown below.

```
do x ← t
   return (x + 1)
```

In Haskell, input/output operations are provided by the *IO* monad. That is, all computations that want to perform I/O operations have to be of the type $IO\ a$, where a is the type of the returned value of the computation. The *LIO* monad provided by the LIO library is intended to be used as a replacement for this type. It provides a collection of operations similar to *IO*, but enriched with security checks that prevent unwanted information flows. The noninterference guarantees provided by LIO only hold *within* this monad: an attacker is a piece of untrusted, potentially malicious LIO code that is run in the same context as the trusted code.

The LIO library employs the *floating-label* approach to dynamic information-flow control, which borrows ideas from the operating systems security research community [Zeldovich et al., 2006] and brings them into the field of language-based security. Assuming a security lattice of labels (with operations $\sqcup$, $\sqcap$ and $\sqsubseteq$ defined on them), the *LIO* monad uses its state to keep track of a *current label*, l_{cur}. This label represents the least upper bound over the labels on which the current computation depends. All the (I/O) operations provided by LIO take care to appropriately validate and adjust this label.

For example, when an LIO computation with current label l_{cur} observes an entity with label l_A, its current label must change (possibly rise) to the least upper bound of the two labels, written $l_{\text{cur}} \sqcup l_A$. As it were, the current label *floats up* in the security lattice, to maintain its position as an upper bound on the security levels of the information that is currently in scope. Similarly, before performing a side-effect visible to label l, LIO first checks that the current label flows to l ($l_{\text{cur}} \sqsubseteq l$) before allowing the operation to take place.

LIO is parameterised by a *label format*, i.e. the type of the labels is not fixed. Instead, actions work on a generic label type. The library leverages Haskell *type classes*, an overloading mechanism, to implement this: LIO actions have the context $(Label\ \alpha) \Rightarrow$ in their type signature, which restricts α to types that are instances of the *Label* type class. This class requires the label type to implement security lattice operations such as $\sqsubseteq$, $\sqcup$, and $\sqcap$,

making the behaviour of these operators depend on the particular label type of the labels to which they are applied.

Labeled values Since LIO protects all values in scope with l_{cur}, the library also provides a way to manipulate differently-labeled data without monotonically increasing the current label, l_{cur}. For this purpose, there is a data type called *Labeled*, which represents explicit references to labeled, immutable data. It is still possible to bind a variable of, say, type *Labeled* L *Int*, which contains an *Int* protected by a label (of type L) different from l_{cur}.

The two most important functions that work on labeled values are **label** and **unlabel**. The action **label** $l\ v$ creates a *Labeled* value with label l and contents v, provided that $l_{\text{cur}} \sqsubseteq l$. That is, the label l has to reflect that the value v can depend on data of label l_{cur}. Dually, the action **unlabel** lv raises l_{cur} to $l_{\text{cur}} \sqcup l$, where l is the label on lv, and returns v.

Labeled references LIO also provides mutable state in the form of references. In Haskell, references are of type *IORef* a, where a is the type of the contents of the reference. LIO introduces *labeled references*, typed *LIORef* L a, where L is the type of the labels and a is the type of the contents of the reference. The primitive operations on *LIORef*s are **newLIORef**, **readLIORef** and **writeLIORef**.

The action **newLIORef** $l\ v$ creates an *LIORef* with label l and contents v, provided that $l_{\text{cur}} \sqsubseteq l$. Given an *LIORef* r with label lr, the action **readLIORef** r returns the value of r, and raises l_{cur} to $l_{\text{cur}} \sqcup lr$. The action **writeLIORef** $r\ v$ replaces the contents of r with v, provided that $l_{\text{cur}} \sqsubseteq lr$. Note that the operations on *LIORef*s and *Labeled* values interact with the current label in analogous ways.

In floating-label systems, any computation on sensitive data raises the current label, even though the result of the computation may never be observable on a lower security level. As a result, it is possible for programs to inadvertently raise their current label to a point where they can no longer perform any useful side-effects, a situation known as *label creep*. For example, after a program reads from a file of a high security level, it is no longer allowed to write to a file of a lower security level, even when the information written does not depend in any way on the information read.

In order to mitigate this label creep, LIO provides the **toLabeled** operation. Given a computation m that would raise l_{cur} to l'_{cur}, **toLabeled** $l\ m$ executes m without raising l_{cur}, and instead encapsulates m's result in a *Labeled* value protected by label l – provided that $l'_{\text{cur}} \sqsubseteq l$. This allows for sub-computations that work on data above l_{cur} without causing the main computation to raise its current label.

Example Figure 2 shows an LIO program working with the lattice given in Figure 1a, of type *User*. The code defines a function *report*, which takes three labeled values as arguments. Information from these values is written

into the files of certain principals, the *LIORef*s *aliceReport* and *bobReport*. These files are assumed to be in scope, with labels Alice and Bob respectively. Firstly, Bob's data is unlabeled with **unlabel**, which raises the current label to Bob and binds b to the contents of the labeled value. Then, a **toLabeled** computation is started, which unlabels data from Alice (raising the current label to Alice), binding it to a. Depending on the value of b, the block returns either $a + b$ or just a, which is bound to the main code block as a labeled value with label Alice. Note the use of **toLabeled** to delimit the scope of Alice's data and demarcate the block of code where her data might influence control flow: after the **toLabeled** block is finished, the current label is restored to Bob and the binding a is no longer accessible.

```
report :: Labeled User Int → Labeled User Int
       → Labeled User Int → LIO User ()
report bobData daveData aliceData =
  do b  ← unlabel bobData
     lv ← (toLabeled Alice
           (do a ← unlabel aliceData
               if b > 10
                 then return (a + b)
                 else return a))
     d ← unlabel daveData
     writeLIORef bobReport (combine d b)
     v ← unlabel lv
     writeLIORef aliceReport v
```

Fig. 2: LIO code example

Afterwards, the function unlabels data from Dave, combines it with Bob's data, and writes it to the reference *bobReport*. These operations would be potentially forbidden if we had not used **toLabeled**, since unlabeling *aliceData* would have permanently raised the current label to Alice. Finally, the code unlabels the value returned from **toLabeled**, which raises the current label to Alice, and writes its contents to the reference *aliceReport*. Using **toLabeled** made it possible to perform side-effects at the level of Bob before the final write to *aliceReport*. As an illustration of the coarse-grainedness of the approach, note that in the label checks that are performed upon execution of **writeLIORef**, the particular values written to the references are irrelevant; when we write to *aliceReport*, the current label must flow to Alice, even if what we are attempting to write did not originate from an entity with label Alice.

6.3 Stateful LIO

In this section we introduce Stateful LIO, or SLIO for short: an extension of LIO with support for dynamic policies.

The key aspect of dynamic policies is that the ordering between labels can vary during execution. We therefore parametrise SLIO not only on the label format, but also on a data type representing the policy-relevant state of the application necessary to derive the relationship between labels. That is, the label type class now takes the form $\mathit{Label}\ \alpha\ \beta$, where α is the label format as before and β is the type of the structure representing the policy-relevant state.

As exemplified in Figure 1b, the ordering between labels does not have to form a lattice per se in a dynamic setting, and the *Label* type class therefore no longer requires instances to implement lattice operations such as $\sqcup$ and $\sqcap$. The only operation that a *Label* instance is required to provide is the (reflexive and transitive) relation between labels in a given state, which we denote by $\sqsubseteq$. That is, $\sqsubseteq$ is of the type $S \to \mathrm{L} \to \mathrm{L} \to \mathit{Bool}$ and $\sqsubseteq\ s\ l_1\ l_2$, denoted $l_1\ \sqsubseteq_s\ l_2$, returns *True* iff l_1 is less restrictive than l_2 in state s.

The LIO library only allows its own operations to interact with the current label. That is, only operations such as *readFile* and **unlabel** are allowed to read and modify l_{cur}. Similarly, SLIO's state contains both the current label and the current policy state st. The SLIO library exports operations that allow computations to read and modify st. Further encapsulations of the SLIO library may decide to only provide a limited interface to these operations, so as to better control the policy changes.

The floating label SLIO only requires label formats to provide the $\sqsubseteq$ relation, but this clashes with the original floating label approach of LIO. LIO tracks l-labeled information entering the computation by computing $l_{\text{cur}} \sqcup l$. With varying policy states, the join-operator $\sqcup$ gives a different result in a different state – at times an upper bound may not even exist, as is the case for Alice $\sqcup$ Bob in Figure 1b. We address this in a manner inspired by the theoretical enforcement mechanisms suggested in [Askarov and Chong, 2012]. We define lset to be a *set* of labels, to be used instead of the current label l_{cur}, and representing all labeled information that is present in the computation. Recording that l-labeled information has become accessible is then done by letting lset 'float up' to $\mathit{lset} \cup \{l\}$. Thus, lset behaves as the floating label in the powerset lattice of labels.

The original checks of the form $l_{\text{cur}} \sqsubseteq l_r$ that occurred e.g. when writing to a reference are replaced with a series of checks $\forall l \in \mathit{lset}.l \sqsubseteq_s l_r$ – where all checks need to hold in order for the flow to be allowed. That is, if all information that has entered the computation is allowed to flow to the label l_r (according to the current policy), we allow the program to write to a file with that label. We abbreviate this check as $\mathit{lset}\ \sqsubseteq_s\ l_r$.

The **toLabeled** operation requires the programmer to explicitly specify the label to be placed on the result of the provided computation. Its operation becomes stateful as well, now checking that $lset' \sqsubseteq_{st'} l$ where $lset'$ and st' are the current label resp. current policy state after executing the computation and l is the provided label.

6.3.1 Exploring SLIO

The principal function of SLIO is to provide off-the-shelf enforcement for encodings of dynamic policy languages such as Paralocks [Broberg and Sands, 2010] and DCLabels [Stefan et al., 2012b]. Before discussing this use of SLIO in more detail in § 6.7, we introduce the basic behaviour of SLIO programs using simple instantiations.

Static Policies SLIO is a strict generalisation of LIO. More concretely, if an instance of *Label* does not use the policy state component in $\sqsubseteq_s$, SLIO behaves exactly like LIO. We demonstrate this for the static lattice shown in Figure 1a by using the unit type () for policy state.

```
data User = Alice | Bob | Carl | Dave

instance Label User () where
  l1 ⊑s Alice = True
  Dave ⊑s l2  = True
  l1 ⊑s l2    = False
```

Instantiating SLIO with this *Label* format effectively enforces noninterference. To demonstrate this and later flows, we introduce a function *copy* which copies information from one reference into another. As is common in LIO we perform this operation in a **toLabeled** computation, to avoid tainting the current label unnecessarily.

```
copy :: LIORef User String → LIORef User String
     → SLIO () User ()
copy from to = toLabeled (labelOf from) (do
  info ← readLIORef from
  writeLIORef to info)
```

SLIO detects a violation of noninterference when data from Carl is copied to Bob.

```
nonInterfering :: SLIO () User ()
nonInterfering = do
  dataAlice ← newLIORef Alice "Alice's data"
  dataBob   ← newLIORef Bob   "Bob's data"
  dataCarl  ← newLIORef Carl  "Carl's data"
  copy dataCarl dataAlice   – Allowed flow.
  copy dataCarl dataBob     – Violation detected.
```

```
relabel l lv = toLabeled l (unlabel lv)
declassify l lv = do
  setState True
  result ← relabel l lv
  setState False
  return result
```

Fig. 3: Declassification.

Dynamic Policies The last information flow to Bob would have been allowed if Bob had been promoted according to the policy depicted in Figure 1b. To enforce this dynamic policy using SLIO we need to incorporate the state component. The most direct way to encode the dynamic nature of the policy is to store the set of allowed flows in the state. In Haskell notation, this is a list of *User* pairs: $[(User, User)]$.

```
instance Label User [(User, User)] where
  l1 ⊑s l2 = l1 ≡ l2 ∨ (l1, l2) ∈ transClosure s
type LIOCompany = SLIO [(User, User)] User
```

We define $\sqsubseteq_s$ such that we can minimise the set of flow relations in the state, using *transClosure* to ensure that the relation is transitive. For brevity we introduce the type synonym *LIOCompany* for this kind of SLIO computations. The following function initialises the policy state to the situation shown in Figure 1a:

```
setInitState :: LIOCompany ()
setInitState = putState [(Dave, Bob), (Dave, Carl)
                        , (Bob, Alice), (Carl, Alice)]
```

The function *aliceLeaves* implements the event where Alice leaves the company, changing the label ordering from Figure 1a to Figure 1b.

```
aliceLeaves :: LIOCompany ()
aliceLeaves = do
  s ← getState
  putState ((s ++ [(Carl, Bob)]) \\
           [(Bob, Alice), (Carl, Alice), (Dave, Carl)])
```

Assuming the references from the previous example, the dynamic nature of the information-flow policy can be manifested as follows:

```
dynamic :: LIOCompany ()
dynamic = do
  setInitState
```

copyFile dataCarl dataAlice – Allowed flow.
aliceLeaves
copyFile dataCarl dataBob – Allowed flow.
copyFile dataCarl dataAlice – Violation detected.

Relabeling Figure 3 shows the function *relabel* which relabels a labeled value lv with the label l. This function can be used to perform declassification [Sabelfeld and Sands, 2005] using the following technique, described in [Broberg et al., 2013]. Assume two security levels *Low* and *High*. The policy state is of type *Bool* and information can only flow from *High* to *Low* when the state is *True*. All other flows are allowed in either state. Figure 3 displays how this allows us to write *declassify* by temporarily changing the state and calling *relabel*. We revisit this pattern in § 6.6 where we construct a security condition which explicitly allows for persistent relabelings of information.

6.4 Conditional Change in Label Ordering

Allowing programs to freely change the policy state results in uncontrolled information flows, previously not present in LIO. This section establishes a condition on state change which ensures the absence of such flows. We identify this condition as a separate contribution of this paper, since it can also be applied on other enforcement mechanisms with dynamic policies (e.g. Paragon [Broberg et al., 2013]).

We demonstrate the type of flow via a minimal example, assuming levels *Low* and *High* and a boolean state as in the relabeling example discussed above. Figure 4 displays a program which creates a *Low* reference r. When the *High* information provided equals 0 the computation changes the state to allow this information to flow to r. The result of the **toLabeled** computation gets labeled *High* but is ignored by the rest of the computation.

We assume the computation starts with $lset = \emptyset$ and policy state *False*. If the value of *highData* is 0, r is updated while $l_{\text{cur}} = \{High\}$, which is allowed since $l_{\text{cur}} \sqsubseteq_{True} Low$. $lset$ is set back to $\emptyset$ after the **toLabeled** computation, and the policy state is again *False*. Thus after reading r and returning its value, $lset = \{Low\}$.

Although it might appear as if information only flows from *High* to *Low* when the policy state is *True*, this is not the case. In particular, when $highData \neq 0$, we learn this by observing that the value in r did not change. Thus information flows from *High* to *Low* even though the policy state is never set to *True* in the entire computation. Clearly the program should not be considered secure.

The computation's decision to allow the flow to *Low* is based on information which, at the moment of decision, is *not* allowed to flow to

```
leak highData = do
  r ← newLIORef Low 1
  _ ← toLabeled High (do
    h ← unlabel highData
    when (h ≡ 0) (do
      setState True
      writeLIORef r 0))
  v ← readLIORef r
  return v
```

Fig. 4: Information leaks via conditional state change.

Low. We identify this as the root of the problem. If instead the policy state changes to *True* just *before* the **when** condition, the program would semantically be secure as it would *allow the conditional flow*, rather than to *conditionally allow the flow*.[3] This could be interpreted as the need to preserve the monotonicity property of the original LIO: information in scope can only become more confidential. That is, if at some point during the computation information can no longer flow to some label l, nothing can flow to l in the rest of this (**toLabeled**) computation either.

In general, whenever the policy state changes from s_1 to s_2, we need to ensure that s_2 does not allow flows from labels in *lset* which were previously disallowed in s_1. In other words, *the upper closure of lset should not increase by changing the ordering from* $\sqsubseteq_{s_1}$ *to* $\sqsubseteq_{s_2}$. To enforce this we require each instance of *Label* to define an operation $incUpperSet :: S \to S \to \mathrm{L} \to Bool$, where $incUpperSet\ s_1\ s_2\ l$ returns *True* if the upper set for label l increases; that is, if there exists an l' such that $l \not\sqsubseteq_{s_1} l'$ and $l \sqsubseteq_{s_2} l'$. The SLIO library then checks whether $\forall l \in lset\ .\ \neg\ (incUpperSet\ s_1\ s_2\ l)$.

In the example displayed in Figure 4, the call **setState** causes the SLIO library to check $incUpperSet\ False\ True\ \{High\}$. This should under a correct implementation return *True*, since the change in ordering increases the upper closure of $lset = \{High\}$ from $\{High\}$ to $\{Low, High\}$.

The requirement that $\sqsubseteq_s$ is a transitive relation is especially relevant here, since this check aims to control the yet unknown remainder of the execution, where flows might happen in a transitive manner. That this check enforces the monotone property of SLIO and prevents the information flows arising from policy state change is an essential step in the proof for our security condition (§ 6.6).

[3] Since the floating label approach does not distinguish explicit from implicit flows, the **setState** operation should, in practice, be placed before the unlabeling of *highData*.

$$
\begin{array}{rcl}
\textit{Values}\ v & ::= & \textit{True} \mid \textit{False} \mid () \mid \lambda x.e \mid \ell \mid \textit{SLIO}\ e \mid \textit{Lb}\ l\ e \\
\textit{Expr.}\ e & ::= & v \mid x \mid e\ e \mid \textbf{fix}\ e \mid \textbf{if}\ t\ \textbf{then}\ e\ \textbf{else}\ e \\
& & \mid \textbf{return}\ e \mid e \gg\!= e \mid \textbf{getLabel} \\
& & \mid \textbf{toLabeled}\ l\ e \mid \textbf{toLabeledRet}\ \textit{ls}\ s\ l\ e \\
& & \mid \textbf{label}\ e\ e \mid \textbf{unlabel}\ e \\
& & \mid \textbf{labelOf}\ e \mid \sqsubseteq \mid \textbf{newLIORef}\ e\ e \\
& & \mid \textbf{writeLIORef}\ e\ e \mid \textbf{readLIORef}\ e \\
& & \mid \textbf{setState}\ e \mid \textbf{getState}
\end{array}
$$

Fig. 5: λ_{SLIO} syntax.

6.5 Semantics

In this section we formalise SLIO as a simply-typed, call-by-name λ-calculus, which we call λ_{SLIO}. Figure 5 gives the formal syntax of λ_{SLIO}, parametric in the label type ℓ. Syntactic categories v and e represent values and expressions, respectively. Expressions of the form $\textit{SLIO}\ e$, $\textit{Lb}\ l\ e$ and **toLabeledRet** $\textit{ls}\ s\ l\ e$ are not part of the surface syntax, i.e., they are not made available to programmers and are solely used internally to provide semantics to the other expressions. Values include standard primitives (Booleans, unit, and λ-abstractions) and terminals corresponding to labels (ℓ) and monadic values ($\textit{SLIO}\ e$). The latter denote effectful computations subject to security checks. Expressions consist of standard constructs (values, variables x, function application, the **fix** operator, and conditionals), a terminal corresponding to $\sqsubseteq$ (the partial order on labels), standard monadic operators (**return** e and $e \gg\!= e$), **getLabel**, **toLabeled**, operations on labeled values and references, and operations for setting and getting the policy state (**setState** and **getState**). Even though the full LIO library can handle several other kinds of entities, such as files, we focus on labeled values and references since they accurately represent the security mechanisms of LIO; the security checks and effects on other kinds of labeled entities are analogous. For brevity, we do not describe the λ_{SLIO} type system since it is standard and is not relevant for security checks. In what follows, we assume that all expressions involved are well-typed.

A top-level λ_{SLIO} computation is a *configuration* of the form $\langle \Sigma | e \rangle$, where e is the monadic expression and Σ is the state associated with the expression. The state Σ contains the current label set $\textit{lset}$, the current policy state st, and the store ϕ (for references). We give a small-step operational semantics for λ_{SLIO} in the form of a reduction relation $\longrightarrow$. Figure 6 shows the relevant reduction rules for $\longrightarrow$. Intuitively, $\langle \Sigma | e \rangle \longrightarrow \langle \Sigma' | e' \rangle$ means that, starting from a configuration $\langle \Sigma | e \rangle$, it is possible to take a step to $\langle \Sigma' | e' \rangle$. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

The reduction rules for λ_{SLIO} are specified using evaluation contexts in the style of Felleisen and Hieb [Felleisen and Hieb, 1992]. Figure 7 defines the evaluation contexts for pure expressions (E) and monadic ($\mathbf{E}$) expressions for λ_{SLIO}. The definitions are mostly standard. Note that monadic expressions are evaluated only at the outermost use of bind ($\mathbf{E} \gg\!= e$), as in Haskell.

Rule (WRITELIOREF) is used to assign a value to a mutable reference. The rule looks up the reference in the memory store ϕ, where it is represented as a labeled value $Lb\ l\ v$. Then, a security check is performed to ensure that the current label set flows to l ($lset\ \sqsubseteq_{st}\ l$) (note that this is actually the conjunction of several checks, one per label in the label set). If the check passes, the memory store is updated with the new value.

Rule (READLIOREF) reads a value from a mutable reference. The current label set is updated to include the label of the reference, to reflect the fact that the contents of the reference are now in scope and could potentially influence side-effects in the future. The rules for labeled values interact with the current label set in an analogous manner.

Rules (SET) and (GET) define the semantics for the new operations in λ_{SLIO}, namely **setState** and **getState**. As expected, they work by writing and reading the policy state in the λ_{SLIO} state, except that **setState** additionally checks that the upper closure is not increased. This check is performed by a user-supplied function, as explained in Section 6.4.

Finally, the rule (TOLABELED) binds the monadic computation m to the internal-only expression **toLabeledRet**. The rule (TOLABELEDRET) resets the policy-relevant components to their value before m, returning the result of m as a labeled value only if the current configuration allows information to flow to the label l specified by the programmer.

6.6 Semantic Soundness

In this section we define a security condition for dynamic policies and show that it is guaranteed by λ_{SLIO}. We first present the attacker model, which is similar to the one used for static policies in LIO [Stefan et al., 2011].

6.6.1 Attacker model

SLIO aims to provide security guarantees even in the presence of untrusted code. Following this assumption, we make configurations the observations of our model.

Definition 1 (Trace). *A configuration produces a* trace *of configurations, written* $\langle \Sigma_0 | e_0 \rangle \Downarrow t$ *with* t *a sequence of configurations* $\langle \Sigma_0 | e_0 \rangle \ldots \langle \Sigma_n | e_n \rangle$, *if there exists an evaluation* $\langle \Sigma_0 | e_0 \rangle \longrightarrow \ldots \longrightarrow \langle \Sigma_n | e_n \rangle$.

$$\textsc{getLabel}\quad \frac{\Sigma = (lset, st, \phi)}{\langle \Sigma | \mathbf{E}\ [\mathbf{getLabel}] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\mathbf{return}\ lset] \rangle}$$

$$\textsc{label}\quad \frac{\Sigma = (lset, st, \phi) \qquad lset\ \sqsubseteq_{st}\ l}{\langle \Sigma | \mathbf{E}\ [\mathbf{label}\ l\ e] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\mathbf{return}\ (Lb\ l\ e)] \rangle}$$

$$\textsc{unlabel}\quad \frac{\Sigma = (lset, st, \phi) \qquad lset' = lset\ \cup\ \{l\} \qquad \Sigma' = (lset', st, \phi)}{\langle \Sigma | \mathbf{E}\ [\mathbf{unlabel}\ (Lb\ l\ e)] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ e] \rangle}$$

$$\textsc{labelOf}\quad \frac{}{E\ [\mathbf{labelOf}\ (Lb\ l\ e)] \longrightarrow E\ [l]}$$

$$\textsc{newLIORef}\quad \frac{\Sigma = (lset, st, \phi) \qquad lset\ \sqsubseteq_{st}\ l \qquad \Sigma' = (lset, st, \phi\ [x \rightarrow Lb\ l\ e]) \qquad \mathrm{fresh}(x)}{\langle \Sigma | \mathbf{E}\ [\mathbf{newLIORef}\ l\ e] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ x] \rangle}$$

$$\textsc{writeLIORef}\quad \frac{\begin{array}{c}\Sigma = (lset, st, \phi) \\ Lb\ l\ v = \phi\ (x) \qquad lset\ \sqsubseteq_{st}\ l \qquad \Sigma' = (lset, st, \phi\ [x \rightarrow Lb\ l\ e])\end{array}}{\langle \Sigma | \mathbf{E}\ [\mathbf{writeLIORef}\ x\ e] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ ()] \rangle}$$

$$\textsc{readLIORef}\quad \frac{\begin{array}{cc}\Sigma = (lset, st, \phi) & Lb\ l\ e = \phi\ (x) \\ lset' = lset\ \cup\ \{l\} & \Sigma' = (lset', st, \phi)\end{array}}{\langle \Sigma | \mathbf{E}\ [\mathbf{readLIORef}\ x] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ e] \rangle}$$

$$\textsc{set}\quad \frac{\Sigma = (lset, st, \phi) \qquad \forall l \in lset\ .\ \neg\ incUpperSet\ (st, v, l) \qquad \Sigma' = (lset, v, \phi)}{\langle \Sigma | \mathbf{E}\ [\mathbf{setState}\ v] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ ()] \rangle}$$

$$\textsc{get}\quad \frac{\Sigma = (lset, st, \phi)}{\langle \Sigma | \mathbf{E}\ [\mathbf{getState}] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\mathbf{return}\ st] \rangle}$$

$$\textsc{toLabeled}\quad \frac{\Sigma = (lset, st, \phi)}{\langle \Sigma | \mathbf{E}\ [\mathbf{toLabeled}\ l\ m] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [m \gg\!= \mathbf{toLabeledRet}\ lset\ st\ l] \rangle}$$

$$\textsc{toLabeledRet}\quad \frac{\Sigma = (lset, st, \phi) \qquad lset\ \sqsubseteq_{st}\ l}{\langle \Sigma | \mathbf{E}\ [\mathbf{toLabeledRet}\ ls\ s\ l\ v] \rangle \longrightarrow \langle (ls, s, \phi) | \mathbf{E}\ [\mathbf{return}\ (\mathtt{Lb}\ l\ v)] \rangle}$$

Fig. 6: SLIO semantics (standard λ-calculus rules elided).

$$
\begin{aligned}
E ::= {} & [\,] \mid E\ e \mid \mathbf{fix}\ E \mid \mathbf{if}\ E\ \mathbf{then}\ e\ \mathbf{else}\ e \\
& \mid \mathbf{label}\ E\ e \mid \mathbf{unlabel}\ E \mid \mathbf{labelOf}\ E \mid \mathbf{toLabeled}\ E\ e \\
& \mid \mathbf{newLIORef}\ E\ e \mid \mathbf{writeLIORef}\ E\ e \mid \mathbf{readLIORef}\ E \\
& \mid \mathbf{setState}\ E \\
\mathbf{E} ::= {} & E \mid \mathbf{E} \gg\!= e
\end{aligned}
$$

Fig. 7: Evaluation contexts for SLIO.

$$
\varepsilon_A(\langle \Sigma | e \rangle \cdot t) = \begin{cases} \langle \varepsilon^s_A(\Sigma) | \varepsilon^s_A(e) \rangle \cdot \varepsilon_A(t) & \text{if } obs_A(\langle \Sigma | e \rangle), \\ & \text{with } s = \Sigma.st \\ \varepsilon_A(t) & \text{otherwise} \end{cases}
$$

$$
\varepsilon^s_A(\Sigma) = \Sigma[\phi \mapsto \varepsilon^s_A(\Sigma.\phi)] \qquad \varepsilon^s_A(\Sigma.\phi) = \{(x, \varepsilon^s_A(\Sigma.\phi\ (x))) \mid x \in \mathrm{dom}(\Sigma.\phi)\}
$$

$$
\varepsilon^s_A(\mathtt{Lb}\ l\ e) = \begin{cases} \mathtt{Lb}\ l\ \varepsilon^s_A(e) & \text{if } l \sqsubseteq_s A \\ \mathtt{Lb}\ l\ \bullet & \text{otherwise} \end{cases}
$$

Fig. 8: Erasure function for non-trivial cases.

As in [Stefan et al., 2011] we use a technique called *term erasure*. Attackers are represented by a security level A. The function $\varepsilon_A(t)$ erases from the trace of configurations t all the information which is not observable on level A.

Since the current label set protects all available information, A can only observe configurations where the current label set can flow to A (according to the current policy state).

Definition 2 (A**-observable configuration).** *A configuration* $\langle \Sigma | e \rangle$ *is* observable *to an attacker on level* A*, written* $obs_A(\langle \Sigma | e \rangle)$*, iff* $\Sigma.lset \sqsubseteq_{\Sigma.st} A$.

Configurations which are not observable to A are removed from the trace entirely, as shown in Figure 8. From the configurations that are not removed, the erasure function erases only the information that cannot flow to A, so the erased configuration is $\langle \varepsilon^s_A(\Sigma) | \varepsilon^s_A(e) \rangle$. Here we fix s as the current policy state in that configuration, i.e, $s = \Sigma.st$.

For most cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon^s_A(\ \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) = \mathbf{if}\ \varepsilon^s_A(e)\ \mathbf{then}\ \varepsilon^s_A(e_1)\ \mathbf{else}\ \varepsilon^s_A(e_2)$). The interesting cases for this function are displayed in Figure 8. The syntax node $\bullet$ represents an erased expression: information that is not observable to an attacker at level A. In particular, $\varepsilon^s_A(\mathtt{Lb}\ l\ e)$ erases to $\mathtt{Lb}\ l\ \bullet$ when $l \not\sqsubseteq_s A$.

6.6.2 Security Condition

The two-run noninterference condition associated with LIO does not translate well to a setting with dynamic policies. Instead, we find that epistemic properties [Askarov and Sabelfeld, 2007, Balliu et al., 2011, Broberg and Sands, 2010] form a more natural basis for defining information flow conditions, in particular in the context of dynamic policies.

As a starting point we adapt the security condition from Askarov and Chong [Askarov and Chong, 2012]. This condition extends from the notion of *gradual release* [Askarov and Sabelfeld, 2007], which builds around the concept of a *knowledge set*: the set of initial inputs that could have resulted in the observations made by an attacker. Following Delft et al. [Delft et al., 2015] we instead talk about the *exclusion* knowledge set: the set of initial inputs that could *not* have resulted in these observations. This matches the intuition that a larger (exclusion) knowledge set implies more knowledge.

Since λ_{SLIO} values can be SLIO computations, we let the initial expression take the role of initial (secret) input. Let e be such a secret input which is evaluated in Σ_0, the initial state with $lset = \emptyset$ and $\phi = \emptyset$ – the initial value of st varies between instantiations. Given $\langle \Sigma_0 | e \rangle \Downarrow t$, i.e. this configuration produces a sequence of configurations t, let $o = \varepsilon_A(t)$ the observations made by attacker A. The exclusion knowledge of A is then defined as the set of inputs that could not have produced the same observations:

$$ek_A(o) = \{e' \mid \neg \exists t'. \langle \Sigma_0 | e' \rangle \Downarrow t' \text{ with } \varepsilon_A(t') = o\}$$

Now let $\langle \Sigma_0 | e \rangle \Downarrow t \cdot \alpha$, with $obs_A(\alpha)$. What an attacker learns from this new configuration α can then be expressed as $ek_A(\varepsilon_A(t \cdot \alpha)) \setminus ek_A(\varepsilon_A(t))$: the set of inputs additionally excluded. To specify that the attacker does not learn anything new from this observation, we can simply require that $ek_A(\varepsilon_A(t \cdot \alpha)) \setminus ek_A(\varepsilon_A(t)) = \emptyset$.

This definition would also not allow the attacker to learn anything from the fact that the computation produced another output after producing trace t. SLIO, and LIO, however do not check for leaks via progress and allow computations to diverge based on sensitive information. This means that information might e.g. be leaked by the fact that a **toLabeled** computation terminated. Askarov and Chong present a termination-insensitive condition by introducing the attacker's *progress knowledge*. That is, we allow the attacker to exclude also those initial commands e' that cannot produce another observation:

$$ek_A^+(o) = \{e' \mid \neg \exists t', \alpha'. \langle \Sigma_0 | e' \rangle \Downarrow t' \cdot \alpha' \text{ with } \varepsilon_A(t') = o \text{ and } obs_A(\alpha')\}$$

Finally, we do allow the attacker to exclude *some* initial inputs using observation α, as long as this is in accordance with the ordering determined by the state s in which α was produced. Following Askarov and Chong,

we allow the attacker to exclude those inputs that are not equal to e when observed under state s.

Definition 3 (Input release). *Given input e, the state s allows an attacker A to exclude the set of inputs $I_A(e, s)$, where*

$$I_A(e, s) = \{e' \mid \varepsilon_A^s(e) \neq \varepsilon_A^s(e')\}$$

The security condition is then that for every $\langle \Sigma_0 | e \rangle \Downarrow t \cdot \alpha$ with $obs_A(\alpha)$, $\alpha = \langle \Sigma_n | e_n \rangle$ and $\Sigma_n.st = s$, the attacker's increase in knowledge is bounded by $I_A(e, s)$:

$$ek_A(\varepsilon_A(t \cdot \alpha)) \setminus ek_A^+(\varepsilon_A(t)) \subseteq I_A(e, s)$$

Relabeling support The examples from § 6.3.1 show that SLIO allows for persistent relabelings of data. By this we mean that we want to have the possibility to place a value of label l_1 in a container with label l_2, after which the value from this container is treated as if it has label l_2. We used this in a simple encoding of declassification, shown in Figure 3.

Such relabelings are a desirable feature of an IFC language. The fact that persistent relabelings are part of various policy languages that we would like SLIO to encode, notably including the DLM described in § 6.7, further motivates the need for a security condition that supports them.

It turns out that our direct adaptation of Askarov and Chong's security condition does *not* allow for persistent relabelings, as the example in Figure 9 shows. An attacker of level *Public* observes the value of *pub* (and therefore learns the value of *top*) when the current ordering does not allow flows from *TopSecret* to *Public*. The security condition requires that a run started in a state with a different value for *top* should yield the same value for *pub* as in the observed run. Since this is not the case, the program violates the security condition. In the terminology of facets of dynamic policies [Broberg et al., 2015], the condition does not allow for the *time-transitive* flows that we desire.

```
add (TopSecret ⊑ Secret)
sec ← relabel Secret top
remove (TopSecret ⊑ Secret)
add (Secret ⊑ Public)
pub ← relabel Public sec
```

Fig. 9: Relabeling example.

We conclude that we need to allow the attacker to exclude inputs based on relabeled information, *in addition to* the inputs described by $I_A(e, s)$.

Given the flow relation determined by policy state s, let L be the set of levels from which A is allowed to learn. That is, $L = \{l \mid l \sqsubseteq_s A\}$. To define the information that is collectively known by L, we introduce the erasure function on traces for multiple levels $\varepsilon_{\mathrm{L}}(t)$, shown in Figure 10. This function only erases labeled data or configurations if none of the levels in L can observe it.

Given $\langle \Sigma_0 | e \rangle \Downarrow t \cdot \alpha$, with $obs_A(\alpha)$, and s the policy state in α. We can specify the information that is released to A by relabelings in t as follows.

Definition 4 (Relabeling release). *Given a trace t, the policy state s allows an attacker A to exclude the set of inputs $R_A(t, s)$, where $L = \{l \mid l \sqsubseteq_s A\}$ in*

$$R_A(t,s) = \{e' \mid \neg\exists t', \alpha'. \langle \Sigma_0 | e' \rangle \Downarrow t' \cdot \alpha' \textit{ with } obs_A(\alpha') \\ \textit{and } \varepsilon_L(t) = \varepsilon_L(t')\}$$

We straightforwardly extend the security condition to additionally allow an attacker to learn information that has been released by relabelings.

Definition 5 (Termination-insensitive security). *The command e is secure against an attacker A if for all traces t and configurations α such that $\langle \Sigma_0 | e \rangle \Downarrow t \cdot \alpha$ with $obs_A(\alpha)$, $\alpha = \langle \Sigma_n | e_n \rangle$ and $\Sigma_n.st = s$, the attacker's increase in knowledge is bounded by $I_A(e, s)$ and $R_A(t, s)$:*

$$ek_A(\varepsilon_A(t \cdot \alpha)) \setminus ek_A^+(\varepsilon_A(t)) \subseteq I_A(e,s) \cup R_A(t,s)$$

$$\varepsilon_L(\langle \Sigma | e \rangle \cdot t) = \begin{cases} \langle \varepsilon_{\mathrm{L}}^s(\Sigma) | \varepsilon_{\mathrm{L}}^s(e) \rangle \cdot \varepsilon_{\mathrm{L}}(t) \\ \quad \text{if } \exists l \in L \,.\, obs_l(\langle \Sigma | e \rangle), s = \Sigma.st \\ \varepsilon_{\mathrm{L}}(t) \quad \text{otherwise} \end{cases}$$

$$\varepsilon_{\mathrm{L}}^s(\texttt{Lb}\ l\ e) = \begin{cases} \texttt{Lb}\ l\ \varepsilon_{\mathrm{L}}^s(e) & \text{if } \exists l' \in L \,.\, l \sqsubseteq_s l' \\ \texttt{Lb}\ l\ \bullet & \text{otherwise} \end{cases}$$

Fig. 10: Multi-level erasure function for cases different from single-level erasure.

Remark 1. Our choice in defining the set $R_A(t, s)$ is not an arbitrary one. In Appendix C we list a collection of other possible definitions that also appear reasonable, but either do not support relabelings to the extent that we find natural, or allow for flows that we consider insecure, such as the release of information via conditional state change.

Remark 2. Askarov and Chong identify that a perfect recall attacker might learn less from an observation than an attacker who has forgotten part of the earlier knowledge (i.e. an attacker with some knowledge $wk \subset ek_A^+(\varepsilon_A(t))$. Although our definitions assume a perfect recall attacker, we observe that $ek_A^+(\varepsilon_A(t)) \subseteq R_A(t,s)$ since L always includes A itself. Therefore the security condition could be specified as $ek_A(\varepsilon_A(t \cdot \alpha)) \subseteq I_A(e,s) \cup R_A(t,s)$. Hence by allowing for relabeling release by Definition 4, a program that is secure by Definition 5 is also secure against even the most forgetful attacker with $wk = \emptyset$.

Theorem 1. *All λ_{SLIO} computations are termination-insensitive secure.*

Proof. See Appendix A. □

6.7 Encodings

To demonstrate the genericity of SLIO we provide encodings for various policy specification frameworks. For each policy language, SLIO provides an enforcement mechanism in exchange for the relatively minor effort of encoding that language. This allows for easy exploration of policy languages, as well as the effects of modifying and extending them. We expect user applications to be typically written against such an encoding, rather than creating an *ad hoc* policy language using 'bare' SLIO (as we did in § 6.3.1). Using an existing policy language one can write natural policy labels with well-established semantics.

The following policy languages have been encoded in SLIO and are available from [Buiras and van Delft]: Two-Point Lattice, Flow-policies for non-disclosure [Matos and Boudol, 2005], the Decentralized Label Model (DLM) [Myers and Liskov, 1998], Disjunction Category Labels [Stefan et al., 2012b] and Paralocks [Broberg and Sands, 2010].

Rather than the dynamic policy-oriented Disjunction Category labels or Paralocks, we use this section to present the DLM encoding in more detail, for the following reasons.

- The DLM is well-known and widely used in research.
- All information relabelings need to pass a dedicated *declassify* function. We show how this common pattern can be enforced with dynamic policies using the right encoding in SLIO (following an encapsulation technique similar to [Broberg et al., 2013]).
- Although typically not supported by implementations, the DLM does contain dynamic features. More specifically, the DLM includes a hierarchy among principals which is subject to change, but these changes are '*assumed to occur infrequently*' [Myers, 1999]. Jif, an extension to Java with support for the DLM, relies on this assumption when verifying that applications are information-flow secure. By encoding the DLM

in SLIO we can guarantee security even in the presence of hierarchy change.

Remark 3. Since its introduction by Myers and Liskov [Myers and Liskov, 1997], various information-flow concepts have been added to the DLM, such as robust declassification [Zdancewic and Myers, 2001] and information erasure [Chong and Myers, 2005]. We consider the DLM as used in the first iteration of the Jif compiler [Myers, 1999], matching most closely the model described in [Myers and Liskov, 1998].

The DLM Language In DLM, the security label $l_1 = \{o_1 : r_2, r_3; o_2 : r_3, r_4\}$ specifies that data is *owned* by the *principals* o_1 and o_2. Each owner specifies a different set of principals they allow to *read* this data. Effectively, the only principal that they both allow to read the data is r_3. The DLM includes an ordering among principals, the *acts-for* hierarchy $\succeq$. In a setting where principal $r_2 \succeq r_4$, label l_1 is equivalent to the label $l_2 = \{o_1 : r_2, r_3;\ o_2 : r_2, r_3, r_4\}$. That is, since o_2 allows r_4 to read the data, o_2 implictly allows r_2 as well. Labels l_1 and l_2 are also equivalent in a setting where $o_1 \succeq o_2$. That is, each principal that is allowed to read data by o_1 is implicitly also allowed to read that data by o_2. This hierarchy may be modified at run time.

The DLM assumes the existence of a *declassify* statement which makes the label of the provided data more permissive, either by extending an owner's reader-set or by removing an owner's concern entirely. Declassification is only permitted if the owners for whom information is declassified allowed for this by giving the computation their *authority*.

Representing the DLM in SLIO The state component of the SLIO encoding of the DLM contains *i*) a boolean indicating whether or not declassification is allowed; *ii*) a set of principals who have given authority to the current computation; and *iii*) the set of principal pairs indicating the current hierarchy $\succeq$, similar to the hierarchy among *User*s in Section 6.3.1. The DLM encoding does not expose the **setState** operation from SLIO directly to user code. Helper functions are provided to change the hierarchy, and information can be declassified using the exposed *declassify* function. The *declassify* function uses the boolean element of the state as in Figure 3 to relabel the information, but only if such declassification is allowed at that moment. By means of this encapsulation, we can provide the necessary guarantees on declassifications, even in the presence of a changing acts-for hierarchy. A more detailed discussion of the DLM encoding can be found in Appendix B.

6.8 Related work

Supporting dynamic policies is the next step in the natural evolution of security conditions from noninterference and declassification [Goguen and

Meseguer, 1982, McCullough, 1988, Sabelfeld and Sands, 2005]. Balliu [Balliu et al., 2011], Broberg and Sands [Broberg and Sands, 2010] and Askarov and Chong [Askarov and Chong, 2012] construct conditions for dynamic policies on top of the epistemic gradual release property, originally created to support declassification [Askarov and Sabelfeld, 2007]. Delft et al. [Delft et al., 2015] show that epistemic properties can be unfolded into two-run properties, a technique we also use in the proof the soundness of our enforcement system.

A different approach to defining dynamic security policies can be traced back to the early work of Goguen and Meseguer on conditional noninterference [Goguen and Meseguer, 1984], where noninterference relations on machine models only need to hold provided that some condition on the execution history holds. Zhang [Zhang, 2012] expands on this, presenting a set of unwinding relations that can be verified by existing proof assistants.

The dynamic policies considered by SLIO are of a synchronous nature. That is, the policy changes deterministically with program execution. Other work considers asynchronous policies, such as Hicks et al. [Hicks et al., 2005] and Swamy et al. [Swamy et al., 2006]. Both approaches do require some synchronisation mechanism between the policy and the program execution.

Concerning IFC libraries for Haskell, the seminal work by Li and Zdancewic [Li and Zdancewic, 2006] consists in a library for enforcing information-flow security using arrows [Hughes, 2005], a generalisation of monads. Russo et al. [Russo et al., 2008] show a monadic IFC security library, which statically enforces noninterference by leveraging Haskell's type system. Stefan et al. [Stefan et al., 2011] propose LIO, which uses monads to track information-flow dynamically. Morgenstern et al. [Morgenstern and Licata, 2010] encode an IF-aware programming language in Agda, without considering computations with side-effects. Devriese and Piessens [Devriese and Piessens, 2010] use monad transformers and parametrised monads to enforce noninterference. Unlike SLIO, none of the approaches mentioned above support dynamic policies or declassification in their semantic conditions, although for practical reasons some of them provide special constructs for declassification in their implementation.

Breeze is a programming language with IFC proposed by Hritcu et al. [Hritcu et al., 2013] which, like LIO, is based on the floating-label approach. In this system, lowering labels on values or the program counter (c.f. current label in LIO) is a privileged operation that requires special authority. Given the design similarities with LIO [Stefan et al., 2011], we believe that our results could be easily adapted to Breeze.

6.9 Conclusions and Future Work

We have explored dynamic policies in a dynamic IFC setting by presenting SLIO, a strict generalisation of LIO with support for generic enforcement of dynamic policies. We have shown SLIO sound with respect to an epistemic security condition for dynamic policies with relabelings. We also demonstrated its practical use by encoding multiple policy frameworks which are available on [Buiras and van Delft] together with the SLIO library and the technical report version of this paper.

As future work, we intend to generalise the singular labels on labeled values and references to become *sets* of labels, thereby making them more homogenous with the rest of the enforcement. That is, like the current label set, these labels become elements in the power set lattice of security labels.

We also propose to examine extensions of SLIO with more advanced language-level features, such as concurreny and exceptions. Supporting concurreny appears to be particularly challenging, since it is not clear whether the policy changes performed in one thread can be made available to other threads while preserving soundness.

Finally, we remark that the library presented here could serve as a convenient testbed for future encodings of policy frameworks and comparing their relative expressive power.

Acknowledgments This paper benefited from the comments of David Sands, Niklas Broberg and the anonymous reviewers. This work is partly funded by the Swedish funding agencies SSF and VR.

BIBLIOGRAPHY

Aslan Askarov and Stephen Chong. Learning is Change in Knowledge: Knowledge-based Security for Dynamic Policies. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 308–322, Piscataway, NJ, USA, June 2012. IEEE Press.

Aslan Askarov and Andrei Sabelfeld. *Security-typed languages for implementation of cryptographic protocols: A case study*. Springer, 2005.

Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 207–221. IEEE, 2007.

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Computer Security-ESORICS 2008*, pages 333–348. Springer, 2008.

Musard Balliu, Mads Dam, and Gurvan Le Guernic. Epistemic temporal logic for information flow security. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, page 6. ACM, 2011.

Niklas Broberg and David Sands. Paralocks – Role-Based Information Flow Control and Beyond. In *POPL'10, Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.

Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *Programming Languages and Systems: 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, pages 217–232. Springer, 2013.

Niklas Broberg, Bart van Delft, and David Sands. The Anatomy and Facets of Dynamic Policies. In *Computer Security Foundations*, 2015. To appear.

Pablo Buiras and Bart van Delft. Dynamic Enforcement of Dynamic Policies - TR. `http://slio.bitbucket.org`. Accessed: 2014-10-18.

Stephen Chong and Andrew C Myers. Language-based information erasure. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 241–254, 2005.

Ellis Cohen. Information transmission in computational systems. In *ACM SIGOPS Operating Systems Review*, volume 11, pages 133–139. ACM, 1977.

Bart van Delft, Sebastian Hunt, and David Sands. Very Static Enforcement of Dynamic Policies. In *Principles of Security and Trust*. Springer, 2015.

D. Devriese and F. Piessens. Noninterference through Secure Multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, 103(2):235–271, 1992.

Joseph A Goguen and José Meseguer. Security policies and security models. In *2012 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE Computer Society, 1982.

Joseph A Goguen and José Meseguer. Unwinding and inference control. In *2012 IEEE Symposium on Security and Privacy*, pages 75–75. IEEE Computer Society, 1984.

Michael Hicks, Stephen Tse, Boniface Hicks, and Steve Zdancewic. Dynamic updating of information-flow policies. In *Proc. of Foundations of Computer Security Workshop*, volume 20, 2005.

Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All Your IFCException Are Belong to Us. *2012 IEEE Symposium on Security and Privacy*, 2013.

John Hughes. Programming with arrows. In *Advanced Functional Programming*, pages 73–129. Springer, 2005.

P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

Ana Almeida Matos and Gerard Boudol. On declassification and the non-disclosure policy. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 226–240. IEEE, 2005.

Daryl McCullough. Noninterference and the composability of security properties. In *Security and Privacy, 1988. Proceedings., 1988 IEEE Symposium on*, pages 177–186. IEEE, 1988.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently typed programming. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.

Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

Andrew C Myers and Barbara Liskov. *A decentralized model for information flow control*, volume 31. ACM, 1997.

Andrew C Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 186–197, 1998.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. pages 13–24. ACM Press, September 2008.

Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. In *In Proceedings of the 18th IEEE Workshop on Computer Security Foundations (CSFW'05*, pages 255–269, 2005.

V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/ soft/flowcaml`, July 2003.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.

Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Maziéres. Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 201–214, New York, NY, USA, 2012a. ACM.

Deian Stefan, Alejandro Russo, David Mazieres, and John C Mitchell. Disjunction category labels. In *Information Security Technology for Applications*, pages 223–239. Springer, 2012b.

Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *Computer Security Foundations Workshop, 2006. 19th IEEE*, pages 13–pp. IEEE, 2006.

David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. In *ACM SIGPLAN Notices*, volume 47, pages 137–148. ACM, 2012.

Steve Zdancewic and Andrew C Myers. Robust declassification. In *Computer Security Foundations Workshop, IEEE*, pages 15–15, 2001.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278. USENIX Association, 2006.

Chenyi Zhang. Conditional information flow policies and unwinding relations. In *Trustworthy Global Computing*, pages 227–241. Springer, 2012.

A Proof for Theorem 1

Notation As additional notation for this appendix, we introduce $\varepsilon_A(\langle \Sigma | e \rangle)$ as a shorthand for $\langle \varepsilon^s_A(\Sigma) | \varepsilon^s_A(e) \rangle$ where $s = \Sigma.st$.

The theorem states that for all commands e, for all traces t and configurations α such that $\langle \Sigma_0 | e \rangle \Downarrow t \cdot \alpha$ with $obs_A(\alpha)$, $\alpha = \langle \Sigma_n | e_n \rangle$ and $\Sigma_n.st = s$,

$$ek_A(\varepsilon_A(t \cdot \alpha)) \setminus ek^+_A(\varepsilon_A(t)) \subseteq I_A(e, s) \cup R_A(t, s)$$

Simple set-theoretic rearrangements give the equivalent containment:

$$\overline{ek^+_A(\varepsilon_A(t))} \cap \overline{I_A(e, s)} \cap \overline{R_A(t, s)} \subseteq \overline{ek_A(\varepsilon_A(t \cdot \alpha))}$$

Following Delft et al. [2015], we rewrite the set containment using logical connectives.

$$\begin{array}{lll}
\forall e, e', t, t', t'', t'''. & & \\
\quad \langle \Sigma_0 | e \rangle \Downarrow t \cdot \langle \Sigma_n | e_n \rangle & \Big\} & \\
\wedge \quad obs_A(\langle \Sigma_n | e_n \rangle) & \Big\} & \text{Set up} \\
\wedge \quad \Sigma_n.st = s & \Big\} & \\
\wedge \quad \langle \Sigma_0 | e' \rangle \Downarrow t' \cdot \alpha' & \Big\} & \\
\wedge \quad obs_A(\alpha') & \Big\} & ek^+_A(\varepsilon_A(t)) \\
\wedge \quad \varepsilon_A(t) = \varepsilon_A(t') & \Big\} & \\
\wedge \quad \varepsilon^s_A(e) = \varepsilon^s_A(e') & \Big\} & I_A(e, s) \\
\wedge \quad \langle \Sigma_0 | e' \rangle \Downarrow t'' \cdot \alpha'' & \Big\} & \\
\wedge \quad obs_A(\alpha'') & \Big\} & R_A(t, s) \\
\wedge \quad \varepsilon_{\mathrm{L}}(t) = \varepsilon_{\mathrm{L}}(t'') & \Big\} & \\
\Rightarrow \quad \langle \Sigma_0 | e' \rangle \Downarrow t''' & \Big\} & \\
\wedge \quad \varepsilon_A(t \cdot \langle \Sigma_n | e_n \rangle) = \varepsilon_A(t''') & \Big\} & ek(\varepsilon_A(t \cdot \alpha))
\end{array}$$

We observe that e' is deterministic, and that since for all s, $A \sqsubseteq_s A$ (reflexivity of $\sqsubseteq$), we have $A \in L$. Therefore $\varepsilon_{\mathrm{L}}(t) = \varepsilon_{\mathrm{L}}(t'')$ implies $\varepsilon_A(t) = \varepsilon_A(t'')$. Since both $obs_A(\alpha')$ and $obs_A(\alpha'')$, it has to be that $t' = t''$ and $\alpha' = \alpha''$.

Furthermore, to satisfy $\varepsilon_A(t \cdot \langle \Sigma_n | e_n \rangle) = \varepsilon_A(t''')$ we simply need that $\varepsilon_A(\langle \Sigma_n | e_n \rangle) = \varepsilon_A(\alpha')$. Unfolding α' to $\langle \Sigma_m | e_m \rangle$, this simplifies the implication to:

$$
\begin{array}{lll}
\forall e, e', t, t'. & & \\
\quad \langle \Sigma_0 | e \rangle \Downarrow t \cdot \langle \Sigma_n | e_n \rangle & \left.\begin{array}{l}\\ \\ \\ \\ \\ \\ \end{array}\right\} & \\
\wedge \quad obs_A(\langle \Sigma_n | e_n \rangle) & & \text{Set up} \\
\wedge \quad \Sigma_n.st = s & & \\
\wedge \quad \langle \Sigma_0 | e' \rangle \Downarrow t' \cdot \langle \Sigma_m | e_m \rangle & & \\
\wedge \quad obs_A(\langle \Sigma_m | e_m \rangle) & & ek_A^+(\varepsilon_A(t)) \\
\wedge \quad \varepsilon_A(t) = \varepsilon_A(t') & & \\
\wedge \quad \varepsilon_A^s(e) = \varepsilon_A^s(e') & & I_A(e,s) \\
\wedge \quad \varepsilon_{\mathrm{L}}(t) = \varepsilon_{\mathrm{L}}(t') & & R_A(t,s) \\
\Rightarrow \quad \varepsilon_A(\langle \Sigma_n | e_n \rangle) = \varepsilon_A(\langle \Sigma_m | e_m \rangle) & & ek(\varepsilon_A(t \cdot \alpha))
\end{array}
$$

Proof. We show this by induction on n in

$$\langle \Sigma_0 | e \rangle \longrightarrow^* \langle \Sigma_{n-1} | e_{n-1} \rangle \longrightarrow \langle \Sigma_n | e_n \rangle$$

- **Case** $n = 0$**:** This means that both t and t' are empty, $\Sigma_n = \Sigma_m = \Sigma_0$, $e_n = e$ and $e_m = e'$. Hence proving $\varepsilon_A(\langle \Sigma_n | e_n \rangle) = \varepsilon_A(\langle \Sigma_m | e_m \rangle)$ is equivalent to showing $\varepsilon_A(\langle \Sigma_0 | e \rangle) = \varepsilon_A(\langle \Sigma_0 | e' \rangle)$, which follows from $\varepsilon_A^s(e) = \varepsilon_A^s(e')$ (by $I_A(e,s)$).
- **Case** $n > 0$**:** We have:

  $$\langle \Sigma_0 | e \rangle \longrightarrow^* \langle \Sigma_{n-1} | e_{n-1} \rangle \longrightarrow \langle \Sigma_n | e_n \rangle$$
  $$\langle \Sigma_0 | e' \rangle \longrightarrow^* \langle \Sigma_{m-1} | e_{m-1} \rangle \longrightarrow \langle \Sigma_m | e_m \rangle$$

  By cases on the reduced expression in e_{n-1}:

  - **Case** $e_{n-1} = \mathbf{E}\ [\mathbf{setState}\ s]$**:** Considering $obs_A(\langle \Sigma_n | e_n \rangle)$, we have by the *incUpperSet* check on evaluating **setState** that also $obs_A(\langle \Sigma_{n-1} | e_{n-1} \rangle)$. By $ek_A^+(\varepsilon_A(t))$ the last A-observable configuration in trace t' must be of the form $\langle \Sigma_i | \mathbf{E}'\ [\mathbf{setState}\ s] \rangle$ and equivalent, i.e., $\varepsilon_A(\langle \Sigma_i | \mathbf{E}'\ [\mathbf{setState}\ s] \rangle) = \varepsilon_A(\langle \Sigma_{n-1} | \mathbf{E}\ [\mathbf{setState}\ s] \rangle)$. We thus have that $\Sigma_i.lset = \Sigma_{n-1}.lset$.

    Since, by $obs_A(\langle \Sigma_n | e_n \rangle)$, $\Sigma_{n-1}.lset \sqsubseteq_s A$, also the next configuration following $\langle \Sigma_i | \mathbf{E}'\ [\mathbf{setState}\ s] \rangle$ must be observable to A. Hence the next configuration is $\langle \Sigma_m | e_m \rangle$ which implies that $\langle \Sigma_i | \mathbf{E}'\ [\mathbf{setState}\ s] \rangle = \langle \Sigma_{m-1} | e_{m-1} \rangle$.

    Since $A \in L$, we also have that $\exists l \in L$ such that $obs_l(\langle \Sigma_{n-1} | e_{n-1} \rangle)$ and $obs_l(\langle \Sigma_{m-1} | e_{m-1} \rangle)$. Therefore, by $R_A(t,s)$, these last configuration must be equivalent: $\varepsilon_{\mathrm{L}}(\langle \Sigma_{n-1} | e_{n-1} \rangle) = \varepsilon_{\mathrm{L}}(\langle \Sigma_{m-1} | e_{m-1} \rangle)$. Since the only change in each configuration is setting the policy state to s, which reveals exactly information labelled with L to A, it follows that $\varepsilon_A(\langle \Sigma_n | e_n \rangle) = \varepsilon_A(\langle \Sigma_m | e_m \rangle)$.

- **Case** $e_{n-1} = \mathbf{E}\ [\mathbf{toLabeledRet}\ ls\ s\ l\ v]$**:** Since $obs_A(\langle \Sigma_n | e_n \rangle)$, the trace produced by $\langle \Sigma_0 | e \rangle$ must be of the form

$$\begin{aligned}&\langle \Sigma_0 | e \rangle \longrightarrow^* \langle \Sigma_i | \mathbf{E}\ [\mathbf{toLabeled}\ l\ m] \rangle \longrightarrow^* \\ &\langle \Sigma_{n-1} | \mathbf{E}\ [\mathbf{toLabeledRet}\ ls\ s\ l\ v] \rangle \longrightarrow \\ &\langle (ls, s, \Sigma_{n-1}.\phi) | \mathbf{E}\ [\mathbf{return}\ (\texttt{Lb}\ l\ v)] \rangle\end{aligned}$$

  where $\Sigma_i.lset = ls = \Sigma_n.lset$ and $\Sigma_i.st = s = \Sigma_n.st$, with $ls \sqsubseteq_s A$. By $ek^+_A(\varepsilon_A(t))$, we also have that the trace produced by $\langle \Sigma_0 | e' \rangle$ must be of the form

$$\begin{aligned}&\langle \Sigma_0 | e' \rangle \longrightarrow^* \langle \Sigma'_i | \mathbf{E}'\ [\mathbf{toLabeled}\ l\ m'] \rangle \longrightarrow^* \\ &\langle \Sigma_{m-1} | \mathbf{E}'\ [\mathbf{toLabeledRet}\ ls\ s\ l\ v'] \rangle \longrightarrow \\ &\langle (ls, s, \Sigma_{m-1}.\phi) | \mathbf{E}'\ [\mathbf{return}\ (\texttt{Lb}\ l\ v')] \rangle\end{aligned}$$

  where $\Sigma_i.lset = \Sigma'_i.lset$ and $\Sigma_i.st = \Sigma'_i.st$. Note that in configuration $m-1$ the evaluated expression must be **toLabeledRet**, because
  - Case $\Sigma_{n-1}.lset \sqsubseteq_{\Sigma_{n-1}.st} A$, the expression must be equal due to $ek^+_A(\varepsilon_A(t))$.
  - Case $\Sigma_{n-1}.lset \not\sqsubseteq_{\Sigma_{n-1}.st} A$, it follows that $\Sigma_{m-1}.lset \not\sqsubseteq_{\Sigma_{m-1}.st} A$ due to $ek^+_A(\varepsilon_A(t))$ (and $incUpperSet$), therefore the only way that $obs_A(\langle \Sigma_m | e_m \rangle)$ is if the expression immediately before it is **toLabeledRet**.

  We already have by $ek^+_A(\varepsilon_A(t))$ that $\varepsilon^s_A(\mathbf{E}) = \varepsilon^s_A(\mathbf{E}')$. We thus need to show that $\varepsilon^s_A(\Sigma_{n-1}.\phi) = \varepsilon^s_A(\Sigma_{m-1}.\phi)$ and that $\varepsilon^s_A(\texttt{Lb}\ l\ v) = \varepsilon^s_A(\texttt{Lb}\ l\ v')$.
  For each reference x in $\Sigma_{n-1}.\phi$, we have that reference x also exists in $\Sigma_{m-1}.\phi$ (by $ek^+_A(\varepsilon_A(t))$ on configuration i, i'). Let $\Sigma_{n-1}.\phi\ (x) = \texttt{Lb}\ l_x\ w$ and $\Sigma_{m-1}.\phi\ (x) = \texttt{Lb}\ l_x\ w'$.
  If $l_x \not\sqsubseteq_s A$ then in both configurations the value erases to $\texttt{Lb}\ l_x\ \bullet$ and we are done.
  If $l_x \sqsubseteq_s A$ then $l_x \in L$. All writes to x, if any, must have happened when $lset \sqsubseteq_{st} l_x$. By $R_A(t, s)$, this means that $\varepsilon^{st}_{\mathrm{L}}(w) = \varepsilon^{st}_{\mathrm{L}}(w')$. Regardless of the value of st, this means that $\varepsilon^s_A(w) = \varepsilon^s_A(w')$. Since this holds for each reference, $\varepsilon^s_A(\Sigma_{n-1}.\phi) = \varepsilon^s_A(\Sigma_{m-1}.\phi)$.
  For the returned value $\texttt{Lb}\ l\ v$ resp. $Lv\ l\ v'$, if $l \not\sqsubseteq_s A$ then both returned values erase to $\texttt{Lb}\ l\ \bullet$.
  If $l \sqsubseteq_s A$, this means that $l \in L$. As the rule for **toLabeledRet** requires $\Sigma_{n-1}.lset \sqsubseteq_{\Sigma_{n-1}.st} l$, this means that (by $R_A(t, s)$) $\varepsilon_{\mathrm{L}}^{\Sigma_{n-1}.st}(v) = \varepsilon_{\mathrm{L}}^{\Sigma_{m-1}.st}(v')$. Again, regardless of the used policy states, this means that $\varepsilon^s_A(v) = \varepsilon^s_A(v')$.
- **All other cases:** For the reductions in which the policy state is unchanged, we have by $obs_A(\langle \Sigma_n | e_n \rangle)$ and $obs_A(\langle \Sigma_m | e_m \rangle)$ that also $obs_A(\langle \Sigma_{n-1} | e_{n-1} \rangle)$ and $obs_A(\langle \Sigma_{m-1} | e_{m-1} \rangle)$.

By $ek^{+}_{A}(\varepsilon_A(t))$ we have that $\varepsilon_A(\langle \Sigma_{n-1}|e_{n-1}\rangle) = \varepsilon_A(\langle \Sigma_{m-1}|e_{m-1}\rangle)$. By the Fixed-State Lemma 1 (below), this gives us $\varepsilon_A(\langle \Sigma_n|e_n\rangle) = \varepsilon_A(\langle \Sigma_m|e_m\rangle)$.

□

Lemma 1 (Fixed-State Lemma). *Given two single-step evaluations* $\langle \Sigma_1|e_1\rangle \longrightarrow \langle \Sigma_1'|e_1'\rangle$ *and* $\langle \Sigma_2|e_2\rangle \longrightarrow \langle \Sigma_2'|e_2'\rangle$ *with* $\Sigma_1.st = \Sigma_1'.st$ *and* $\Sigma_1.lset \sqsubseteq_s A$. *For all attacker levels A, if* $\varepsilon_A(\langle \Sigma_1|e_1\rangle) = \varepsilon_A(\langle \Sigma_2|e_2\rangle)$ *then* $\varepsilon_A(\langle \Sigma_1'|e_1'\rangle) = \varepsilon_A(\langle \Sigma_2'|e_2'\rangle)$.

Proof. Let $s = \Sigma_1.st$. Since $\Sigma_1.lset \sqsubseteq_s A$ also $\Sigma_2.lset \sqsubseteq_s A$ and therefore the next redex in e_1 and e_2 is equal and their evaluation contexts are equal after erasure. We thus have that $\varepsilon^s_A(e_1') = \varepsilon^s_A(e_2')$. By cases on the next redex.

- **Case redex is label** $l\ v$**, writeLIORef** $r\ v$**, newLIORef** $l\ v$. In all cases, the values v, l and r appear unlabeled in the redex and are therefore equal in e_1 and e_2. Thus, changes to Σ_1 are equal to changes to Σ_2. It therefore follows that $\varepsilon_A(\langle \Sigma_1'|e_1'\rangle) = \varepsilon_A(\langle \Sigma_2'|e_2'\rangle)$.
- **Any other redex**. In these evaluation steps the state component is not used, and the result follows directly from Lemma 1 in the formalisation of LIO with static policies Stefan et al. [2011].

□

B DLM encoding

Exported API The complete enconding of the DLM can be found in Figure 11. A DLM label consists of multiple components (*Comp*), and each component consists of an owner principal and a set (list) of readers. As discussed in § 6.7, the policy state component contains a boolean indicating whether declassification is allowed, a list of the principals who authorised this computation to perform declassifications for them, and the acts-for hierarchy (denoted $\succeq$), stored as a list of acts-for relations.

The *Label* instance for this label and state implements the required $\sqsubseteq$ and *incUpperSet* functions. If the declassification-component of the state is set to *True*, this means that the declassification check was successful so any relabelling is allowed (see the function *declassify*). Otherwise, $l_1 \sqsubseteq_s l_2$ if every owners concern in label l_1 is reflected in label l_2, which matches the relabeling rule from Myers and Liskov [1998]. For *incUpperSet*, clearly the decision to declassify increase the upper set from any label. Effectively, this means that user code written against this DLM encoding should never have unlabeled data in scope when declassifying. The only other change to the state by user code is changing the acts-for hierarchy. This increases the upper set of label l if for any of the principals p in l there exists a principal

```
type Princ = String
data Comp = Comp Princ [Princ]
type DLMLabel = [Comp]
type DLMState =
  (Bool                 – Allowing declassification.
  ,[Princ]              – Computation's authority.
  ,[(Princ, Princ)]     – Acts-for hierarchy.
  )
instance Label DLMLabel DLMState where
  l1 ⊑s l2 = case s of
    (True  ,_,_ ) ⇒ True
    (False ,_,≽ ) ⇒
      ∀ Comp o1 rs1 ∈ l1
        ∃ Comp o2 rs2 ∈ l2
          o2 ≽ o1 ∧ ∀ r2 ∈ rs2 ∃ r1 ∈ rs1 r1 ≽ r2
  incUpperSet (False,_,_) (True,_,_) l = True
  incUpperSet (_,_,≽O) (_,_,≽N) l =
    ∃ p ∈ l.∃ p'.p' ⋡O p ∧ p' ≽N p
type DLM = SLIO DLMLabel DLMState


declassify :: Labeled DLMLabel a → DLMLabel
           → DLM (Maybe (Labeled DLMLabel a))
declassify lv l = do
  s ← getState
  let (_, auth, ≽) = s
  let checkAuth = l ⧺ { Comp p ∅ | p ∈ auth }
  if (labelOf lv) ⊑s checkAuth
    then do (res ← toLabeled (do
              setState (True, auth, ≽)
              v ← unlabel lv
              return v)
              return (Just res))
    else return Nothing


addActsFor :: (Princ, Princ) → LIO DLM ()
addActsFor rel = do
  (d, a, ≽) ← getState
  putState (d, a, rel: ≽)
removeActsFor :: (Princ, Princ) → LIO DLM ()
removeActsFor rel = do
  (d, a, ≽) ← getState
  setState (d, a, ≽ \\ [rel])


runDLM ≽ auth comp = runLIO comp (LIOState
  {lcur = ∅, st = (False, auth, ≽), ϕ = ∅})
```

Fig. 11: Encoding DLM in SLIO.

p' that did not act for p in the old hierarcy $\succeq_O$, but does so in the new hierarchy $\succeq_N$.

The function *declassify* attempts to relabel a labeled value lv with the provided label l. Since declassification needs to be permitted by the right principals in the authority of the computation, the function constructs the label $checkAuth$ to check this. The label $checkAuth$ is defined by adding $p : \emptyset$ for each principal $p \in auth$, effectively removing the concerns for these principals (conform L_A in Myers and Liskov [1997]). If the computation does not have sufficient authority, the option type *Maybe* is used to return the value *Nothing* instead of the relabeld value.

The DLM encoding only provides restricted state modification operations (*addActsFor* and *removeActsFor*), using Haskell's encapsulation to prevent user code from declassifying information without using the exported *declassify* function. To give a complete guarantee on this, Safe Haskell should be used Terei et al. [2012].

Finally, the *runDLM* function starts a computation with the specified authority and initial acts-for hierarchy.

Example We consider a simple program that combines information of employees Bob and Carl in one document. While combining the information, we explicitly include Dave as a reader for this document. The acts-for hierarchy between all the employees in the company is displayed in Figure 12.

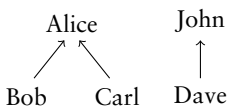

Fig. 12: Acts-for hierarchy.

The function *addInfo emp doc* in Figure 13 creates some data labeled with $\{emp:\}$ and adds it to the document *doc* (using the unspecified function *combine*). It then declassifies this document to include Dave as a reader.

The *example* code calls this function on an intially empty document for both Bob and Carl. It is therefore important that *main* calls this function with their authority otherwise the declassifications would fail. After adding both employees' information, the label on *doc* has become {Bob:Dave; Carl: Dave}.

Since Alice acts for both Bob and Carl, she can claim ownership over this report. And because John, acting for Dave, was already implicitly a reader of the data, the relabeling to {Alice : John} succeeds, without requiring any authority or declassification. Were we instead to uncomment the line removing the acts-for relation between John and Dave, the relabeling would

become invalid. The encoding and example serve to illustrate that SLIO easily captures the dynamic nature of the DLM.

```
addInfo emp doc = do
  let dl = labelOf doc
  d  ← label {emp:} (emp ++ "'s data.")
  d' ← toLabeled ({emp:} ++ dl) (combine d doc)
  (Just d'') ← declassify d' ({emp : Dave} ++ dl)
  return d''

example = do
  doc ← label {} ""
  doc ← addInfo Bob doc
  doc ← addInfo Carl doc
    – delActsFor (John,Dave)
  doc ← relabel {Alice : John} doc

main = runDLM ⪰ {Bob, Carl} example
```

Fig. 13: Using the DLM encoding.

C Other Relabeling Release Definitions

Our definition of Relabeling Release (Definition 4) is not arbitrary. In this appendix we list a number of different definitions that, although sounding reasonable at first glance, do not match our intuition of what is released via relabelings.

C.1 Release knowledge by A and s

Consider an attacker A_s who also makes observations on level A, but pretending that the policy state was s for the duration of the whole execution. To allow the attacker A to learn information resulting from relabelings to levels $l \sqsubseteq_s A$, we share the knowledge that A_s has gained so far:

$$R_A(t,s) = \{e' \mid \neg\exists t', \alpha'.\langle \Sigma_0 | e' \rangle \Downarrow t' \cdot \alpha' \text{ with } obs_A(\alpha') \\ \text{and } \varepsilon^s_A(t) = \varepsilon^s_A(t')\}$$

Here, $\varepsilon^s_A(t)$ fixes the policy state s to consider already at the level of the trace, ignoring the actual state in each configuration:

$$\varepsilon^s_A(\langle \Sigma | e \rangle \cdot t) = \begin{cases} \langle \varepsilon^s_A(\Sigma) | \varepsilon^s_A(e) \rangle \cdot \varepsilon^s_A(t) & \text{if } \Sigma.lset \sqsubseteq_s A \\ \varepsilon^s_A(t) & \text{otherwise} \end{cases}$$

Although this does release the relabeling information from our example program in Figure 9, it does not allow all relabelings that we would intuitively mark secure. As an example, consider the following program (in the dynamic policy *User* setting from § 6.3):

```
setState [(Bob, Carl)]
_ ← toLabeled Bob (do
  setState [(Alice, Bob)]
  a ← readLIORef aliceRef
  writeLIORef bobRef a)
```

When returning from this **toLabeled** computation, Carl learns the information that is in *aliceRef* since the current state allows him to see Bob's data, thus *bobRef*. We would argue that this is secure, since Bob learns Alice's data in a state where this was allowed, and Carl in turn learns Bob's (and thereby Alice's data via relabeling) in a state where this is allowed. However, the suggested set $R_A(t, s)$ does not release the value of *aliceRef* to Carl.

Consider the observer Carl_s who observes as if the policy state is always [(Bob, Carl)]. After the instruction **readLIORef** *aliceRef* the current label becomes {Alice}, meaning that this and all configuration to the end of the **toLabeled** computation are not visible to Carl_s. Hence, Carl_s does not learn the value of *aliceRef* and this is therefore not released to Carl.

C.2 Release knowledge by A, s and *lset*

As a possible correction to the A_s attacker, we could consider the A_s^{lset} attacker who also makes observations on level A, but pretending that the policy state was s *and* the current label set was *lset* for the duration of the whole execution. Here *lset* is the current label set when the new observation α was produced – i.e. this attacker fixes all the policy-relevant components. To allow the attacker A to learn information resulting from relabelings to levels $l \sqsubseteq_s A$, we share the knowledge that A_s^{lset} has gained so far:

$$R_A(t, s) = \{e' \mid \neg\exists t', \alpha'.\langle \Sigma_0 | e' \rangle \Downarrow t' \cdot \alpha' \text{ with } obs_A(\alpha') \\ \text{and } \varepsilon_A^{s,lset}(t) = \varepsilon_A^{s,lset}(t')\}$$

Here, $\varepsilon_A^{s,lset}(t)$ fixes the policy state s amd the current label set, ignoring their actual values in each configuration. Hence since $obs_A(\alpha)$, all previous configurations are observable:

$$\varepsilon_A^{s,lset}(\langle \Sigma | e \rangle \cdot t) = \langle \varepsilon_A^s(\Sigma) | \varepsilon_A^s(e) \rangle \cdot \varepsilon_A^{s,lset}(t)$$

This indeed allows the secure program from § C.1, but also labels the following program secure, which we argued in § 6.4 to be clearly insecure due to conditionally allowing the flow from *High* to *Low*:

```
leak highData = do
  r ← newLIORef Low 1
  _ ← toLabeled High (do
    h ← unlabel highData
    when (h ≡ 0) (do
      setState True
      writeLIORef r 0
  v ← readLIORef r))
  return v
```

Although the Low_s^{lset} attacker does not observe the value of h when it is not 0, still this attacker learns that since the next expression to reduce after **unlabel** is **toLabeledRet**, that the value of h was not 0. This is exactly the information leaked to *Low*, so this release policy allows for that leak.

C.3 Release knowledge by all $l \sqsubseteq_s A$

Finally we consider one definition that is close to the one we selected. Rather than defining the multi-level erasure function $\varepsilon_{\mathrm{L}}(\cdot)$, we could say that we release the knowledge for each level $l \sqsubseteq_s A$ individually:

$$R_A(t,s) = \{e' \mid \neg\exists t', \alpha'.\langle \Sigma_0 | e' \rangle \Downarrow t' \cdot \alpha' \text{ with } obs_A(\alpha') \\ \text{and } \varepsilon_l(t) = \varepsilon_l(t') \text{ for all } l \sqsubseteq_s A\}$$

This does disallow the leak via policy state change and it allows for the relabel examples shown in this paper so far. However, it does not consider the following program secure, which we would intuitively label as such:

```
setState []
one ← label Alice 1
two ← label Alice 2
bobData ← toLabeled Bob (do
  setState [(Carl, Bob)]
  d ← unlabel carlData
  return (if d then one else two))
daveData ← toLabeled Dave (do
  setState [(Bob, Dave)]
  d ← unlabel bobData
  return d)
setState [(Alice, Dave)]
```

Returning from the second **toLabeled** command, information from Alice is not allowed to flow to Dave, so the value of *daveData* as observed by Dave is (Lb Dave (Lb Alice •)). After the last **setState** command Dave learns that the value at • was either 1 or 2, and from that gains knowledge

about the value in *carlData*. The problem with the suggested definition is that it does allow Dave to learn this information, but not at the right point in the execution!

When unlabeling *bobData*, information may flow from Bob to Dave. Hence, at this point $R_A(t, s)$ allows Dave to learn what Bob has learned, which include the earlier observation of the value in *carlData*. However, Dave only sees (Lb Alice $\bullet$) and does not learn this information yet. When Dave does learn the information, the state only allow information from Alice to flow to Dave. Alice has not been able to observe any configuration where *carlData* was unlabeled, so sharing Alice's knowledge with Dave does not allow Dave to learn anything about *carlData*.

The final definition for $R_A(t, s)$ given in Definition 4 resolves this by combining the observations from all levels $l \sqsubseteq_s A$ at each point. With $\mathrm{L} = \{\mathrm{Alice}, \mathrm{Dave}\}$ the projection $\varepsilon_{\mathrm{L}}(t)$ contains **unlabel** *bobData* which releases whether the value labeled with Alice is 1 or 2, as we desired.

CHAPTER
SEVEN

IT'S MY PRIVILEGE: CONTROLLING DOWNGRADING IN DC-LABELS

Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, Alejandro Russo

Abstract. Disjunction Category Labels (DC-labels) are an expressive label format used to classify the sensitivity of data in information-flow control systems. DC-labels use capability-like *privileges* to downgrade information. Inappropriate use of privileges can compromise security, but DC-labels provide no mechanism to ensure appropriate use. We extend DC-labels with the novel notions of *bounded privileges* and *robust privileges*. Bounded privileges specify and enforce upper and lower bounds on the labels of data that may be downgraded. Bounded privileges are simple and intuitive, yet can express a rich set of desirable security policies. Robust privileges can be used only in downgrading operations that are *robust*, i.e., the code exercising privileges cannot be abused to release or certify more information than intended. Surprisingly, robust downgrades can be expressed in DC-labels as downgrading operations using a weakened privilege. We provide *sound and complete* run-time security checks to ensure downgrading operations are robust. We illustrate the applicability of bounded and robust privileges in a case study as well as by identifying a vulnerability in an existing DC-label-based application.

7.1 Introduction

Information-flow control (IFC) systems track the flow of information by associating *labels* with data. Disjunction Category Labels (DC-labels) are a practical and expressive label format that can capture the security concerns of principals. IFC systems and DC-labels can provide strong, expressive, and practical information security guarantees, preventing exploitation of, for example, cross-site scripting and code injection vulnerabilities [Stefan et al., 2014, Giffin et al., 2012, Sabelfeld and Myers, 2003, Zeldovich et al., 2006, Krohn et al., 2007].

IFC systems often need to *downgrade* information: *declassification* downgrades confidentiality, and *endorsement* downgrades integrity. Downgrading of DC-labels occurs via operations that require unforgeable capability-like tokens known as *privileges*. Unfortunately, DC-labels offer no methodology to protect developers from the *discretionary* (i.e., unrestricted) exercise of privileges—even a minor mistake in handling privileges can compromise the whole system's security. For example, we found a one-line vulnerability in an existing DC-label application written by experts that enabled confidential information to be inappropriately released, thus violating the application's intended security properties.

To address this, we introduce *restricted privileges*: privileges that are limited in their ability to declassify and endorse information. By declaratively restricting the use of privileges, developers can reason about the security properties of the system, regardless of the code that may possess or use the restricted privileges. Thus, the developer's local declaration of restrictions enables the enforcement of global information security guarantees.

We present two kinds of restricted privileges: *bounded privileges* and *robust privileges*. A bounded privilege imposes upper and lower bounds on the DC-labels of data that is declassified or endorsed using that privilege. Robust privileges avoid the accidental or malicious use of privileges to declassify or endorse more information than intended, achieving a property known as *robustness* [Zdancewic and Myers, 2001, Myers et al., 2006].

Bounded Privileges. A bounded privilege wraps an unrestricted privilege with two *immutable* labels that indicate upper and lower bounds for downgrading. DC-labels form a lattice structure (described in Section 7.2), and thus a bounded privilege restricts where in the lattice downgrading may occur. A bounded privilege also has a *mode*, indicating whether the bounded privilege may be used for declassification, endorsement, or both declassification and endorsement.

In terms of confidentiality, the upper bound limits the confidentiality of information that can be declassified using the privilege, and the lower bound limits the confidentiality of the information after declassification. For example, suppose principal `fb.com` passes a bounded privilege to `gogl.com`.

If the lower bound of the bounded privilege is the label "`gogl.com`" then the privilege can be used to declassify information only from `fb.com` to `gogl.com`. Even if `gogl.com` passes the bounded privilege to another domain, say `evil.com`, the bounded privilege cannot be used to declassify information from `fb.com` to `evil.com`.

In terms of integrity, the upper bound of a bounded privilege indicates the least trustworthy level of information the privilege can be used to endorse, and the lower bound limits the integrity of the information after endorsement. For example, by setting the upper bound appropriately, `fb.com` can create a bounded privilege that can be used to endorse data only from `gogl.com`, and cannot be used to endorse other data, say from `evil.com`.

Robust Privileges. The security of a system might be at risk if an attacker is able to influence the decision to declassify or endorse information, or can influence what information is declassified. For example, consider a routine that receives a secret pair `(username,password)` and uses a privilege to declassify the first component of the pair. If an attacker (from another system component) can influence the pair to be `(password,username)` and trigger the declassification, the password will be leaked.

Robust declassification [Zdancewic and Myers, 2001] and *qualified robustness* [Myers et al., 2006] are end-to-end semantic security guarantees that ensure that attackers are unable to inappropriately influence what information is revealed to them. These security conditions can be enforced by restricting declassification and endorsement operations. A robust privilege wraps a privilege and ensures that it is used only in declassification and endorsement operations that satisfy appropriate robustness checks.

This paper makes the following contributions: (i) We introduce bounded and robust privileges to limit the exercise of privileges for declassification and endorsement. (ii) We present a semantic characterization of how bounded privileges and robust privileges restrict declassification and endorsement operations. (iii) We define run-time security checks for bounded privileges and robust privileges that soundly and completely enforce the semantic characterization of restricted downgrading operations. The run-time checking for robust downgrading is effectively a weakening of the underlying unrestricted privilege: a surprisingly simple characterization of robustness. (iv) We illustrate the applicability of bounded and robust privileges via a case study. Moreover, use of restricted privileges identified a vulnerability in an existing DC-label-based application.

This paper is organized as follows. Section 7.2 introduces the DC-label model. Section 7.3 characterizes downgrading operations that use restricted privileges, and Section 7.4 provides the corresponding enforcement. Section 7.5 describes security properties in the presence of multiple restricted privileges. Case studies are given in Section 7.6. Section 7.7 examines related work and Section 7.8 concludes.

7.2 Background

We briefly define three concepts fundamental to our presentation: the DC-label model, privileges, and floating label systems.

Label Lattice DC-labels [Stefan et al., 2011a] are pairs of confidentiality and integrity policies. Confidentiality polices describe who may learn information. Integrity polices describe who takes responsibility or vouches for information. Both confidentiality and integrity policies are positive propositional formulas in conjunctive normal form, where propositional constants represent *principals*. Let CNF denote the set of all positive propositional formulas in conjunctive normal form; we use the term *formula* to range over CNF. We assume that operations on formulas always reduce their results to conjunctive normal form.

$$
\begin{array}{rcl}
C_1 \sqsubseteq^{\mathrm{C}} C_2 & \iff & C_2 \Rightarrow C_1 \\
C_1 \sqcup^{\mathrm{C}} C_2 & \iff & C_1 \wedge C_2 \\
C_1 \sqcap^{\mathrm{C}} C_2 & \iff & C_1 \vee C_2 \\
\bot^{\mathrm{C}} \equiv True & & \top^{\mathrm{C}} \equiv False
\end{array}
$$

Fig. 1: Confidentiality Lattice

$$
\begin{array}{rcl}
I_1 \sqsubseteq^{\mathrm{I}} I_2 & \iff & I_1 \Rightarrow I_2 \\
I_1 \sqcup^{\mathrm{I}} I_2 & \iff & I_1 \vee I_2 \\
I_1 \sqcap^{\mathrm{I}} I_2 & \iff & I_1 \wedge I_2 \\
\bot^{\mathrm{I}} \equiv False & & \top^{\mathrm{I}} \equiv True
\end{array}
$$

Fig. 2: Integrity Lattice

Both confidentiality policies and integrity policies form lattices—see Figures 1 and 2. We interpret $C_1 \sqsubseteq^{\mathrm{C}} C_2$ as: C_2 is at least as confidential as C_1. For instance, Alice $\vee$ Bob $\sqsubseteq^{\mathrm{C}}$ Alice, which means that data readable by either Alice or Bob is less confidential than data readable only by Alice. Conjunctions of principals represent the multiple interest of principals to protect the data. Conversely, disjunctions of principals represent groups wherein any member may learn the information. The integrity lattice is dually defined [Biba, 1977]; we interpret $I_1 \sqsubseteq^{\mathrm{I}} I_2$ as: I_1 is at least as trustworthy as I_2. For example, Alice $\wedge$ Bob $\sqsubseteq^{\mathrm{I}}$ Alice, which indicates that data vouched for by Alice $\wedge$ Bob is more trustworthy than data vouched for only by Alice. In this case, conjunctions of principals represent groups whose members are independently responsible for the information. For example, data with integrity Alice $\wedge$ Bob means that Alice is completely responsible for the data, and so is Bob. Conversely, disjunctions of principals represent groups that collectively take responsibility for the information, however, no principal takes sole responsibility. For example, data with integrity Alice $\vee$ Bob means that Alice and Bob collectively are responsible for the data, i.e., both may have contributed to, or influenced the computation of the data.

Formally, a DC-label is a pair of a confidentiality policy C and an integrity policy I, written $\langle C, I \rangle$. DC-labels form a product lattice given in Figure 3. The $\sqsubseteq$ relation is called the *can-flow-to* relation because it describes information flows that respect confidentiality and integrity policies.

$$\begin{array}{rcl}
\langle C_1, I_1\rangle \sqsubseteq \langle C_2, I_2\rangle & \iff & C_1 \sqsubseteq^{\mathrm{C}} C_2 \text{ and } I_1 \sqsubseteq^{\mathrm{I}} I_2 \\
\langle C_1, I_1\rangle \sqcup \langle C_2, I_2\rangle & \equiv & \langle C_1 \sqcup^{\mathrm{C}} C_2, I_1 \sqcup^{\mathrm{I}} I_2\rangle \\
\langle C_1, I_1\rangle \sqcap \langle C_2, I_2\rangle & \equiv & \langle C_1 \sqcap^{\mathrm{C}} C_2, I_1 \sqcap^{\mathrm{I}} I_2\rangle \\
\mathrm{C}(\langle C, I\rangle) \equiv C & & \mathrm{I}(\langle C, I\rangle) \equiv I
\end{array}$$

Fig. 3: Security lattice for DC-labels

$$\begin{array}{rcl}
\langle C_1, I_1\rangle \sqsubseteq_p \langle C_2, I_2\rangle & \iff & C_1 \sqsubseteq_p^{\mathrm{C}} C_2 \text{ and } I_1 \sqsubseteq_p^{\mathrm{I}} I_2 \\
\text{where} \quad C_1 \sqsubseteq_p^{\mathrm{C}} C_2 & \iff & C_1 \sqsubseteq^{\mathrm{C}} C_2 \sqcup^{\mathrm{C}} p \\
I_1 \sqsubseteq_p^{\mathrm{I}} I_2 & \iff & I_1 \sqcap^{\mathrm{I}} p \sqsubseteq^{\mathrm{I}} I_2
\end{array}$$

Fig. 6: Relation can-flow-to-with-privilege-p

We write $\mathrm{C}(\cdot)$ and $\mathrm{I}(\cdot)$ for the projection of confidentiality and integrity components, respectively.

Downgrading In the DC-label model, one security policy *downgrades* to another security policy if they do not satisfy the can-flow-to relation. Consider the pair of security labels in Figure 4. The first security label enforces the policy that data is vouched for by Charlie. The second security label enforces the policy that data is vouched for by Charlie and Alice, therefore a secure system cannot permit data to flow from the sources protected by the first policy to sinks protected by the second policy. This downgrade is an *endorsement*, since it downgrades only integrity, i.e., it makes a value more trustworthy. Dually, a *declassification* downgrades only confidentiality, i.e., it makes a value less confidential. Consider the pair of security labels in Figure 5: The first security label enforces the policy that data is confidential to Alice $\wedge$ Bob. The second security label enforces that data is confidential to Bob. Permitting data to flow from a source protected by the first policy to a sink protected by the second policy violates the confidentiality expectations of the source.

$$\langle \mathsf{Alice}, \mathsf{Charlie}\rangle \not\sqsubseteq \langle \mathsf{Alice}, \mathsf{Charlie} \wedge \mathsf{Alice}\rangle$$

Fig. 4: Downgrading integrity

$$\langle \mathsf{Alice} \wedge \mathsf{Bob}, \mathsf{Charlie}\rangle \not\sqsubseteq \langle \mathsf{Bob}, \mathsf{Charlie}\rangle$$

Fig. 5: Downgrading confidentiality

Privileges Practical systems must permit some downgrading. The DC-label model controls downgrading with *privileges*, where every principal has an associated privilege, and a principal's privilege enables downgrading. More precisely, given principal p, the *can-flow-to-with-privilege-p* relationship, written $\sqsubseteq_p$, describes the information flows permitted with p's privilege—see Figure 6. Observe that both downgrading examples from the previous section are now permitted by the can-flow-to-with-privilege relationship

for the principal Alice, i.e., $\langle \mathsf{Alice}, \mathsf{Charlie} \rangle \sqsubseteq_{\mathsf{Alice}} \langle \mathsf{Alice}, \mathsf{Charlie} \wedge \mathsf{Alice} \rangle$ and $\langle \mathsf{Alice} \wedge \mathsf{Bob}, \mathsf{Charlie} \rangle \sqsubseteq_{\mathsf{Alice}} \langle \mathsf{Bob}, \mathsf{Charlie} \rangle$.

Floating label systems DC-labels are usually part of *floating label systems* like LIO [Stefan et al., 2011b], Hails [Giffin et al., 2012], and COWL [Stefan et al., 2014]. Such systems associate a *current label*, L_{pc}, with every computational task—this label plays a role similar to the *program counter* (PC) in more traditional language-based IFC approaches [Sabelfeld and Myers, 2003]. The current label denotes the fact that a computation depends only on data with labels bounded above by L_{pc}. When a task with current label L_{pc} observes information with label L_A, the current label after observation, L'_{pc}, "floats" above both the previous current label and the observed information's label, i.e., $L'_{pc} = L_{pc} \sqcup L_A$. Importantly, and to respect the security lattice, the current label restricts the subsequent writes to communication channels. Specifically, a task with current label L_{pc} is prevented from writing to channels protected by policy L_A if $L_{pc} \not\sqsubseteq L_A$.

Floating-label systems typically use some run-time representation of principals' privilege, and downgrading operations require the run-time representation of a principal p's privilege to be presented in order to use the can-flow-to-with-privilege-p relation, $\sqsubseteq_p$. Thus, the run-time representation of a principal's privilege acts like a capability to downgrade that principal's information. We write $p^{\Large\text{♕}}$ for the run-time representation of the privilege of principal p, and refer to this value as a *raw privilege* (to contrast it with the restricted privileges that we introduce in this paper).

7.3 Security Definitions

If a system contains $p^{\Large\text{♕}}$, then downgrading of data with policies involving p depends entirely on how $p^{\Large\text{♕}}$ is used in the system. Reasoning about what downgrading occurs may require reasoning about global properties of the system. Indeed, we found a vulnerability in a Hails example application [Giffin et al., 2012] of a web-based rock-paper-scissors game where use of a raw privilege was localized to one component, but arbitrary data could be passed to this component to be downgraded. This motivates our work to restrict privileges, and enable local reasoning about downgrading that may occur in a system.

A *restricted privilege* is a raw privilege "wrapped" with limitations on its use. These limitations enable sound reasoning about the downgrading that may be performed using the restricted privilege, even if arbitrary code uses the restricted privilege. Thus, local reasoning that ensures $p^{\Large\text{♕}}$ is always appropriately restricted provides global guarantees about the downgrading that can occur with respect to policies involving p.

We present two kinds of restricted privileges, *bounded privileges* and *robust privileges*, which provide simple declarative limitations on the use of raw privileges.

Bounded Privileges A bounded privilege wraps a raw privilege with *downgrading bounds* and a downgrading mode. A downgrading bound is a pair of security lattice labels L_{high} and L_{low} that provide upper and lower bounds on downgrading, and the mode indicates whether the bounded privilege can be used to both declassify and endorse, only to declassify, or only to endorse.

Definition 1 (Downgrading bounds). *An operation that downgrades from security policy L_{from} to security policy L_{to} in a computational context with current label L_{pc}* satisfies downgrading bounds *L_{high} and L_{low} if and only if $(L_{from} \sqcup L_{pc}) \sqsubseteq L_{high}$ and $L_{low} \sqsubseteq (L_{to} \sqcup L_{pc})$*

Definition 2 (Bounded privileges). *A* bounded privilege with bounds L_{high} and L_{low} and mode m on privilege $p^{\Large\text{♕}}$*, written* ${}^{m}[p^{\Large\text{♕}}]_{L_{low}}^{L_{high}}$*, can be used only for downgrading operations with privilege $p^{\Large\text{♕}}$ that satisfy downgrading bounds L_{high} and L_{low}. Mode m is one of* de*,* d*, or* e*. Declassification operations are permitted only if the mode is* de *or* d*; endorsement operations are permitted only if the mode is* de *or* e*.*

Figure 7 shows a visualization of bounded downgrading. The security lattice on the left is overlaid with a visualization of where bounded downgrading can occur (shaded) with respect to bounds L_{high} and L_{low}. The security lattice on the right shows an example of what labeled values can be declassified (shaded) with a bounded declassification privilege with bounds $L_{high} = \langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A} \vee \mathsf{B} \rangle$ and $L_{low} = \langle \mathsf{A} \vee \mathsf{B}, \mathsf{A} \wedge \mathsf{B} \rangle$.

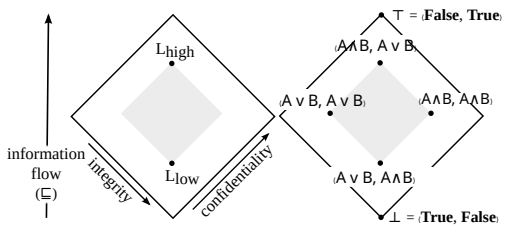

Fig. 7: Bounded Downgrading

In essence, the confidentiality lattice has collapsed $\mathsf{c}(L_{high})$ and $\mathsf{c}(L_{low})$ and all points in between: information that has confidentiality up to $\mathsf{c}(L_{high})$ may be declassified to confidentiality $\mathsf{c}(L_{low})$—all other points in the confidentiality lattice are not affected. Guarantees for endorsement with respect to bounded privileges are similar, but for integrity instead of confidentiality.

Example 1 (Policy: Only Bob controls Alice's privilege). Principal Alice allows Bob to declassify her data provided that Bob vouches for the data and the decision to declassify. In other words, information labeled with Alice can be declassified only after endorsement by Bob. This property can be captured by a bounded privilege with mode d and bounds: $L_{high} = \langle \top^{\mathsf{c}}, \mathsf{Bob} \rangle, L_{low} = \langle \bot^{\mathsf{c}}, \mathsf{Bob} \rangle$. If the privilege is used to declassify information that is not endorsed by Bob or in a context where the current label is not endorsed by Bob, then the declassification fails. In general, data must be vouched for by Bob (e.g., by using $\mathsf{Bob}^{\Large\text{♕}}$ or another restricted

privilege) before the bounded privilege for Alice can be used. For example, if a computational task has a current label $L_{pc} = \langle \mathsf{Alice}, \mathsf{Bob} \vee \mathsf{Charlie} \rangle$, the current label must be endorsed by Bob first. By endorsing the current label, Bob effectively vouches for any influence Charlie may have had on the computational task.

Example 2 (Policy: "A close source said..."). The bounded privilege given as ${}^{\mathsf{d}}[\mathsf{Alice}^{\Cheese}]^{\langle \top^{\mathrm{C}}, \top^{\mathrm{I}} \rangle}_{\langle \bot^{\mathrm{C}}, \top^{\mathrm{I}} \rangle}$ requires that the integrity of data being declassified is $\top^{\mathrm{I}}$, i.e., data that no principal takes responsibility for. Alice may wish to impose this restriction on declassification involving data confidential to her to ensure that she has plausible deniability regarding the source of the data released. That is, the bounded privilege can not be used to declassify data for which Alice is explicitly responsible.

Robust Privileges *Robustness* [Zdancewic and Myers, 2001, Myers et al., 2006] is a semantic security condition that limits downgrading based on which principals might benefit from the downgrading, and which principals have influenced the data to downgrade and the decision to downgrade.

Consider a declassification of information from a source protected by label L_{from} to a sink protected by label L_{to}. A formula A (representing a principal or party of principals) will benefit from the declassification if A cannot read from the source, but can read the sink, i.e., $\mathrm{C}(L_{from}) \not\sqsubseteq^{\mathrm{C}} A$ and $\mathrm{C}(L_{to}) \sqsubseteq^{\mathrm{C}} A$. A robust declassification does not permit any principal that benefits from it to influence either the decision to declassify or the data to declassify. A influences the decision to declassify if $A \sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{pc})$, and A influences the data to declassify if $A \sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{from})$.

Definition 3 (Robust declassification). *A* robust declassification *using privilege $p^{\Cheese}$ from a source protected by L_{from} to a sink protected by L_{to}, in a computational context with current label L_{pc} is a declassification (i.e., $\mathrm{C}(L_{from}) \sqsubseteq^{\mathrm{C}}_{p} \mathrm{C}(L_{to})$) where $\forall A \in CNF.\, \mathrm{C}(L_{to}) \sqsubseteq^{\mathrm{C}} A \wedge \mathrm{C}(L_{from}) \not\sqsubseteq^{\mathrm{C}} A \Rightarrow A \not\sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{pc}) \wedge A \not\sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{from})$.*

For endorsement, a principal benifits if it may be held responsible for information from the source but is not held responsible for information from the sink. In other words, A benefits from an endorsement if A gets absolved of responsibility for a value, i.e., $A \sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{from}) \wedge A \not\sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{to})$. Robust endorsement does not permit principals that benefit from it to influence the decision to endorse.

Definition 4 (Robust endorsement). *A* robust endorsement *using privilege $p^{\Cheese}$ from a source protected by L_{from} to a sink protected by L_{to}, in a computational context with current label L_{pc} is an endorsement (i.e., $\mathrm{I}(L_{from}) \sqsubseteq^{\mathrm{I}}_{p} \mathrm{I}(L_{to})$) where $\forall A \in CNF. A \sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{from}) \wedge A \not\sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{to}) \Rightarrow A \not\sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{pc})$.*

A *robust privilege* is a privilege that can only be used for robust downgrading operations.

Definition 5 (Robust privilege). *A* robust privilege with mode m on privilege $p^{\text{♕}}$, *written* $rbst^m\{p^{\text{♕}}\}$, *restricts downgrading operations where it is used to those that are robust for* $p^{\text{♕}}$. *Mode* m *is one of* de, d, *or* e. *Declassification operations are permitted only if the mode is* de *or* d; *endorsement operations are permitted only if the mode is* de *or* e.

The definitions of robust declassification and endorsements both quantify over all formulas A in the (possibly infinite) set CNF. In Section 7.4, we consider how to implement efficient checks that do not use universal quantification.

Figure 8 shows a visualization of where robust declassification is allowed for a given robust privilege. The security lattice on the left is overlaid with a visualization of where a value with label L_{from} can be declassified to (shaded line) using a robust declassification privilege. (Note that the current label L_{pc} is not included in the diagram for brevity.) I represents the boolean formula for the integrity of the labeled value. L_{low} is one of the lowest points where L_{from} can be declassified to while still being a robust declassification, i.e., $L_{low} \sqsubseteq L_{to}$. That is, the integrity of the label of the value for declassification (together with the integrity of the current label of the process) is used as a lower bound for declassification. Intuitively, those who influence a declassification should not learn from it. In the right hand side of Figure 8, the shaded line indicates to where a robust privilege may declassify the labeled value $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A}\rangle$. The declassification is robust if A is not able to learn from the declassification. As a result, the value could not be declassified to $\langle \mathsf{A} \vee \mathsf{B}, \mathsf{A}\rangle$ as A would learn from a declassification that it influenced. In contrast, it is robust to declassify it to $\langle \mathsf{B}, \mathsf{A}\rangle$.

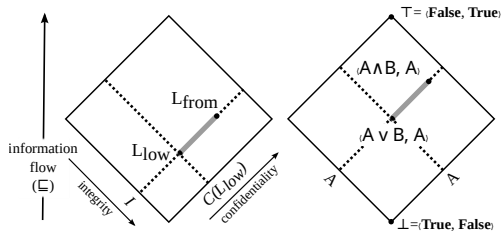

Fig. 8: Robust Declassification

7.4 Enforcement for robust privileges

In this section we describe enforcement mechanisms for restricted privileges that satisfy their semantic characterizations described in Section 7.3. We have implemented these mechanisms in LIO and use them in our case study (see Section 7.6).

When a bounded privilege (Definition 2) is used at run time, it is simple to check that the downgrading operation satisfies the appropriate bounds,

since the labels relevant to the downgrading (L_{from}, L_{to}, and L_{pc}) are all available at run time, and the label ordering relation can be easily checked dynamically.

Robust privileges (Definition 5) impose restrictions on downgrading operations which quantify over formulas A. However, attempting to explicitly check each possible formula A at run time is not feasible. We can however, derive simple and efficient run-time checks that are sound and complete with respect to their semantic characterizations. These checks are inspired by Chong and Meyers [Chong and Myers, 2006], who provide run-time checks for robustness that are sound but not complete.

The following theorem shows that the semantic characterization of robust declassification (Definition 3) is equivalent to two confidentiality-policy comparisons involving only L_{from}, L_{to}, and L_{pc}.

Theorem 1 (Robust declassification check). *A declassification using privilege $p^{\Downarrow}$ from a source protected by L_{from} to a sink protected by L_{to} in a computational context with current label L_{pc} is robust if and only if* $\mathrm{C}(L_{from}) \sqsubseteq^{\mathrm{C}}_{p} \mathrm{C}(L_{to})$, $\mathrm{C}(L_{from}) \sqsubseteq^{\mathrm{C}} \mathrm{C}(L_{to}) \sqcup^{\mathrm{C}} \mathrm{I}(L_{pc})$, *and* $\mathrm{C}(L_{from}) \sqsubseteq^{\mathrm{C}} \mathrm{C}(L_{to}) \sqcup^{\mathrm{C}} \mathrm{I}(L_{from})$.

The run-time check ensures that if there is any formula A that benefits from the declassification ($\mathrm{C}(L_{from}) \not\sqsubseteq^{\mathrm{C}} A$ and $\mathrm{C}(L_{to}) \sqsubseteq^{\mathrm{C}} A$) then $A \not\sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{pc})$ (or, equivalently, $\mathrm{I}(L_{pc}) \not\sqsubseteq^{\mathrm{C}} A$), and similarly that $A \not\sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{from})$. Thus, the run-time check converts a comparison of integrity policies to a comparison of integrity policies that does not involve A.

The next theorem describes a simple run-time check for robust endorsement.

Theorem 2 (Robust endorsement check). *An endorsement using privilege $p^{\Downarrow}$ from a source protected by L_{from} to a sink protected by L_{to} in a computational context with current label L_{pc} is robust (Definition 4) if and only if* $\mathrm{I}(L_{from}) \sqsubseteq^{\mathrm{I}}_{p} \mathrm{I}(L_{to})$, *and* $\mathrm{I}(L_{pc}) \sqcap^{\mathrm{I}} \mathrm{I}(L_{from}) \sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{to})$.

The run-time check that all formulas A that may be responsible for either the current label ($A \sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{pc})$) or the data itself ($A \sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{from})$) should also be responsible for the data after endorsement ($A \sqsubseteq^{\mathrm{I}} \mathrm{I}(L_{to})$). Proofs of Theorems 1 and 2 are in Appendix A.

Alternative formulation In DC-labels, privileges can be arbitrary formulas, which can be stronger or weaker than privileges for individual principals. For example, a privilege for $\mathsf{A} \wedge \mathsf{B}$ can downgrade more information than a privilege for A or B alone, whereas a privilege for $\mathsf{A} \vee \mathsf{B}$ can downgrade less information than a privilege for A or B alone. Leveraging this feature, we show how robust downgrading can be seen (and enforced) as normal downgrading operations that use a weakened privilege. That is, the privilege

used in a downgrading operation is weakened so as to permit all and only robust downgrading operations.

The next corollaries follow from Theorems 1 and 2 and the definition for the *can-flow-to-with-privilege-p* relation.

Corollary 1. *A declassification using raw privilege* $p^{\Large\♛}$ *from a source protected by* L_{from} *to a sink protected by* L_{to} *in a computational context with current label* L_{pc} *is robust (Definition 3) if and only if*

$$\mathrm{C}(L_{from}) \sqsubseteq^{\mathrm{C}}_{p \,\vee\, \mathrm{I}(L_{from}) \,\vee\, \mathrm{I}(L_{pc})} \mathrm{C}(L_{to}).$$

This indicates that robust declassification can be achieved by simply weakening privilege $p^{\Large\♛}$ with the integrity labels of the current label and the data to be released, i.e., $p \vee \mathrm{I}(L_{from}) \vee \mathrm{I}(L_{pc})$. Robust endorsement has a similar corollary.

Corollary 2. *An endorsement using raw privilege* $p^{\Large\♛}$ *from a source protected by* L_{from} *to a sink protected by* L_{to} *in a computational context with current label* L_{pc} *is robust (Definition 3) if and only if* $\mathrm{I}(L_{from}) \sqsubseteq^{\mathrm{I}}_{p \,\vee\, \mathrm{I}(L_{pc})} \mathrm{I}(L_{to})$.

The proof of Corollary 1 is in Appendix A; the proof of Corollary 2 is similar.

The current implementation of DC-labels [Stefan et al., 2011a] provides the ability to infer appropriate L_{to} labels of downgrading operations given a privilege p. By expressing the runtime checks for robust downgrading operations as a standard downgrading operation with a weakened privilege, we can take advantage of this feature and automatically infer a suitable L_{to} label if one exists. This reduces the burden on the programmer.

7.5 Interaction among restricted privileges

We can extend restricted privileges to allow them to be composed, i.e., by allowing bounded privileges and robust privileges to wrap around other restricted privileges, as well as raw privileges. The guarantee provided by the composition of restricted privileges is the intersection of their individual guarantees. For example, a bounded privilege composed with another bounded privilege will require that downgrading operations satisfy the bounds of both privileges. A bounded privilege composed with a robust privilege (and vice-versa) requires the downgrading both to be robust and satisfy the downgrading bounds. Robust privileges are idempotent: a robust privilege composed with a robust privilege will simply require all downgrade operations to be robust.

Privileges might also interact because a system has multiple privileges available. Unlike composed privileges (which further restrict possible information flows), multiple privileges enable additional information flows.

In the remainder of the section, we discuss the guarantees that result from the use of multiple restricted privileges. In the accompanying figures, bounded privileges are depicted as a shaded rectangle corresponding to their bounds. Robust declassification privileges are depicted as a pair of dashed lines: one line represents the integrity of the source and the other line represents the lower bound to which data may be declassified. Labels are depicted as points along with their names.

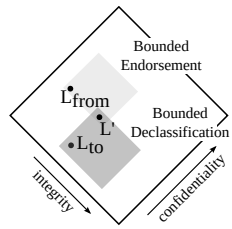

Fig. 9: Multiple bounds.

Bounded declassification and bounded endorsement Figure 9 depicts two bounded privileges, one for declassification and one for endorsement, as well as a label, L_{from} that is outside the bounds of the declassification privilege. Because the bounds of the privileges overlap, data can transitively flow from L_{from} to L_{to}. The endorsement privilege enables data from L_{from} to be endorsed to L'. The bounded declassification privilege can then declassify data from L' to L_{to}.

Bounded declassification and robust declassification Figure 10 depicts two declassification privileges, one robust and one bounded, and a label that is outside the bounds of the bounded declassification privilege. Neither privilege alone permits a flow from L_{from} to L_{to}. However, when used together, the robust declassification privilege permits declassification of data from L_{from} to L' and the bounded declassification permits a flow from L' to L_{to}, completing a flow from L_{from} to L_{to}.

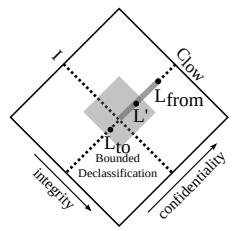

Fig. 10: Bounded and robust declassification.

Endorsement and robust declassification
In a system with unrestricted endorsement, robust declassification provides almost no protection against attackers influencing what they learn. Intuitively, the endorsement of data by p can make the data trustworthy enough to make a subsequent declassification robust. Consider a declassification of a value from label $L_{from} = \langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A} \rangle$ to $L = \langle \mathsf{A}, \mathsf{A} \rangle$ using the robust privilege $rbst^{\mathsf{d}}\{\mathsf{B}^{\Large\♕}\}$. This declassification is not robust: principal A, who benefits from this declassification, may be held responsible for the value, i.e., A may have decided what gets declassified. However, an unrestricted endorsement privilege $\mathsf{B}^{\Large\♕}$ could be used to endorse the value—effectively endorsing any possible influence by A. In other words, $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A} \rangle$ can be

endorsed to $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{B} \rangle$, and a subsequent declassification from $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{B} \rangle$ to $\langle \mathsf{A}, \mathsf{B} \rangle$ is robust.

Bounded endorsement effectively limits the aforementioned deletrious effects of unrestricted endorsement to the bounded area of the lattice, Figure 11 depicts this situation. Besides mitigating the effects of unrestricted endorsement, bounded endorsement is useful to relax robust declassification so that it succeeds for principals collaborating in achieving a common goal—see, for example, Section 7.6.

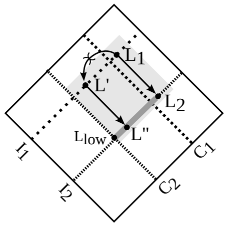

Fig. 11: Bounded endorsement and robust declassification.

Bounded and robust declassification Figure 10 shows the guarantees when a robust declassification-only privilege (i.e., $rbst^{\mathsf{d}}\{p^{♛}\}$) and a bounded declassification-only privilege (i.e., ${}^{\mathsf{d}}[p^{♛}]^{L_{high}}_{L_{low}}$) for the same principal are both available in the system. Intuitively, p's information can be declassified from L_{from} to L' using the robust privilege. The information can then be declassified again to L_{to} using the bounded privilege, even though L_{from} is below the threshold imposed by robust declassification (i.e., the lowest possible label that robust declassification could declassify label L_{from}). Thus, the presence of a bounded declassification-only privilege can bypass the guarantees provided by robust declassification.

Several bounded privileges Multiple robust privileges for the same principal do not add any additional complexity, as all robust privileges are equivalent (up to their modes). Bounded privileges, however, may differ on the bounds they impose. The presence of multiple bounded privileges in a system for principal p collapses the label lattice for principal p in complex ways. For instance, the left diagram of Figure 9 illustrates an example where there is a bounded endorsement-only privilege and a bounded declassification-only privilege with different bounds. It may be possible for a value labeled L_{from} to be relabeled to L_{to} via an endorsement to L' followed by a declassification. Thus, labels between L_{from} and L' and between L' and L_{to} are effectively collapsed, since the bounded privileges allow a value with any of these labels to be relabeled to any other of these labels. More generally, as more overlapping bounded privileges exist for a given principal, data can be downgraded in more possible ways.

7.6 Case studies

7.6.1 Calendar Case Study

We have extended LIO [Stefan et al., 2011b] with support for bounded privileges and robust privileges, and used them to develop a Calendar ap-

plication to explore and illustrate the utility of restricted privileges. The application allows users to view their appointments, and schedule appointments with each other. DC-label principals are the calendar users. A user's appointments are confidential to that user.

We consider a setting where principals belong to groups and a principal is willing to disclose her availability to all and only members of her groups. For example, if Bob wants to schedule an appointment with Alice at time t, the application will check Alice's calendar and inform Bob whether Alice is available at that time. This operation, which declassifies Alice's availability at time t to Bob, should succeed only if Alice and Bob are in the same group.

Each user A has a robust declassification privilege $rbst^{\mathsf{d}}\{\mathsf{A}^{\Large\♕}\}$, and, for each group G that A belongs to, a bounded endorsement privilege ${}^{\mathsf{e}}[\mathsf{A}^{\Large\♕}]^{\langle \top^{\mathsf{c}}, G\rangle}_{\langle \bot^{\mathsf{c}}, \bot^{\mathsf{I}}\rangle}$, where G is the disjunction of all users in the group. These are the only privileges available in the system for user A, and thus all endorsements must be bounded appropriately, and all declassifications must be robust.

Joint scheduling between A and B works as follows:

1. User B sends a scheduling request for time t labeled $\langle \mathsf{B}, \mathsf{B}\rangle$ to user A.
2. User A computes her availability for time t. Because the context that computes the availability reads data labeled $\langle \mathsf{A}, \mathsf{A}\rangle$ and $\langle \mathsf{B}, \mathsf{A}\rangle$, the label of the availability result is $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A} \vee \mathsf{B}\rangle$.
3. If A and B are both in some group G, then A uses her bounded privilege to endorse the availability result to $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A}\rangle$, since she is prepared to take sole responsibility for the availability result. Since both A and B are in the same group, the endorsement satisfies the bounds (i.e., $\mathsf{A} \vee \mathsf{B} \sqsubseteq^{\mathsf{I}} G$). If there is no group for which both A and B are members, then A has no bounded endorsement privilege for which the bounds will be satisfied.
4. User A uses her robust privilege to declassify the availability result to $\langle \mathsf{B}, \mathsf{A}\rangle$. The declassification is robust.
5. User A sends the declassified value to B.

Because all downgrading in the system relevant to user A must use A's restricted privileges, we obtain strong system-wide guarantees, even if A's restricted privileges manage to escape from the scheduling component, and even if if B sends malicious scheduling requests. Section 7.5 (Figure 11) discusses in more detail the system-wide guarantees that hold when both a bounded endorsement privilege and a robust declassification privilege are available.

7.6.2 Restricted Privileges in Existing Applications

Using our restricted privileges, we found a security vulnerability in an application written using Haskell Automatic Information Labeling System (Hails) [Giffin et al., 2012]. Hails is a web framework built on LIO that

extends the traditional Model-View-Controller paradigm to Model-Policy-View-Controller. The policy module specifies all models and describes the labels for data fetched from the database. When data is stored in the database, Hails checks labels against the policy module to ensure appropriate data integrity. The policy module has access to a privilege that can declassify all models. As a design pattern, policy modules export functions that perform declassification for untrusted applications using the privilege; untrusted applications never have direct access to the privilege.

Rock-Paper-Scissors[1] is a Hails application that contains a security vulnerability due to misuse of the policy privilege, despite being written by security experts who developed Hails.

The policy module includes a function to get the outcome of a match given a particular move by a player. This function can be exploited to reveal the opponent's move before the player has actually committed to a move by submitting it to the database. As a result, a player can always win a match by exploiting this function to determine which move will win, and then committing to that winning move. When we replaced the policy module's raw privilege with a robust privilege, the robust declassification check signalled a potential security vulnerability. To fix the vulnerability, we added code that checks whether a player had committed to a move (i.e., the move is in the database), and, if so, endorses the submitted move. This endorsement allows the robust declassification check to succeed. Endorsing only when the player has committed to his move fixes the security vulnerability.

7.7 Related Work

Declassification can be characterized into different dimensions: *who*, *what*, *where*, and *when* [Sabelfeld and Sands, 2005]. Our work can be considering as restricting *where* in the security lattice downgrading may occur (bounded downgrading) and *who* may influence downgrading (robustness). Almeida Matos and Boudol [Almeida Matos and Boudol, 2005] introduce a construct **flow** $p \prec q$ **in** c to indicate *where* additional information flows are allowed within a lexical scope. Intransitive noninterference [Roscoe and Goldsmith, 1999, Mantel and Sands, 2004, van der Meyden, 2007] posits a non-transitive information flow ordering which describes *what* downgrading operations are permitted. Mantel and Sands [Mantel and Sands, 2004] combine intransitive noninterference with language techniques that use declassification annotations to explicitly identify non-transitive information flows. In our bounded declassification mechanism, violating the normal ordering of security levels is tied to a runtime value, and not lexically scoped or marked by annotations.

In Jif [Myers et al., 2001–], declassifications may explicitly state where in the security lattice the declassification occurs. By contrast, our bounded

[1] `https://github.com/scslab/hails/tree/master/examples/hails-rock`

mechanisms declare this restriction on the run-time value that authorizes downgrading. Jif uses a form of access control to restrict which code may downgrade information, coined *selective declassification* by Pottier and Conchon [Pottier and Conchon, 2000]. Specifically, a downgrading operation that may compromise the security of principal p may only occur in code that has been (statically or dynamically) authorized by p. Similarly, the authority to declassify or endorse information in Asbestos [Efstathopoulos et al., 2005], HiStar [Zeldovich et al., 2006], Flume [Krohn et al., 2007], and COWL [Stefan et al., 2014] must come from the creator of the exercised privileges. By contrast, LIO associates the authority to declassify or endorse a principal's information with a run-time value. This capability-like approach to authorizing downgrading enables our local declarative approach to restrict downgrading. Birgisson et al. [Birgisson et al., 2011] use capabilities to restrict the ability to read and write memory locations, but do not consider the use of capabilities to restrict downgrading.

Zdancewic and Myers [Zdancewic and Myers, 2001] introduce the semantic security condition of *robust declassification*, and Myers et al. [Myers et al., 2006] enforce robust declassification with a security type system [Volpano et al., 1996, Sabelfeld and Myers, 2003], and introduce *qualified robustness*, which extends the concept to reason about endorsement. Askarov and Myers [Askarov and Myers, 2010] subsequently present a semantic framework for downgrading, and present a crisper version of qualified robustness. Chong and Myers [Chong and Myers, 2006] extend the notion of robust declassification to the Decentralized Label Model [Myers and Liskov, 1998, 1997]. The run-time checks used in this work to enforce robustness are analogous to the run-time checks Chong and Myers introduce for the DLM. In other work, Chong and Myers [Chong and Myers, 2005] note that the semantic security condition for robust declassification applies to information flow of confidential information generally, including, for example, information erasure, and is more general than just declassification. If the only privilege for p available in the system is a robust privilege with mode mode d then the system will be robust for p. If the privilege for that mode is de (i.e., robust declassification operations and robust endorsement operations are possible), then the end-to-end security guarantee is *qualified robustness* [Myers et al., 2006, Askarov and Myers, 2010]. A system satisfies qualified robustness if the only way an attacker can influence what information is released to it is via robust endorsement operations.

Foley et al. incorporate bounds constraints on a system with relabeling operations on objects [Foley et al., 1996]. Our model performs relabeling based on the use of capability-like tokens rather than with respect to a particular subject. Bound restrictions can be placed per privilege rather than on all relabeling operations, so the guarantees of this work are more dependent on what sorts of privileges are available for use, but do not require changes to the trusted computing base.

The system HiStar [Zeldovich et al., 2006] provides the notion of gates: entities designed to encapsulate privileges so that processes can safely switch their current label by exercising them through the gate. Gates have a clearance component which imposes an upper bound on the label that results from using it. Gates can be leveraged to restrict the use of privileges similar to upper bounds in bounded privileges. Similar to our approach, Flume[Krohn et al., 2007] distinguishes privileges used for declassification (symbol $-$) and endorsement (symbol $+$).

7.8 Conclusion

Restricted privileges are a new mechanism to control declassification and endorsement in DC-labels that is simple and intuitive yet expresses a rich set of desirable policies. Bounded privileges impose upper and lower bounds on data that is declassified or endorsed. Robust privileges help prevent the accidental or malicious exercise of privileges to downgrade more information than intended, and can provide the end-to-end security guarantees of robustness and qualified robustness. We provide sound and complete efficient security checks for downgrading using restricted privileges. We note that robust downgrading operations can be viewed as privileged downgrading with a weakened privilege. We explore the guarantees provided by combining the use of bounded and robust privileges as well as their composition in a case study. This work establishes a basis for better design of IFC systems that use privileges for downgrading information.

BIBLIOGRAPHY

Ana Almeida Matos and Gerard Boudol. On declassification and the non-disclosure policy. In *Proc. 18th IEEE Computer Security Foundations Workshop*, pages 226–240, 2005.

Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In *Proc. 19th European Symposium on Programming*, 2010.

K. J. Biba. Integrity considerations for secure computer systems. ESD-TR-76-372, 1977.

Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Capabilities for information flow. In *Proc. 6th Workshop on Programming Languages and Analysis for Security*, 2011.

Stephen Chong and Andrew C. Myers. Language-based information erasure. In *Proc. 18th IEEE Computer Security Foundations Workshop*, pages 241–254, June 2005.

Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proc. 19th IEEE Workshop on Computer Security Foundations*, pages 242–256, 2006.

Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proc. 20th ACM Symposium on Operating Systems Principles*, 2005.

Simon Foley, Li Gong, and Xiaolei Qian. A security model of dynamic labeling providing a tiered approach to verification. In *Proc. 1996 IEEE Symposium on Security and Privacy*, pages 142–158, 1996.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. Symposium on Operating Systems Design and Implementation*, 2012.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control

for standard OS abstractions. In *Proc. 21st Symp. on Operating Systems Principles*, October 2007.

Heiko Mantel and David Sands. Controlled Declassification based on Intransitive Noninterference. In *Proc. 2nd Asian Symposium on Programming Languages and Systems*, volume 3303, pages 129–145, November 2004.

A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release. `http://www.cs.cornell.edu/jif`, 2001–.

Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 16th ACM Symposium on Operating System Principles*, pages 129–142, New York, NY, USA, 1997.

Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 186–197, May 1998.

Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, April 2006.

François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming*, pages 46–57, New York, NY, USA, 2000.

A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. 12th IEEE Computer Security Foundations Workshop*, 1999.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1): 5–19, January 2003.

Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proc. 18th IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *16th Nordic Conference on Security IT Systems*, volume 7161, pages 223–239, October 2011a.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible Dynamic Information Flow Control in Haskell. In *Proc. 4th ACM symposium on Haskell*, pages 95–106, New York, NY, USA, 2011b.

Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *Proc. 11th Symposium on Operating Systems Design and Implementation*, October 2014.

Ron van der Meyden. What, indeed, is intransitive noninterference? In *Proc. 12th European Symposium On Research In Computer Security*, volume 4734, pages 235–250, September 2007.

Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proc. 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, 2006.

A Proofs

The integrity and confidentiality lattices are dual lattices, as described below. This fact is used in subsequent proofs.

$$\begin{aligned} I_1 \sqsubseteq^{\mathrm{I}} I_2 = I_1 \Rightarrow I_2 &= I_2 \sqsubseteq^{\mathrm{C}} I_1 \\ I_1 \sqcup^{\mathrm{I}} I_2 \;=\; I_1 \vee I_2 \;&= I_1 \sqcap^{\mathrm{C}} I_2 \\ I_1 \sqcap^{\mathrm{I}} I_2 \;=\; I_1 \vee I_2 \;&= I_1 \sqcup^{\mathrm{C}} I_2 \end{aligned}$$

Recall the robust declassification check from Theorem 1:

$$\mathrm{C}(L_{from}) \sqsubseteq^{\mathrm{C}} \mathrm{C}(L_{to}) \sqcup^{\mathrm{C}} \mathrm{I}(L_{pc}) \tag{1}$$

and

$$\mathrm{C}(L_{from}) \sqsubseteq^{\mathrm{C}} \mathrm{C}(L_{to}) \sqcup^{\mathrm{C}} \mathrm{I}(L_{from}) \tag{2}$$

Proof (Proof of Theorem 1). As a first step, we adapt the goal in Definition 3 to $\sqsubseteq^{\mathrm{C}}$ instead of $\sqsubseteq^{\mathrm{I}}$ by way of the lattice duality.
Soundness: We prove that (1) and (2) implies Definition 3. For that, we assume (1), (2), and

$$\forall A \in \mathrm{CNF}\,, \mathrm{C}(L_{to}) \sqsubseteq^{\mathrm{C}} A \text{ and } \mathrm{C}(L_{from}) \not\sqsubseteq^{\mathrm{C}} A \tag{3}$$

having to prove that

$$\mathrm{I}(L_{pc}) \not\sqsubseteq^{\mathrm{C}} A \tag{4}$$

and

$$\mathrm{I}(L_{from}) \not\sqsubseteq^{\mathrm{C}} A \tag{5}$$

We only prove (4) by contradiction (the proof for (5) follows analogously). We have that,

$\mathrm{I}(L_{pc}) \sqsubseteq^{\mathrm{C}} A$	contradiction supposition
$\mathrm{I}(L_{pc}) \sqsubseteq^{\mathrm{C}} A$ and $\mathrm{C}(L_{to}) \sqsubseteq^{\mathrm{C}} A$	obtained from (3) $\Longrightarrow$
$\mathrm{C}(L_{to}) \sqcup^{\mathrm{C}} \mathrm{I}(L_{pc}) \sqsubseteq^{\mathrm{C}} A$	least upper bound
$\mathrm{C}(L_{from}) \sqsubseteq^{\mathrm{C}} A$	from (1) and transitivity

Completeness: We show that Definition 3 implies (1) and (2). Again, we prove only the left side of the conjunct, the right side follows analogously. Consider the contrapositive of the robust declassification condition (see Definition 3) with $A = \mathrm{C}(L_{to}) \sqcup \mathrm{I}(L_{pc})$, where we disregard the $\mathrm{I}(L_{from})$ half of the antecedent:

$$\begin{aligned} &\text{if } \mathrm{I}(L_{pc}) \sqsubseteq^{\mathrm{C}} \mathrm{I}(L_{pc}) \sqcup^{\mathrm{C}} \mathrm{C}(L_{to}) \\ &\text{then } \mathrm{C}(L_{to}) \not\sqsubseteq^{\mathrm{C}} \mathrm{I}(L_{pc}) \sqcup^{\mathrm{C}} \mathrm{C}(L_{to}) \\ &\quad \text{or } \mathrm{C}(L_{from}) \sqsubseteq^{\mathrm{C}} \mathrm{I}(L_{pc}) \sqcup^{\mathrm{C}} \mathrm{C}(L_{to}) \end{aligned}$$

The left side of the disjunction is false by the definition of least upper bound. We conclude that $\mathrm{C}(L_{from}) \sqsubseteq^{\mathrm{C}} \mathrm{I}(L_{pc}) \sqcup^{\mathrm{C}} \mathrm{C}(L_{to})$ which proves the left side of the conjunct of the robust declassification check.

Proof (Proof of Theorem 2). Soundness: We show that the robust endorsement condition (see Definition 4) implies the robust endorsement check. Consider the contrapositive of the robust endorsement condition:

$$\begin{aligned}\forall A \in \text{CNF}.\ &\text{if } q \sqsubseteq^{\text{I}} \text{I}(L_{pc}) \\ &\text{then } q \sqsubseteq^{\text{I}} \text{I}(L_{to}) \text{ or } q \not\sqsubseteq^{\text{I}} \text{I}(L_{from})\end{aligned}$$

Setting $A = \text{I}(L_{pc}) \sqcap^{\text{I}} \text{I}(L_{from})$ allows us to conclude that the robustness condition,

$$\text{I}(L_{pc}) \sqcap^{\text{I}} \text{I}(L_{from}) \sqsubseteq^{\text{I}} \text{I}(L_{to})$$

is true because the right side of the disjunction must be false by the greatest upper bound property.
Completeness: Now we show that the robust endorsement check implies the robust endorsement condition. We take as suppositions the robust endorsement check and the antecedent of the robust endorsement condition:

$$\begin{aligned}&\text{I}(L_{pc}) \sqcap^{\text{I}} \text{I}(L_{from}) \sqsubseteq^{\text{I}} \text{I}(L_{to}) \\ &\forall A \in \text{CNF}\,, A \not\sqsubseteq^{\text{I}} \text{I}(L_{to}) \text{ and } A \sqsubseteq^{\text{I}} \text{I}(L_{from})\end{aligned}$$

The proof is by contradiction.

$$\begin{aligned}&A \sqsubseteq^{\text{I}} \text{I}(L_{pc}) && \text{contradiction supposition} \\ &A \sqsubseteq^{\text{I}} \text{I}(L_{pc}) \text{ and } A \sqsubseteq^{\text{I}} \text{I}(L_{from}) \implies && \\ &\quad A \sqsubseteq^{\text{I}} \text{I}(L_{pc}) \sqcap^{\text{I}} \text{I}(L_{from}) && \text{greatest lower bound} \\ &A \sqsubseteq^{\text{I}} \text{I}(L_{to}) && \text{transitivity}\end{aligned}$$

Which contradicts the antecedent of the robustness condition.

Proof (Proof of Corollary 1). The robust declassification checks

$$\begin{aligned}&\text{C}(L_{from}) \sqsubseteq^{\text{C}}_{p} \text{C}(L_{to}) \\ \text{and}\quad &\text{C}(L_{from}) \sqsubseteq^{\text{C}} \text{C}(L_{to}) \sqcup^{\text{C}} \text{I}(L_{pc}) \\ \text{and}\quad &\text{C}(L_{from}) \sqsubseteq^{\text{C}} \text{C}(L_{to}) \sqcup^{\text{C}} \text{I}(L_{from})\end{aligned}$$

are equivalent to

$$\text{C}(L_{from}) \sqsubseteq^{\text{C}}_{p \vee \text{I}(L_{pc}) \vee \text{I}(L_{from})} \text{C}(L_{to})$$

which can be derived by unfolding the definitions of $\sqsubseteq^{\text{C}}$ and $\sqsubseteq^{\text{C}}_{p}$ and applying the logical tautology

$$(\phi \wedge (\psi_1 \vee \psi_2) \implies \chi) \iff (\phi \wedge \psi_1 \implies \chi) \wedge (\phi \wedge \psi_2 \implies \chi).$$

Part III

Alternative enforcement mechanisms

CHAPTER
EIGHT

ON DYNAMIC FLOW-SENSITIVE FLOATING-LABEL SYSTEMS

Pablo Buiras, Deian Stefan, Alejandro Russo

Abstract. Flow-sensitive analysis for information-flow control (IFC) allows data structures to have mutable security labels, i.e., labels that can change over the course of the computation. This feature is often used to boost the permissiveness of the IFC monitor, by rejecting fewer runs of programs, and to reduce the burden of explicit label annotations. However, adding flow-sensitive constructs (e.g., references or files) to a dynamic IFC system is subtle and may also introduce high-bandwidth covert channels. In this work, we extend LIO—a language-based floating-label system—with flow-sensitive references. The key insight to safely manipulating the label of a reference is to not only consider the label on the data stored in the reference, i.e., the reference label, but also the *label on the reference label* itself. Taking this into consideration, we provide primitives *upgrade* and *downgrade* that can be used to change the label of a reference in a safe manner. We additionally provide a mechanism for automatic upgrades to eliminate the burden of determining when a reference should be upgraded. This approach naturally extends to a concurrent setting, which has not been previously considered by dynamic flow-sensitive systems. For both our sequential and concurrent calculi we prove non-interference by embedding the flow-sensitive system into the original, flow-insensitive LIO calculus—a surprising result on its own.

8.1 Introduction

Modern software systems are composed of many complex components that handle sensitive data. In many cases (e.g., mobile and web applications) these disparate components are provided by different authors, of varying trustworthiness. Unfortunately, because today's software development tools do not provide a means for protecting sensitive data from untrusted code, data theft and corruption is prevalent.

Information-flow control (IFC) is a promising approach to security that provides data confidentiality and integrity in the presence of untrusted code. At a high level, IFC tracks and controls the flow of information through a system according to a security policy, usually *non-interference* [Goguen and Meseguer, 1982]. Non-interference states that public events should not depend on sensitive data and dually, trusted data should not be affected by untrusted events. Hence, with IFC, the program is guaranteed to preserve data confidentiality and integrity, even when composed of untrusted components. Indeed, this appealing guarantee has recently led to significant research and development efforts that use IFC to secure web applications (e.g., [De Groef et al., 2012, Giffin et al., 2012, Yang et al., 2013, Hedin et al., 2014]) and mobile platforms (e.g., [Enck et al., 2010, Jia et al., 2013]).

To ensure data confidentiality and integrity, these dynamic IFC systems associate *security labels* with data and monitor where such data can flow [Myers and Liskov, 1997, Stefan et al., 2012b]. In this paper, we use the labels **H** and **L**, to respectively denote secret and public data, and ensure that information cannot flow from a secret entity into a public one, i.e., the labels are ordered such that $\mathbf{L} \sqsubseteq \mathbf{H}$ and $\mathbf{H} \not\sqsubseteq \mathbf{L}$. In general, the partial order $\sqsubseteq$ (label check) is used to govern the allowed flows. We remark that our results apply to arbitrary lattices that may also express integrity concerns [Myers and Liskov, 1997, Stefan et al., 2012b], we only use the two-point lattice for simplicity of exposition.

One of the facets of IFC analysis lies in how such labels, when associated with objects, are treated [Hunt and Sands, 2006]. Specifically, some IFC systems (e.g., [Hritcu et al., 2013, Cheng et al., 2012, Stefan et al., 2011, 2012a, Zeldovich et al., 2006, Efstathopoulos et al., 2005, Krohn et al., 2007]) treat labels on objects as *immutable* and do not allow for changes over the lifetime of the program, i.e., labels of objects are *flow-insensitive*. In contrast, other systems (e.g., [Zdancewic, 2002, Austin and Flanagan, 2009, 2010]) are *flow-sensitive*, i.e., they allow object labels to change, in certain conditions, according to the sensitivity of the data that is stored in the object. In general, these flow-sensitive systems are more permissive, i.e., they allow programs that flow-insensitive monitors would reject.

Consider, for instance, a web application that writes to a labeled log while servicing user requests. If the label of the log is **L**, a flow-insensitive IFC monitor would disallow writing any sensitive data (e.g., error messages containing user-supplied data) to the log, since this would constitute a leak.

However, in a flow-sensitive system, the label of the log can change (to **H**), as to accommodate the kinds of data being written to the log. For many applications, allowing labels to change in such a way is very desirable—it alleviates the burden of having to a-priori determine the precise labels of objects (e.g., the log).

Unfortunately, naively introducing flow-sensitive objects to a dynamic IFC system can turn label changes into a covert channel [Russo and Sabelfeld, 2010]. Consider the code fragment of Figure 1 where references l and h are respectively labeled **L** and **H**. By naively allowing arbitrary label changes—even if the new label is more restricting—we can leak the contents of h into l. In particular, suppose that the temporary variable tmp is initially labeled **L**. If the value stored in h is $True$, then in the first conditional we assign $True$ into tmp and raise its label to **H**, reflecting the fact that the branch condition depends on sensitive data. Since the tmp is $True$, the branch condition for the second conditional is $False$ and thus the value and label of l are left intact, i.e., $True$ at **L**. However, if h is $False$, then the value and label of tmp do not change—the first assignment is not executed. Instead, the second assignment, which sets l to $False$, is performed; since the label of the branch condition is **L**, the label of l remains **L**. Note that in both cases the label of l stays **L**, but the value of l becomes the same as the secret h. (Hence *label change* is considered a covert channel.) In systems such as LIO and Breeze, which allow labels to be inspected, this attack can be further simplified by simply checking the label of tmp after the first assignment—if the secret is true then the label will be **H**, otherwise it will be **L**.

$l \quad := True$
$tmp := False$
if h **then** $tmp := True$
if $\neg\ tmp$ **then** $l := False$

Fig. 1: Flow-sensitive attack

This attack is not new, and, to ensure that the covert channel is not introduced when adding flow-sensitive references in such a way, several solutions have already been proposed. These solutions fall into roughly three categories. First, the IFC monitor can incorporate static information to ensure that such leaks are disallowed [Russo and Sabelfeld, 2010]. Second, the IFC monitor can forbid certain label changes, depending on the context (e.g., the program counter (pc) label [Sabelfeld and Myers, 2003]). For instance, the *no-sensitive upgrades* policy disallows raising the label of a public reference in a sensitive context (e.g., when a branch condition is **H**) [Zdancewic, 2002, Austin and Flanagan, 2009]. And, third, the monitor can disallow branches that depend on certain variables, for which the label was mutated, as done by the *permissive upgrades* policy [Austin and Flanagan, 2010].

In this paper, we take a fresh perspective on flow-sensitivity in the context of coarse-grained floating-label systems, in particular, the LIO IFC

system [Stefan et al., 2011, 2012a]. LIO brings ideas from IFC Operating Systems—notably, HiStar [Zeldovich et al., 2006]—into a language-based setting. In particular, LIO takes an OS-like coarse-grained approach by associating a single "current" floating-label with a computation (and everything in scope), instead of heterogeneously labeling every variable, as typically done by language-based systems (e.g., [Myers et al., 2001, Simonet, 2003]). This floating-label is raised (e.g., from **L** to **H**) to accommodate reading sensitive data and thus serves as a form of "taint" reflecting the sensitive of data in context, i.e., LIO is flow-sensitive in the current label. (This can be seen as raising the *pc* in more traditional language-based systems.) In turn, the LIO monitor uses the "current" label to restrict where the computation can write (e.g., once the current label is raised to **H**, it can no longer write to references labeled **L**). However, like other IFC systems, LIO is *flow-insensitive* in object labels.

This work extends the LIO IFC system, both the sequential and concurrent versions, to incorporate flow-sensitive references. A key insight of this work is to consider labels of references as being composed of two elements: the reference label describing the confidentiality (integrity) of the stored value, and another label, called *the label on the label*, which describes the confidentiality (integrity) of the reference label itself. Our monitor, then, only forbids changing a label of a reference if *the label on the label is below the current floating-label*. Inspired by [Hedin et al., 2014], we add a primitive for safely and explicitly *upgrading* labels. This boosts the permissiveness of LIO, and, for instance, allows programs, such as the logging web application described above, which would otherwise be rejected by the IFC monitor.

To reduce the burden of introducing upgrade annotations, our calculus provides a means for automatically upgrading references for which the computation is about to "lose" write access, i.e., before tainting the computation by raising the current label, we first upgrade all the references whose labels are below the (new) current label. While secure, this feature facilitates a form of *label creep*, wherein all flow-sensitive references might end up with labels that are "too high." To further address this, we propose a block-structured primitive which only upgrades the labels of declared flow-sensitive references, while disallowing writes to undeclared ones.

By taking a fresh perspective on flow-sensitivity, we also show that our flow-sensitive extension can be entirely encoded using existing flow-insensitive constructs—the key insight is to explicitly model flow-sensitive values as *nested flow-insensitive labeled references*. In the context of LIO, this encoding has the added benefit of allowing us to prove non-interference by simply invoking previous results. Equally important, the sequential semantics for LIO with flow-sensitive references directly extend to the concurrent setting.

The contributions of this paper are as follows:

- We extend LIO to incorporate flow-sensitive objects, with a focus on references. Specifically, we introduce two explicit primitives to safely raise (**upgrade**) or lower (**downgrade**) the security label of references. This extension not only increases LIO's permissiveness, but also provides a means for safely combining flow-insensitive and flow-sensitive references.
- We present a uniform treatment for flow-insensitive and flow-sensitive references in both sequential and concurrent settings. To the best of our knowledge, we are the first to analyze the challenges of purely dynamic monitors with flow-sensitive references in the presence of concurrency.
- A non-interference proof for the different calculi that leverages the encoding of flow-sensitive references using flow-insensitive constructs.

The novel aspect of this article, with respect to its conference version [Buiras et al., 2014], is the extension of our formal results to consider a **downgrade** primitive that further boosts permissiveness. Additionally, we compare our approach with *no-sensitive-upgrade* [Zdancewic, 2002] and *permissive-upgrade* [Austin and Flanagan, 2009]—two known policies for label changes.

We remark that while our development focuses on LIO, we believe that our results generalize to other sequential and concurrent floating-label systems (e.g., [Hritcu et al., 2013, Efstathopoulos et al., 2005, Zeldovich et al., 2006, Krohn et al., 2007]).

The rest of the paper is organized as follows. Section 8.2 provides an introduction to LIO and its formalization. Section 8.3 presents our flow-sensitivity extensions and enforcement mechanism. Section 8.4 extends this approach to the concurrent setting. Section 8.5 presents the embedding of our enforcement using flow-insensitive constructs, from which our formal security guarantees follow. We compare our approach with other policies for label change in Section 8.6. We discuss related work in Section 8.7 and conclude in Section 8.8.

8.2 Introduction to LIO

LIO is a language-level IFC system, implemented as a library in Haskell. The library provides a new *monad*, *LIO*, atop which programmers implement computations, which may use the LIO API to perform side effects (e.g., mutate a reference or write to a file).

The *LIO* monad implements a purely dynamic execution monitor. Specifically, *LIO* encapsulates the state necessary to enforce IFC for the computation under evaluation. Part of this state is the current (floating) label. Intuitively, the current label serves a role similar to the program counter (*pc*) of more-traditional IFC systems (e.g., [Simonet, 2003]): it is used to restrict the current computation from performing side-effects that

$$
\begin{array}{lll}
\text{Values } v & ::= & \mathit{True} \mid \mathit{False} \mid () \mid \lambda x.t \mid \ell \mid \mathit{LIO}^{\text{TCB}}\ t \\
\text{Terms } t & ::= & v \mid x \mid t\ t \mid \mathbf{fix}\ t \mid \mathbf{if}\ t\ \mathbf{then}\ t\ \mathbf{else}\ t \\
 & & \mid t \otimes t \mid \mathbf{return}\ t \mid t \gg\!= t \mid \mathbf{getLabel} \\
\text{Types } \tau & ::= & \mathit{Bool} \mid () \mid \tau \to \tau \mid \ell \mid \mathit{LIO}\ \tau \\
\text{Ops}_\ell \quad \otimes & ::= & \sqcup \mid \sqcap \mid \sqsubseteq
\end{array}
$$

Fig. 2: Syntactic categories for base $\lambda_\ell^{\text{LIO}}$.

may compromise the confidentiality or integrity of data (e.g., by restricting where the current computation may write).

To soundly reason about IFC, every piece of data *must* be labeled, including literals, terms, and labels themselves. However and different from most language-based systems (e.g., [Myers and Liskov, 2000, Simonet, 2003, Hritcu et al., 2013]) where every value is explicitly labeled, LIO takes a coarse-grained approach and uses the current label to protect all values in scope. As in IFC operating systems [Efstathopoulos et al., 2005, Zeldovich et al., 2006], in LIO, the current label l_{cur} is the label on all the non-explicitly labeled values in the context of a computation.

To allow for computations on sensitive data, LIO raises the current label to protect newly read data. That is, the current label is raised to "float" above the labels of all the objects read by the current computation. 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 $l'_{\text{cur}} \sqsubseteq l_e$. Hence, for example, a public LIO computation can read secret data by first raising l_{cur} from **L** to **H**. Importantly, however, the new current label prevents the computation from subsequently writing to public entities.

8.2.1 $\lambda_\ell^{\text{LIO}}$: A coarse-grained IFC calculus

We give the precise semantics for LIO by extending the simply-typed, call-by-name λ-calculus; we call this extended IFC calculus $\lambda_\ell^{\text{LIO}}$. The formal syntax of the core $\lambda_\ell^{\text{LIO}}$ calculus, parametric in the label type ℓ, is given in Figure 2. Syntactic categories v, t, and τ represent values, terms, and types, respectively. Values include standard primitives (Booleans, unit, and λ-abstractions) and terminals corresponding to labels (ℓ) and monadic values ($\mathit{LIO}^{\text{TCB}}\ t$).[1] We note that values of the form $\mathit{LIO}^{\text{TCB}}\ t$ denote computations subject to security checks. (In fact, security checks are only applied to such

[1] We restrict our formalization to computations implemented in the *LIO* monad and only consider Haskell features relevant to IFC, similar to the presentation of LIO in [Stefan et al., 2012c].

values.) Terms are composed of standard constructs (values, variables x, function application, the **fix** operator, and conditionals), terminals corresponding to label operations ($t \otimes t$, where $\sqcup$ is the join, $\sqcap$ is the meet, and $\sqsubseteq$ is the partial-order on labels), standard monadic operators (**return** t and $t \gg= t$), and **getLabel**, a term for inspecting the current label, as further explained below. We do not consider terms annotated with $\cdot^{\text{TCB}}$ as part of the surface syntax, i.e., such syntax nodes are not made available to programmers and are solely used internally in our semantic description. Types consist of Booleans, unit, function types, labels, and *LIO* computations; since the $\lambda_\ell^{\text{LIO}}$ type system is standard, we do not discuss it further.

We include monadic terms in our calculus since (in Haskell) monads dictate the evaluation order of a program and encapsulate all side-effects, including I/O [Moggi, 1991, Wadler, 1992]; LIO leverages monads to precisely control what (side-effecting) operations the programmer is allowed to perform at any given time. In particular, an LIO program is simply a computation in the *LIO* monad, composed from simpler monadic terms using *return* and *bind*. Term **return** t produces a computation which simply returns the value denoted by t. Term $\gg=$, called *bind*, is used to sequence LIO computations. Specifically, term $t \gg= (\lambda x.t')$ takes the result produced by term t and applies function $\lambda x.t'$ to it. (This operator allows computation t' to depend on the value produced by t.) We sometimes use Haskell's **do**-notation to write such monadic computations. For example, the term $t \gg= \lambda x.\mathbf{return}\ (x+1)$, which simply adds 1 to the value produced by the term t, can be written using **do**-notation as shown in Figure 3.

do $x \leftarrow t$
  return $(x+1)$

Fig. 3: **do**-notation

A top-level $\lambda_\ell^{\text{LIO}}$ computation is a *configuration* of the form $\langle \Sigma | t \rangle$, where t is the monadic term and Σ is the state associated with the term. As in [Stefan et al., 2011, 2012a], we take an imperative approach to modeling the LIO state as a separate component of the configuration (as opposed to being part of the term). We partially define the state of $\lambda_\ell^{\text{LIO}}$ to (at least) contain the current label l_{cur}, i.e., $\Sigma = (l_{\text{cur}}, \ldots)$; here, ... denotes other parts of the state not relevant at this point. Under this definition, a top-level well-typed $\lambda_\ell^{\text{LIO}}$ term has the form $\Delta, \Gamma \vdash t : \mathit{LIO}\ \tau$, where Δ is the store typing, and Γ is the usual type environment.

We use evaluation contexts in the style of Felleisen and Hieb to specify the reduction rules for $\lambda_\ell^{\text{LIO}}$ [Felleisen and Hieb, 1992]. Figure 4 defines the evaluation contexts for pure terms (E) and monadic terms (**E**) for the base $\lambda_\ell^{\text{LIO}}$. The definitions are standard; we solely highlight that monadic terms are evaluated only at the outermost use of bind (**E** $\gg= t$), as in Haskell. For the base $\lambda_\ell^{\text{LIO}}$, we also give the reduction rule for the monadic term **getLabel**, which simply retrieves the current label. As shown later, it is

$$E ::= E\ t \mid \mathbf{fix}\ E \mid \mathbf{if}\ E\ \mathbf{then}\ t\ \mathbf{else}\ t \mid E \otimes t \mid v \otimes E$$
$$\mathbf{E} ::= [\,] \mid E \mid \mathbf{E} \gg\!= t$$

$$\textsc{getLabel}\ \frac{\Sigma = (l_{\text{cur}}, \ldots)}{\langle \Sigma | \mathbf{E}\ [\mathbf{getLabel}] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\mathbf{return}\ l_{\text{cur}}] \rangle}$$

Fig. 4: Evaluation contexts and **getLabel** reduction rule.

$$v ::= \cdots \mid Lb^{\text{TCB}}\ l\ t$$
$$t ::= \cdots \mid \mathbf{label}\ t\ t \mid \mathbf{unlabel}\ t \mid \mathbf{labelOf}\ t$$
$$\tau ::= \cdots \mid \mathit{Labeled}\ \tau$$
$$E ::= \cdots \mid \mathbf{label}\ E\ t \mid \mathbf{unlabel}\ E \mid \mathbf{labelOf}\ E$$

$$\textsc{label}\ \frac{\Sigma = (l_{\text{cur}}, \ldots) \qquad l_{\text{cur}} \sqsubseteq l}{\langle \Sigma | \mathbf{E}\ [\mathbf{label}\ l\ t] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\mathbf{return}\ (Lb^{\text{TCB}}\ l\ t)] \rangle}$$

$$\textsc{unlabel}\ \frac{\Sigma = (l_{\text{cur}}, \ldots) \qquad l'_{\text{cur}} = l_{\text{cur}} \sqcup l \qquad \Sigma' = (l'_{\text{cur}}, \ldots)}{\langle \Sigma | \mathbf{E}\ [\mathbf{unlabel}\ (Lb^{\text{TCB}}\ l\ t)] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ t] \rangle}$$

$$\textsc{labelOf}\ \frac{}{E\ [\mathbf{labelOf}\ (Lb^{\text{TCB}}\ l\ t)] \longrightarrow E\ [l]}$$

Fig. 5: Extending base $\lambda^{\text{IO}}_{\ell}$ with labeled values.

precisely this label that is used to restrict the reads/writes performed by the current computation. The rest of the reduction rules for the base calculus are straightforward and given Appendix A.

8.2.2 Labeled values

Using l_{cur} as the label on all terms in scope makes it trivial to deal with implicit flows. Branch conditions, which are simply values of type *Bool*, are already implicitly labeled with l_{cur}. Consequently, all the subsequent writes cannot leak this bit—the current label restricts all the possible writes. However, this coarse-grained labeling approach suffers from a severe restriction: a piece of code cannot, for example, write a public value (e.g., 42) to a public channel labeled **L** after observing secret data, even if the value is independent from the secret—once secret data is read, the current label is raised to **H** thereby "over tainting" the public data in scope.

To address this limitation, LIO provides *Labeled* values. A *Labeled* value is a term that is explicitly protected by a label, other than the current label. Figure 5 shows the extension of the base $\lambda_\ell^{\text{LIO}}$ with *Labeled* values.

The **label** terminal is used to explicitly label a term. As rule (LABEL) shows, **label** $l\ t$ associates the supplied label l with term t by wrapping the term with the Lb^{TCB} constructor. Importantly, it first asserts that the new label (l), which will be used to protect t, is at least as restricting as the current label, i.e., $l_{\text{cur}} \sqsubseteq l$.

Dually, terminal **unlabel** unwraps explicitly labeled values. As defined in rule (UNLABEL), given a labeled value $Lb^{\text{TCB}}\ l\ t$, **unlabel** returns the wrapped term t. Since the returned term is no longer explicitly labeled by l, and is instead protected by the current label, l_{cur} must be at least as restricting as l. To ensure this, the current label is raised from l_{cur} to $l_{\text{cur}} \sqcup l$, capturing the fact that the remaining computation might depend on t. This rule highlights the fact that the current label always "floats" above the labels of the values observed by the current computation.

The **labelOf** function provides a means for inspecting the label of a labeled value. As detailed by reduction rule (LABELOF), given a labeled value $Lb^{\text{TCB}}\ l\ t$, the function returns the label l protecting term t. This allows code to check the label of a labeled value before deciding to unlabel it, and thereby raising the current label. It it worth noting that regardless of the current label in the configuration, the label of a labeled value can be inspected—hence labels are effectively "public."[2]

A common problem with dynamic IFC systems is *label creep* [Sabelfeld and Myers, 2003]—the raising of the current label to a point where the computation can no longer do anything useful. To avoid label creep, LIO provides **toLabeled** as a way to allow the current label to be *temporarily* raised during the execution of a given computation. We extend the terms and the pure evaluation context as $t ::= \cdots \mid \textbf{toLabeled}\ t\ t$ and $E ::= \cdots \mid \textbf{toLabeled}\ E\ t$, respectively, and give the precise semantics for **toLabeled** as follows:

$$\frac{\begin{array}{c}\Sigma = (l_{\text{cur}}, \ldots) \qquad l_{\text{cur}} \sqsubseteq l \qquad \langle \Sigma | t \rangle \longrightarrow^* \langle \Sigma' | LIO^{\text{TCB}}\ t' \rangle \\ \Sigma' = (l'_{\text{cur}}, \ldots) \qquad l'_{\text{cur}} \sqsubseteq l \qquad \Sigma'' = \Sigma \ltimes \Sigma'\end{array}}{\langle \Sigma | \mathbf{E}\ [\textbf{toLabeled}\ l\ t] \rangle \longrightarrow \langle \Sigma'' | \mathbf{E}\ [\textbf{label}\ l\ t'] \rangle}\ \text{TOLABELED}$$

If the current label at the point of executing **toLabeled** $l\ t$ is l_{cur}, **toLabeled** evaluates t to completion ($\langle \Sigma | t \rangle \longrightarrow^* \langle \Sigma' | LIO^{\text{TCB}}\ t' \rangle$) and restores the current label to l_{cur}, i.e., **toLabeled** provides a separate context in which t is evaluated. (Here, the state merge function $\ltimes$ is defined as: $\Sigma \ltimes \Sigma' \triangleq \Sigma$, in

[2] Since labeled values can be nested, this only applies to the labels of top-level labeled values. Indeed, even these labels are not public—they are protected by the current label. However, since code can always observe objects labeled at the current label, this is akin to being public.

the next section we present an alternative definition.) We note that returning the result of evaluating t directly (e.g., as $\langle \Sigma | \mathbf{E}\ [\mathbf{toLabeled}\ l\ t] \rangle \longrightarrow \langle \Sigma'' | \mathbf{E}\ [t'] \rangle$) would allow for trivial leaks; thus, **toLabeled** labels t' with l ($\langle \Sigma'' | \mathbf{E}\ [\mathbf{label}\ l\ t'] \rangle$). This effectively states that the result of t is protected by label l, as opposed to the current label (l'_{cur}) at the point t completed. Importantly, this requires that the result not be more sensitive than l, i.e., $l'_{\text{cur}} \sqsubseteq l$.

8.2.3 Labeled references

To complete the description of LIO, we extend the $\lambda^{\text{UO}}_{\ell}$ calculus with mutable, flow-insensitive references. Conceptually, flow-insensitive references are simply mutable *Labeled* values. Like labeled values, the label of a reference is immutable and serves to protect the underlying term. The immutable label makes the semantics straightforward: writing a term to a reference amounts to ensuring that the reference label is as restrictive as the current label, i.e., the reference label must be above the current label; reading from a reference taints the current label with the reference label.

The syntactic extensions to our calculus are shown in Figure 6. We use meta-variable s to distinguish flow-insensitive (FI) and flow-sensitive (FS) productions—the latter are described in Section 8.3. We also extend configurations to contain a reference (memory) store μ_{FI}: $\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \ldots)$; μ_{FI} maps memory addresses—spanned over by metavariable a—to *Labeled* values.

When creating a flow-insensitive reference, $\mathbf{newRef}_{\text{FI}}\ l\ t$ creates a labeled value that guards t with label l ($Lb^{\text{TCB}}\ l\ t$) and stores it in the memory store at a fresh address a ($\mu_{\text{FI}}\ [a \mapsto Lb^{\text{TCB}}\ l\ t]$). Subsequently, the function returns a value of the form $Ref^{\text{TCB}}_{\text{FI}}\ l\ a$ which simply encapsulates the reference label and address where the term is stored. We remark that since any references created within a **toLabeled** block may outlive the **toLabeled** block computation, the merge function used in rule (TOLABELED) must also account for this, i.e., $(l_{\text{cur}}, \mu_{\text{FI}}, \ldots) \ltimes (l'_{\text{cur}}, \mu'_{\text{FI}}, \ldots) = (l_{\text{cur}}, \mu'_{\text{FI}}, \ldots)$.

Rule (READREF-FI) gives the semantics for reading a labeled reference; reading the term stored at address a simply amounts to unlabeling the value $\mu\ (a)$ stored at the underlying address ($\mathbf{unlabel}\ \mu_{\text{FI}}\ (a)$).

Terminal $\mathbf{writeRef}_{\text{FI}}$ is used to update the memory store with a new term. Note that $\mathbf{writeRef}_{\text{FI}}$ *leaves the label of the reference intact*, i.e., the label of a flow-insensitive reference is never changed, but, as rule (WRITEREF-FI) shows in turn, requires the current label to be below the reference label when performing the write ($l_{\text{cur}} \sqsubseteq l$).

Terminal $\mathbf{labelOf}_{\text{FI}}$ has the benefit of allowing code to always inspect the label of a reference.

The purpose of terminal **copyRef** is to copy the contents of one reference to another, without inspecting the contents of either reference. As given by rule (COPYREF), the function copies the contents of a labeled reference into

$$
\begin{array}{rcl}
v & ::= & \cdots \mid Ref_{\text{FI}}^{\text{TCB}}\ l\ a \\
t & ::= & \cdots \mid \textbf{newRef}_s\ t\ t \mid \textbf{writeRef}_s\ t\ t \mid \textbf{readRef}_s\ t \\
 & & \mid \textbf{labelOf}_s\ t \mid \textbf{copyRef}\ t\ t \\
\tau & ::= & \cdots \mid Ref_s\ \tau \\
E & ::= & \cdots \mid \textbf{newRef}_s\ E\ t \mid \textbf{writeRef}_s\ E\ t \mid \textbf{readRef}_s\ E \\
 & & \mid \textbf{labelOf}_s\ E \mid \textbf{copyRef}\ E\ t \mid \textbf{copyRef}\ v\ E
\end{array}
$$

NEWREF-FI

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \ldots) \qquad l_{\text{cur}} \sqsubseteq l \qquad \mu'_{\text{FI}} = \mu_{\text{FI}}\,[a \mapsto Lb^{\text{TCB}}\ l\ t] \qquad \Sigma' = (l_{\text{cur}}, \mu'_{\text{FI}}, \ldots)}{\langle \Sigma | \mathbf{E}\ [\textbf{newRef}_{\text{FI}}\ l\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{return}\ (Ref_{\text{FI}}^{\text{TCB}}\ l\ a)] \rangle}\ \text{fresh}(a)$$

READREF-FI

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \ldots)}{\langle \Sigma | \mathbf{E}\ [\textbf{readRef}_{\text{FI}}\ (Ref_{\text{FI}}^{\text{TCB}}\ l\ a)] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\textbf{unlabel}\ \mu_{\text{FI}}\ (a)] \rangle}$$

WRITEREF-FI

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \ldots) \qquad l_{\text{cur}} \sqsubseteq l \qquad \mu'_{\text{FI}} = \mu_{\text{FI}}\,[a \mapsto Lb^{\text{TCB}}\ l\ t] \qquad \Sigma' = (l_{\text{cur}}, \mu'_{\text{FI}}, \ldots)}{\langle \Sigma | \mathbf{E}\ [\textbf{writeRef}_{\text{FI}}\ (Ref_{\text{FI}}^{\text{TCB}}\ l\ a)\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{return}\ ()] \rangle}$$

LABELOF-FI

$$\frac{}{E\ [\textbf{labelOf}_{\text{FI}}\ (Ref_{\text{FI}}^{\text{TCB}}\ l\ a)]) \longrightarrow E\ [l]}$$

COPYREF

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \ldots) \qquad l_1 \sqsubseteq l_2 \qquad l_{\text{cur}} \sqsubseteq l_2 \qquad Lb^{\text{TCB}}\ l_1\ v_1 = \mu_{\text{FI}}\ (a_1) \qquad \mu'_{\text{FI}} = \mu_{\text{FI}}\,[a_2 \mapsto Lb^{\text{TCB}}\ l_2\ v_1] \qquad \Sigma' = (l_{\text{cur}}, \mu'_{\text{FI}}, \ldots)}{\langle \Sigma | \mathbf{E}\ [\textbf{copyRef}\ (Ref_{\text{FI}}^{\text{TCB}}\ l_1\ a_1)\ (Ref_{\text{FI}}^{\text{TCB}}\ l_2\ a_2)] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{return}\ ()] \rangle}$$

Fig. 6: Extending $\lambda_\ell^{\text{LIO}}$ with references.

another one, as long as the source-reference label (l_1) flows to the target-reference label (l_2) and the usual condition for writing to an entity with label l_2 also holds ($l_{\text{cur}} \sqsubseteq l_2$). Since the computation does not read the source reference, the current label remains unchanged. We remark that while **copyRef** can be encoded using **toLabeled**, we introduce **copyRef** explicitly since the use of **toLabeled** is prohibited in concurrent settings and our results rely on such a feature in both contexts (see Section 8.5).

8.3 Flow-sensitivity extensions

The flow-insensitive references described in the previous section are inflexible. Consider, for example, an application that uses a reference as a log. Since the log may contain sensitive information, it is important that the reference be labeled. Equally important is to be able to read the log at any point in the program to, for instance, save it to a file. Although labeling the reference with the top element in the security lattice ($\top$) would always allow writes to the log, and **toLabeled** can be used to read the log and then write it to a file, this is unsatisfactory: it assumes the existence of a top element, which in some practical IFC systems, including HiStar [Zeldovich et al., 2006] and Hails [Giffin et al., 2012], does not exist. Moreover, it almost always over-approximates the sensitivity of the log. Hence, for example, a computation that never reads sensitive data, yet wishes to read the log content as to send error message to a user over the network (e.g., as done in a web application) cannot do so—LIO prevents the computation from reading the log, which results in the computation getting tainted by $\top$, and subsequently writing to the network.[3] It is clear that even for such a simple use case, having references with labels that vary according to the sensitivity of what is stored in the reference is useful.

However, naively implementing flow-sensitive references can effectively introduce label changes as a covert channel. Suppose that we allow for the label of a reference to be raised to the current label at the time of the **writeRef**. So, for example, if the label of our log reference is **L** and the computation has read sensitive data (such that the current label is **H**), subsequently writing to the log will raise the label of the reference to **H**. Unfortunately, while this may appear safe, as previously shown in [Austin and Flanagan, 2009, Russo and Sabelfeld, 2010, Austin and Flanagan, 2010], the approach is unsound.

The code fragment in Figure 7 defines a function, *leakRef*, that can be used to leak the contents of a reference by leveraging the newly introduced covert channel: the label of references. (In this and future examples we use function **when** to denote an **if** statement without the **else** branch and ($) as lightweight notation for function application, i.e., $f \$ x$ is the same as $f\ (x)$.) To illustrate an attack, suppose that the current label is public (**L**)

[3] Here, as in most IFC systems, we assume the network is public.

```
leakRef :: Ref_FS Bool → LIO Bool
leakRef href = do
  tmp ← newRef L ()
  toLabeled H $ do h ← readRef href
                   when h $ writeRef tmp ()
  return $ labelOf tmp ≡ H
```

Fig. 7: Attack on LIO with naive treatment of flow-sensitive references. We omit subscripts for clarity.

and *leakRef* is called with a secret (**H**) reference (*href*). *leakRef* first creates a public reference *tmp* and, then, within the **toLabeled** block—which is used to ensure that the current label remains **L**—the label of this reference is changed to **H** if the secret stored in *href* is *True*, and left intact (**L**) if the secret is *False*. The value stored in *href* is revealed by simply inspecting the label of the *tmp* reference.[4]

Fundamentally, the label protecting the *label* of an object, such as a reference or labeled value, is the current label l_{cur} at the time of creation. Hence, to modify the label of the object within some context (e.g., **toLabeled** block) wherein the current label is l'_{cur}, *it must be the case that* $l'_{\text{cur}} \sqsubseteq l_{\text{cur}}$, i.e., we must be able to write data at sensitivity level l'_{cur} into an entity—the label of the object—labeled l_{cur}. This restriction is especially important if $l_{\text{cur}} \sqsubset l'_{\text{cur}}$ and we can restore the current label from l'_{cur} to l_{cur}, since a leak would then be observable within the program itself. In the case where the label of the object is immutable, as is the case for flow-insensitive references (and labeled values), this is not a concern: even if the current label is raised to l'_{cur} and then restored to l_{cur}, we do not learn any information more sensitive than l_{cur}—the label of the label at the time of creation—by inspecting the label of the reference (or value): the label has not changed!

Thus, to extend LIO with flow-sensitive references, we must account for the label on the label of the reference at the time of creation, l_{cur}. (This label is, however, immutable.) In turn, when changing the label of the reference, we must ensure that no data from the context at the time of the change, whose label is l'_{cur}, is leaked into the label of the reference by ensuring that $l'_{\text{cur}} \sqsubseteq l_{\text{cur}}$, i.e., we can write data labeled l'_{cur} into the label that is labeled l_{cur}.

Formally, we extend the $\lambda^{\text{LIO}}_{\ell}$ syntax and reduction rules as shown in Figure 8; we call this calculus $\lambda^{\text{LIO}}_{\ell,\text{FS}}$. To create a flow-sensitive reference $\mathbf{newRef}_{\text{FS}}\ l\ t$ creates a labeled value that guards t with label l ($Lb^{\text{TCB}}\ l\ t$). Since we wish to allow programmers to modify the label l of the reference, we additionally store the label on l, i.e., the current label l_{cur}, by simply

[4] The use of **labelOf** is not fundamental to this attack and in Appendix B we show an alternative attack that does not rely on such label inspection.

$$
\begin{array}{rcl}
v & ::= & \cdots \mid Ref_{\text{FS}}^{\text{TCB}}\ t \\
t & ::= & \cdots \mid \mathbf{upgrade}_{\text{FS}}\ t\ t \mid \Uparrow \mid \mathbf{downgrade}_{\text{FS}}\ t\ t \\
E & ::= & \cdots \mid \mathbf{upgrade}_{\text{FS}}\ E\ t \mid \mathbf{upgrade}_{\text{FS}}\ v\ E \\
 & & \mid \mathbf{downgrade}_{\text{FS}}\ E\ t \mid \mathbf{downgrade}_{\text{FS}}\ v\ E
\end{array}
$$

NEWREF-FS

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad l_{\text{cur}} \sqsubseteq l \qquad \text{fresh}(a) \qquad \mu'_{\text{FS}} = \mu_{\text{FS}}\,[a \mapsto Lb^{\text{TCB}}\ l_{\text{cur}}\ (Lb^{\text{TCB}}\ l\ t)] \qquad \Sigma' = (l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}})}{\langle \Sigma | \mathbf{E}\,[\mathbf{newRef}_{\text{FS}}\ l\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\,[\mathbf{return}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)] \rangle}$$

READREF-FS

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l'\ t) \qquad l'' = l \sqcup l'}{\langle \Sigma | \mathbf{E}\,[\mathbf{readRef}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\,[\mathbf{unlabel}\ (Lb^{\text{TCB}}\ l''\ t)] \rangle}$$

WRITEREF-FS

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l'\ t') \qquad l_{\text{cur}} \sqsubseteq (l \sqcup l') \qquad \mu'_{\text{FS}} = \mu_{\text{FS}}\,[a \mapsto Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l'\ t)] \qquad \Sigma' = (l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}})}{\langle \Sigma | \mathbf{E}\,[\mathbf{writeRef}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\,[\mathbf{return}\ ()] \rangle}$$

WRITEREF-FS-FAIL

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l'\ t') \qquad l_{\text{cur}} \not\sqsubseteq (l \sqcup l')}{\langle \Sigma | \mathbf{E}\,[\mathbf{writeRef}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\,[\mathbf{unlabel}\ (Lb^{\text{TCB}}\ l \Uparrow)] \rangle}$$

LABELOF-FS

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l'\ t)}{\langle \Sigma | \mathbf{E}\,[\mathbf{labelOf}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\,[\mathbf{unlabel}\ (Lb^{\text{TCB}}\ l\ l')] \rangle}$$

UPGRADEREF

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l''\ v) \qquad l_{\text{cur}} \sqsubseteq l \qquad \mu'_{\text{FS}} = \mu_{\text{FS}}\,[a \mapsto Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ (l'' \sqcup l')\ v)] \qquad \Sigma' = (l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}})}{\langle \Sigma | \mathbf{E}\,[\mathbf{upgrade}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)\ l'] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\,[\mathbf{return}\ ()] \rangle}$$

DOWNGRADEREF

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l''\ v) \qquad l_{\text{cur}} \sqsubseteq l \qquad \mu'_{\text{FS}} = \mu_{\text{FS}}\,[a \mapsto Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ (l \sqcup (l'' \sqcap l'))\Uparrow)] \qquad \Sigma' = (l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}})}{\langle \Sigma | \mathbf{E}\,[\mathbf{downgrade}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)\ l'] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\,[\mathbf{return}\ ()] \rangle}$$

Fig. 8: $\lambda^{\text{LIO}}_{\ell,\text{FS}}$: extension of $\lambda^{\text{LIO}}_{\ell}$ with flow-sensitive references.

labeling the already-guarded term ($\mu'_{\text{FS}} = \mu_{\text{FS}}\,[a \mapsto Lb^{\text{TCB}}\ l_{\text{cur}}\ (Lb^{\text{TCB}}\ l\ t)]$), as shown in rule (newRef-fs). Primitive **newRef**$_{\text{FS}}$ returns a $Ref^{\text{TCB}}_{\text{FS}}\ a$ which simply encapsulates the fresh reference address where the doubly-labeled term is stored. Different from the constructor $Ref^{\text{TCB}}_{\text{FI}}$, the constructor $Ref^{\text{TCB}}_{\text{FS}}$ does not encapsulate the label of the reference. This is precisely because the label of a flow-sensitive reference is mutable and must be looked up in the store. As given by rule (labelOf-fs), **labelOf**$_{\text{FS}}$ returns the label of the reference after raising the current label (with **unlabel**) to account for the fact that the label of the reference l' is a value at sensitivity level l, i.e., we raise the current label to the join of the current label and the label on the label.

The rule for reading flow-sensitive references is standard. As given by rule (readRef-fs), **readRef**$_{\text{FS}}$ simply raises the current label to the join of the reference label and label on the reference label ($l \sqcup l'$) and returns the protected value. This reflects the fact that the computation is observing both data at level l (the label on the reference) and l' (the actual term).

The rule for writing flow-sensitive references deserves more attention. First, **writeRef**$_{\text{FS}}$ as given by rule (writeRef-fs), ensures that the current computation can write to the reference by checking that $l_{\text{cur}} \sqsubseteq (l \sqcup l')$. We impose this condition instead of the two conditions $l_{\text{cur}} \sqsubseteq l$ and $l_{\text{cur}} \sqsubseteq l'$—which respectively check that the current computation can modify both, the label of the reference, and the reference itself—since it is more permissive, yet still safe. When imposing the two conditions independently, certain programs, such as the one given in Figure 9, would fail. In this program, we first create a flow-sensitive reference labeled **H** when the current label is **L** (and thus the label on **H** is **L**). Then, we raise the label by reading from the reference. Finally, we attempt to write to the reference. Under our semantics, this program behaves as expected; however, when imposing the two conditions independently, the write fails—the current label does not flow to the label on the label of the reference.

```
do r ← newRef_FS H ()
   readRef_FS r
   writeRef_FS r ()
```

Fig. 9: Permissiveness test.

Another case for **writeRef**$_{\text{FS}}$ which we must handle is when current label does not flow to the join of the reference label, i.e., $l_{\text{cur}} \not\sqsubseteq l \sqcup l'$, and the write is disallowed. If the semantics simply got stuck, the current label (at the point of the stuck term) would not reflect the fact that the success of applying such rule depends on the label l', which is itself protected by l. Indeed, this might lead to information leaks and we thus provide an explicit rule, (writeRef-fs-fail), for this failure case that first raises the current label (via **unlabel**) to l and then diverges; in the rule, $\Uparrow$ represents a divergent term for which we do not provide a reduction rule.

UPGRADESTORE

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu_{\text{FS}} = \{\, a_1 \mapsto v_1, ..., a_n \mapsto v_n \,\} \qquad t_i = \textbf{upgrade}_{\text{FS}}\ (Ref^{\text{TCB}}_{\text{FS}}\ a_i)\ l, i = 1, ..., n}{\langle \Sigma | \mathbf{E}\ [\textbf{upgradeStore}_{\text{FS}}\ l] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [t_1 \gg ... \gg t_n] \rangle}$$

UNLABEL-AU

$$\frac{\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad l'_{\text{cur}} = l_{\text{cur}} \sqcup l \qquad \langle \Sigma | \textbf{upgradeStore}_{\text{FS}}\ l'_{\text{cur}} \rangle \longrightarrow^* \langle l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}} | LIO^{\text{TCB}}\ () \rangle \qquad \Sigma' = (l'_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}})}{\langle \Sigma | \mathbf{E}\ [\textbf{unlabel}\ (Lb^{\text{TCB}}\ l\ t)] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\textbf{return}\ t] \rangle}$$

Fig. 10: $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$: Extending $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ with auto-upgrades.

Note that $\textbf{writeRef}_{\text{FS}}$ does not modify the label of the reference. This is, in part, because we wish to keep the difference between flow-insensitive and flow-sensitive references as small as possible. Instead, we provide $\textbf{upgrade}_{\text{FS}}$ precisely for this purpose; this primitive is used to raise the label of a reference. Rule (UPGRADEREF) is straight forward—it simply ensures that the current computation can modify the label of the reference by checking that the current label flows to the label on the label ($l_{\text{cur}} \sqsubseteq l$). Similarly, $\textbf{downgrade}_{\text{FS}}$ is used to lower the label of the reference, destroying its contents, i.e., replacing its value with $\Uparrow$. Rules (UPGRADEREF) and (DOWNGRADEREF) are analogous; the main difference is that the former uses the join operation to combine the old and new labels ($l'' \sqcup l'$), whereas the latter uses the meet operation ($l'' \sqcap l'$). The $\textbf{downgrade}_{\text{FS}}$ primitive is useful when one wishes to store information that is less sensitive into a reference. Both $\textbf{upgrade}_{\text{FS}}$ and $\textbf{downgrade}_{\text{FS}}$ highlight that it is safe to raise or lower the label of a flow-sensitive reference, if the label on the label still flows to the final label in the nested Lb^{TCB} structure.

8.3.1 Automatic upgrades

We can use $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ to implement various applications that rely on flow-sensitive references, even those that rely on policies such as the popular no-sensitive upgrades [Austin and Flanagan, 2009]. Using $\lambda^{\text{LIO}}_{\ell,\text{FS}}$, we can also safely implement our logging application using a flow-sensitive reference. Unfortunately, our system (and others like it) requires that we insert **upgrade**s before we raise the current label so that it is possible to write references in a more-sensitive context, e.g., to modify a public reference after reading a secret. In the case of the logging example, we would need to upgrade the label before reading any sensitive data, if we later wish to write to the log.

We provide an extension to $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ that can be used to automatically upgrade references. This extension, called $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$, is given in Figure 10. Intuitively, whenever the current label is about to be raised, we first upgrade

$$
\begin{array}{rl}
v & ::= \cdots \mid \overline{v, \ldots} \mid \epsilon_{\overline{\tau,\ldots}} \\
t & ::= \cdots \mid \overline{t, \ldots} \mid \mathbf{withRefs}_{\text{FS}}\ v\ t \\
\tau & ::= \cdots \mid \overline{\tau, \ldots} \\
E & ::= \cdots \mid \overline{E, t, \ldots} \mid \overline{v, E, t, \ldots}
\end{array}
$$

$$
\begin{array}{ll}
addrs(\epsilon_{\overline{\tau,\ldots}}) & \triangleq \emptyset \\
addrs(\overline{Ref_{\text{FS}}^{\text{TCB}}\ a_1, Ref_{\text{FS}}^{\text{TCB}}\ a_2, \ldots}) & \triangleq \{a_1, a_2, \ldots\} \\
addrs^{+}_{\mu}(\epsilon_{\overline{\tau,\ldots}}) & \triangleq \emptyset \\
addrs^{+}_{\mu}(\overline{v_1, v_2, \ldots}) & \triangleq \bigcup \{addrs^{+}_{\mu}(v_1), addrs^{+}_{\mu}(v_2), \ldots\} \\
addrs^{+}_{\mu}(Ref_{\text{FS}}^{\text{TCB}}\ a) & \triangleq \{a\} \cup addrs^{+}_{\mu}(\mu\ (a)) \\
addrs^{+}_{\mu}(v) & \triangleq \emptyset
\end{array}
$$

WITHREFS-CTX

$$
\frac{\begin{array}{c}
\Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \mu'_{\text{FS}} = \{a \mapsto \mu_{\text{FS}}\ (a) \mid a \in \text{dom}\ \mu_{\text{FS}} \cap (addrs^{+}_{\mu_{\text{FS}}}(v))\} \\
\langle l_{\text{cur}}, \mu_{\text{FI}}, \mu'_{\text{FS}} | \mathbf{E}\ [t] \rangle \longrightarrow \langle l'_{\text{cur}}, \mu'_{\text{FI}}, \mu''_{\text{FS}} | \mathbf{E}\ [t'] \rangle \\
\Sigma'' = (l'_{\text{cur}}, \mu'_{\text{FI}}, \mu''_{\text{FS}} \Lsh \mu_{\text{FS}}) \qquad v' = addrs^{-1}(\text{dom}\ \mu''_{\text{FS}})
\end{array}}{\langle \Sigma | \mathbf{E}\ [\mathbf{withRefs}_{\text{FS}}\ v\ t] \rangle \longrightarrow \langle \Sigma'' | \mathbf{E}\ [\mathbf{withRefs}_{\text{FS}}\ v'\ t'] \rangle}
$$

WITHREFS-DONE

$$
\frac{}{\langle \Sigma | \mathbf{E}\ [\mathbf{withRefs}_{\text{FS}}\ v\ v'] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [v'] \rangle}
$$

TYPE-WITHREF

$$
\frac{\begin{array}{c}
\Delta' = \{a \mapsto \Delta\ (a) \mid a \in \text{dom}\ \Delta \cap (addrs(v))\} \\
\Delta', \Gamma \vdash v : \overline{Ref_{\text{FS}}\ \tau_1, \ldots} \qquad \Delta', \Gamma \vdash t : LIO\ \tau
\end{array}}{\Delta, \Gamma \vdash \mathbf{withRefs}_{\text{FS}}\ v\ t : LIO\ \tau}
$$

Fig. 11: Extending $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ and $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ with $\mathbf{withRefs}_{\text{FS}}$.

all the references in the μ_{FS} store and then raise the current label. Rule (UPGRADESTORE) upgrades every reference in the flow-sensitive store μ_{FS} by executing $t_1 \gg t_2 \gg \cdots \gg t_n$, where $t_i = \mathbf{upgrade}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a_i)\ l$. Term $t \gg t'$ is similar to bind except that it discards the result produced by t. Since **unlabel** is the only function that raises the current label, we augment the (UNLABEL) rule with (UNLABEL-AU), given in Figure 10. This ensures that as the computation progresses it does not "lose" write access to its references. Returning to our logging example, with auto-upgrades the reference used as the log never needs to be explicitly upgraded and can always be written to—an interface expected of a log.

Recall that **toLabeled** is used to avoid label creep by allowing code to only temporarily raise the current label. Unfortunately, with auto-upgrades, when the current label gets raised within a **toLabeled** block, the upgrades of the flow-sensitive references remain even after the current label is restored. Thus, reading from any flow-sensitive reference after the **toLabeled** block will raise the current label to (at least) the current label at the end of the **toLabeled** block (since all references are upgraded every time the

current label gets raised). This can be used to carry out a *poison pill*-like attack [Hritcu et al., 2013], wherein the (usually untrusted) computation executing within the **toLabeled** block will render the outer computation useless via label creep. (We note that this attack is possible in $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$ without the auto-upgrade, but requires the attacker to manually insert all the upgrades.)

To address this issue, we extend $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$ (and $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS+AU}}$) with $\mathbf{withRefs}_{\mathrm{FS}}\ v\ t$, which takes a bag (strict heterogeneous list) v of references and a computation t, and executes t in a configuration where the flow-sensitive reference store only contains the subset of references v (and any nested references). This extension and type rule (Type-withRef), which ensures that a term cannot access a reference outside its store, are shown in Figure 11.

A bag is either empty $\epsilon_{\overline{\tau,\ldots}}$, or it may contain a set of references of (potentially) distinct types $\overline{v,\ldots}$. Rules (withRefs-Ctx) and (withRefs-Done) precisely define the semantics of this new primitive, where the meta-level function $addrs(\cdot)$ converts a bag of references to a set of their corresponding addresses, $addrs^{-1}(\cdot)$ performs the inverse conversion, and $\ltimes$ is used to merge the stores, giving preferences to the left-hand-side store, i.e., when there is a discrepancy on a stored value between both stores, it chooses the one appearing on the left-hand-side. The function $addrs^{+}_{\mu}(\cdot)$ computes the closure of $addrs(\cdot)$ under store μ, so as to include the addresses of arbitrarily-nested references. Note that if we did not include these addresses in the restricted store μ'_{FS}, evaluation might get stuck if the program attempted a $\mathbf{readRef}_{\mathrm{FS}}$ operation on a nested reference. We note that (withRefs-Ctx) is triggered until the term under evaluation is reduced to a value, at which point (withRefs-Done) is triggered, returning said value; we specify this big-step rule in terms of small-steps to facilitate the formalization of our concurrent calculus (see Section 8.4). Aside from the modeling of bags, the $\mathbf{withRefs}_{\mathrm{FS}}$ primitive is straightforward and mostly standard; indeed, the programming paradigm is similar to that already present in some mainstream languages (e.g., C++'s lambda closures require the programmer to specify the captured references). Lastly, we note that the poison pill attack can now be addressed by simply wrapping **toLabeled** with $\mathbf{withRefs}_{\mathrm{FS}}$, which prevents (untrusted) code within the **toLabeled** block from upgrading arbitrary references.

8.4 Concurrency

In this section, we consider flow-sensitive references in the presence of concurrency (e.g., a web application in which request-handling threads share a common log). Concretely, we extend our sequential $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$ and $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS+AU}}$ calculi with threads and a new terminal, **forkLIO**, for dynamically creating new threads, as in the concurrent version of LIO [Stefan et al., 2012a]. Intuitively, this concurrent calculus $\lambda^{\parallel\text{-}\mathsf{LIO}}_{\ell}$ simply defines a scheduler over se-

$$t ::= \cdots \mid \textbf{forkLIO}\ t \mid \cancel{\textbf{toLabeled}}\ \mathrm{t}\ \mathrm{t}$$

$$\textsc{forkLIO}\quad \frac{}{\langle \Sigma | \mathbf{E}\ [\textbf{forkLIO}\ t]\rangle \xrightarrow{fork(t)} \langle \Sigma | \mathbf{E}\ [\textbf{return}\ ()]\rangle}$$

$$\textsc{withRefs-Opt}\quad \frac{\begin{array}{c} v = addrs^{-1}((addrs(v_1)) \cap (addrs(v_2))) \\ \langle \Sigma | \mathbf{E}\ [\textbf{withRefs}_{\text{FS}}\ v\ t] \longrightarrow \langle \Sigma' | \mathbf{E}\ [t']\rangle\rangle \end{array}}{\langle \Sigma | \mathbf{E}\ [\textbf{withRefs}_{\text{FS}}\ v_1\ (\textbf{withRefs}_{\text{FS}}\ v_2\ t)]\rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [t']\rangle}$$

$$\textsc{T-step}\quad \frac{\begin{array}{c} \Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \langle \Sigma | \textbf{withRefs}_{\text{FS}}\ v\ t\rangle \longrightarrow \langle \Sigma' | t'\rangle \\ \Sigma' = (l'_{\text{cur}}, \mu'_{\text{FI}}, \mu'_{\text{FS}}) \qquad v' = addrs^{-1}(\text{dom}\ \mu'_{\text{FS}}) \end{array}}{\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{\text{cur}}, v, t\rangle, k_2, ...\} \longrightarrow \{\mu'_{\text{FI}}, \mu'_{\text{FS}} | k_2, ..., \langle l'_{\text{cur}}, v', t'\rangle\}}$$

$$\textsc{T-stuck}\quad \frac{}{\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{\text{cur}}, v, \Uparrow\rangle, k_2, ...\} \longrightarrow \{\mu_{\text{FI}}, \mu_{\text{FS}} | k_2, ...\}}$$

$$\textsc{T-done}\quad \frac{}{\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{\text{cur}}, v, v'\rangle, k_2, ...\} \longrightarrow \{\mu_{\text{FI}}, \mu_{\text{FS}} | k_2, ...\}}$$

$$\textsc{T-fork}\quad \frac{\begin{array}{c} \Sigma = (l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}}) \qquad \langle \Sigma | \textbf{withRefs}_{\text{FS}}\ v\ t\rangle \xrightarrow{fork(t')} \langle \Sigma' | t''\rangle \\ \Sigma' = (l'_{\text{cur}}, \mu'_{\text{FI}}, \mu'_{\text{FS}}) \qquad v' = addrs^{-1}(\text{dom}\ \mu'_{\text{FS}}) \qquad k_{\text{new}} = \langle l'_{\text{cur}}, v', t'\rangle \end{array}}{\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{\text{cur}}, v, t\rangle, k_2, ...\} \longrightarrow \{\mu'_{\text{FI}}, \mu'_{\text{FS}} | k_2, ..., \langle l'_{\text{cur}}, v', t''\rangle, k_{\text{new}}\}}$$

Fig. 12: Semantics for $\lambda_\ell^{\parallel\text{-LIO}}$, parametric in the flow-sensitivity policy, i.e., with and without auto-upgrade.

quential threads, such that taking a step in the concurrent calculus amounts to taking a step in a sequential thread and context switching to a different one. For brevity, we restrict our discussion in this section to the case where the underlying sequential calculus is $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$, since this calculus extends $\lambda^{\text{LIO}}_{\ell,\text{FS}}$.

Figure 12 shows our extended concurrent calculus, $\lambda_\ell^{\parallel\text{-LIO}}$. A concurrent program configuration has the form $\{\mu_{\text{FI}}, \mu_{\text{FS}} | k_1, k_2, ...\}$, where μ_{FI} and μ_{FS} are respectively the flow-insensitive and flow-sensitive stores shared by all the threads $k_1, k_2, ...$ in the program. Since the memory stores are global, a thread k is simply a tuple encapsulating the current label of the thread l_{cur}, the term under evaluation t, and a bag of references v the thread may access, i.e., $k = \langle l_{\text{cur}}, v, t\rangle$.

The reduction rules for concurrent programs are mostly standard. Rule (T-step) specifies that if the first thread in the thread pool takes a step in $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$, the whole concurrent program takes a step, moving the thread

to the end of the pool. We note that the thread term t executed with its stored current label l_{cur}, and a subset of the flow-sensitive memory store, by wrapping it in $\textbf{withRefs}_{\text{FS}}$. While the use of $\textbf{withRefs}_{\text{FS}}$ makes the extension straightforward, one peculiarity arises: since (T-STEP) always wraps the thread term t with $\textbf{withRefs}_{\text{FS}}$, if t does not reduce in one step to a value, and instead reduces to a term t', the next time the thread is scheduled, we will superfluously wrap $\textbf{withRefs}_{\text{FS}}\ t'$ with yet another $\textbf{withRefs}_{\text{FS}}$—thus preventing the thread from making progress! To address this problem, we extend the calculus with rule (WITHREFS-OPT) that collapses nested $\textbf{withRefs}_{\text{FS}}$ blocks.[5]

Rules (T-DONE) and (T-STUCK) specify that once a thread term has reduced to a value or got stuck, which is represented by $\Uparrow$, the scheduler removes it from the thread pool and schedules the next thread.

As shown in Figure 12, to allow for dynamic thread creation, we extend $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$'s terms with **forkLIO**, and add a new reduction rule that sends an event to the scheduler, specifying the term to execute in a new thread.[6] Rule (T-FORK) describes the corresponding scheduler rule, triggered when a $\mathit{fork}\ (t')$ event is received. Here, we create a new thread k_{new} whose current label l'_{cur} and partition of the store, i.e., bag of references v', is the same as that of the parent thread; the term evaluated in the newly created thread is provided in the event: t'. Subsequently, we add the new thread to the thread pool.

The final modification in extending $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ to $\lambda^{\parallel\text{-LIO}}_{\ell}$ is the removal of **toLabeled** from the underlying calculus. As described in [Stefan et al., 2012a], we must remove **toLabeled** to guarantee termination-sensitive non-interference. Importantly, however, **forkLIO** with synchronization primitives (e.g., flow-insensitive labeled MVars, as discussed in [Stefan et al., 2012a]) can be used to provide functionality equivalent to that of **toLabeled**. We omit synchronization primitives from $\lambda^{\parallel\text{-LIO}}_{\ell}$; we only remark that extending $\lambda^{\parallel\text{-LIO}}_{\ell}$ to provide flow-sensitive labeled MVars follows in a straightforward way.

Since the flow-sensitive attack in Figure 7 relied on **toLabeled** to restore the current label, a natural question, given that we remove **toLabeled**, is whether we can use the naive flow-sensitive reference semantics of Section 8.3 for concurrent LIO. As shown by the attack code in Figure 13, in which we use **forkLIO** instead of **toLabeled** to address a potential label creep, the fundamental problem remains: the label on the reference label is not protected! This precisely motivated our principled approach of ex-

[5] This change also requires modifying (WITHREFS-CTX) to not be triggered when the term being evaluated is a $\textbf{withRefs}_{\text{FS}}$ term.

[6] In fact, the reduction rule for $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ must be changed to account for events. However, since *fork* is the only event in our system, we treat $\longrightarrow$ as implicitly carrying an empty event.

```
leakRef :: Ref^TCB Bool → LIO Bool
leakRef href = do
  tmp ← newRef L ()
  forkLIO $ do h ← readRef href
               when h $ writeRef tmp ()
  delay
  return $ labelOf tmp ≡ H
```

Fig. 13: Attack on concurrent LIO with naive flow-sensitive reference extension.

tending $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ to a concurrent setting as opposed to extending concurrent LIO with flow-sensitive references.

8.5 Formal results

In this section, we show that our flow-sensitive enforcement can be embedded into the flow-insensitive version of LIO. Additionally, we provide security guarantees in terms of non-interference definitions by reusing previous results on LIO.

8.5.1 Embedding into $\lambda^{\text{LIO}}_{\ell}$

Every flow-sensitive reference with label l_d created in a context where the current label is l_o (and thus stored in μ_{FS} as $Lb^{\text{TCB}}\ l_o\ (Lb^{\text{TCB}}\ l_d\ t)$), can be represented by a flow-insensitive reference with label l_o, whose contents are another flow-insensitive reference containing t and labeled l_d.

Figure 14 gives our encoding of the flow-sensitive reference operations in terms of flow-insensitive references. For a given store Σ, we define the $[\![-]\!]^{\Sigma}_{\text{FI}}$ function, which given a term t in $\lambda^{\text{LIO}}_{\ell,\text{FS}}$, produces a term $[\![t]\!]^{\Sigma}_{\text{FI}}$ in $\lambda^{\text{LIO}}_{\ell}$, expanding the definitions of flow-sensitive operations in terms of flow-insensitive ones. This function is applied homomorphically in all other cases. We use the *WrapRef* constructor to mark the flow-insensitive references that are being used to represent flow-sensitive ones, so as to distinguish them from other flow-insensitive references. The functions *wrap* and *unwrap* are used to add and remove this boundary encoding. In the embedding of $\textbf{writeRef}_{\text{FS}}$, we use **toLabeled** to make any changes to the current label (possibly caused by reading the outer reference) local to this operation. Inside **toLabeled**, the code fetches the inner reference ($\textbf{readRef}_{\text{FI}}$), and then performs the actual write of the new value. If this fails, the computation diverges, but, importantly, the current label was raised (with $\textbf{readRef}_{\text{FI}}$) to reflect the fact that the label on the label of the reference was observed. The embedding of $\textbf{upgrade}_{\text{FS}}$ relies on flow-insensitive primitives to implement

the **upgrade** operation. As in **writeRef**$_{\text{FS}}$, a **toLabeled** block is used to delimit the taint on the current label. Inside the block, the code fetches the inner reference (**readRef**$_{\text{FI}}$), which taints the current label with l', and makes a new reference n (**newRef**$_{\text{FI}}$) with the upgraded label ($l_{\text{cur}} \sqcup (l \sqcup \textbf{labelOf}\ i)$) and an undefined value ($\bot$). Observe that the operation for creating the reference always succeeds since its label is above the current label, i.e., l_{cur}. Then, **copyRef** is used to copy the value of the original inner reference into the new one, n. As before, this action always succeeds because the label of the reference bound to i (**labelOf** i) is below the label of the new reference n. Finally, the reference n is stored in place of the original inner reference using **writeRef**.. Importantly, this instruction only succeeds when the current label at the time of writing, i.e., $lc \sqcup l'$ in Figure 14, is below or equal to l' (the label of the outer reference), i.e., $lc \sqcup l' \sqsubseteq l'$. This restrictions holds when the current label at the time of upgrade, i.e., lc, is below or equal to l'—effectively encoding the non-sensitive upgrade policy for label changes. The embedding of **downgrade**$_{\text{FS}}$ follows similarly, except that the label of the new reference is computed using $\sqcap$ instead of $\sqcup$ (to achieve the downgrade), and the **copyRef** step is omitted, since the original value must be destroyed. We remark that the mapping mimics the behavior described by the rules in Figure 8.

We extend this definition naturally to convert $\lambda^{\text{LIO}}_{\ell,\text{FS}}$ environments into $\lambda^{\text{LIO}}_{\ell}$ environments, by having $[\![(l_{\text{cur}}, \mu_{\text{FI}}, \mu_{\text{FS}})]\!]_{\text{FI}} \triangleq (l_{\text{cur}}, \mu'_{\text{FI}})$ where μ'_{FI} is obtained by extending μ_{FI} with the pair of bindings $a_i \mapsto Lb^{\text{TCB}}\ l_o\ (Ref^{\text{TCB}}_{\text{FI}}\ l_d\ b_i)$, $b_i \mapsto (Lb^{\text{TCB}}\ l_d\ v)$ (with b_i being a fresh name) for each binding of the form $a_i \mapsto Lb^{\text{TCB}}\ l_o\ (Lb^{\text{TCB}}\ l_d\ v_i)$ in μ_{FS}. Note that the domains of μ_{FI} and μ_{FS} are disjoint because the fresh$(\cdot)$ predicate that we use in the semantics is assumed to produce globally unique addresses.

In order to prove that our implementation is correct with respect to the semantics, we show that, if we take a program with flow-sensitive operations, and expand those operations, replacing them by the code in Figure 14, then its behavior corresponds with the flow-sensitive semantics.

Theorem 1 (Embedding $\lambda^{\text{LIO}}_{\ell,\text{fs}}$ in $\lambda^{\text{LIO}}_{\ell}$). *Let t be a well-typed term in $\lambda^{\text{LIO}}_{\ell,\text{FS}}$. Then if $\langle \Sigma | t \rangle \longrightarrow^* \langle \Sigma' | v \rangle$, we have $\langle [\![\Sigma]\!]_{\text{FI}} | [\![t]\!]^{\Sigma}_{\text{FI}} \rangle \longrightarrow^* \langle [\![\Sigma']\!]_{\text{FI}} | [\![v]\!]^{\Sigma}_{\text{FI}} \rangle$, and if $\langle \Sigma | t \rangle \longrightarrow^* \langle \Sigma' | \Uparrow \rangle$, then $\langle [\![\Sigma]\!]_{\text{FI}} | [\![t]\!]^{\Sigma}_{\text{FI}} \rangle \longrightarrow^* \langle [\![\Sigma']\!]_{\text{FI}} | \Uparrow \rangle$.*

Proof. See Appendix C.

While straightforward, this theorem highlights an important result: in floating label systems, flow-sensitive references can be encoded in a calculus with flow-insensitive references and explicitly labeled values.

8.5.2 Security guarantees

From previous results [Stefan et al., 2011], we know that LIO satisfies termination-insensitive non-interference (TINI) in the sequential setting,

$$
\begin{aligned}
&\mathit{wrap}\ r \triangleq \mathit{WrapRef}\ r\\
&\mathit{unwrap}\ (\mathit{WrapRef}\ r) \triangleq r\\
&[\![\mathit{Ref}^{\mathsf{TCB}}_{\mathrm{FS}}\ r]\!]^{(l_{\mathrm{cur}},\mu_{\mathrm{FI}},\mu_{\mathrm{FS}})}_{\mathrm{FI}} \triangleq \mathit{wrap}\ (\mathit{Ref}^{\mathsf{TCB}}_{\mathrm{FI}}\ (\mathbf{labelOf}_{\mathrm{FI}}\ \mu_{\mathrm{FS}}\ (r))\ r)\\
&[\![\mathbf{newRef}_{\mathrm{FS}}]\!]^{\Sigma}_{\mathrm{FI}} \triangleq \lambda l\ t.\mathbf{do}\\
&\quad i \leftarrow \mathbf{newRef}_{\mathrm{FI}}\ l\ t\\
&\quad l_{\mathrm{cur}} \leftarrow \mathbf{getLabel}\\
&\quad o \leftarrow \mathbf{newRef}_{\mathrm{FI}}\ l_{\mathrm{cur}}\ i\\
&\quad \mathbf{return}\ (\mathit{wrap}\ o)\\
&[\![\mathbf{readRef}_{\mathrm{FS}}]\!]^{\Sigma}_{\mathrm{FI}} \triangleq \lambda r.\mathbf{readRef}_{\mathrm{FI}}\ (\mathit{unwrap}\ r) \gg\!= \mathbf{readRef}_{\mathrm{FI}}\\
&[\![\mathbf{writeRef}_{\mathrm{FS}}]\!]^{\Sigma}_{\mathrm{FI}} \triangleq \lambda r\ t.\mathbf{let}\ o = \mathit{unwrap}\ r\ \mathbf{in\ do}\\
&\quad l_{\mathrm{cur}} \leftarrow \mathbf{getLabel}\\
&\quad \mathbf{toLabeled}\ (l_{\mathrm{cur}} \sqcup (\mathbf{labelOf}\ o))\ \$\ \mathbf{do}\\
&\quad\quad i \leftarrow \mathbf{readRef}_{\mathrm{FI}}\ o\\
&\quad\quad \mathbf{writeRef}_{\mathrm{FI}}\ i\ t\\
&\quad \mathbf{return}\ ()\\
&[\![\mathbf{labelOf}_{\mathrm{FS}}]\!]^{\Sigma}_{\mathrm{FI}} \triangleq \lambda r.\\
&\quad \mathbf{readRef}_{\mathrm{FI}}\ (\mathit{unwrap}\ r) \gg\!= \mathbf{return}.\mathbf{labelOf}_{\mathrm{FI}}\\
&[\![\mathbf{upgrade}_{\mathrm{FS}}]\!]^{\Sigma}_{\mathrm{FI}} \triangleq \lambda r\ l.\mathbf{let}\ o = \mathit{unwrap}\ r\\
&\qquad\qquad\qquad\qquad\qquad\qquad l' = \mathbf{labelOf}\ o\ \mathbf{in\ do}\\
&\quad lc \leftarrow \mathbf{getLabel}\\
&\quad \mathbf{toLabeled}\ (lc \sqcup l')\ \$\ \mathbf{do}\\
&\quad\quad i \leftarrow \mathbf{readRef}_{\mathrm{FI}}\ o\\
&\quad\quad l_{\mathrm{cur}} \leftarrow \mathbf{getLabel}\\
&\quad\quad n \leftarrow \mathbf{newRef}_{\mathrm{FI}}\ (l_{\mathrm{cur}} \sqcup (l \sqcup \mathbf{labelOf}\ i))\ \bot\\
&\quad\quad \mathbf{copyRef}\ i\ n\\
&\quad\quad \mathbf{writeRef}_{\mathrm{FI}}\ o\ n\\
&\quad \mathbf{return}\ ()\\
&[\![\mathbf{downgrade}_{\mathrm{FS}}]\!]^{\Sigma}_{\mathrm{FI}} \triangleq \lambda r\ l.\mathbf{let}\ o = \mathit{unwrap}\ r\\
&\qquad\qquad\qquad\qquad\qquad\qquad l' = \mathbf{labelOf}\ o\ \mathbf{in\ do}\\
&\quad lc \leftarrow \mathbf{getLabel}\\
&\quad \mathbf{toLabeled}\ (lc \sqcup l')\ \$\ \mathbf{do}\\
&\quad\quad i \leftarrow \mathbf{readRef}_{\mathrm{FI}}\ o\\
&\quad\quad l_{\mathrm{cur}} \leftarrow \mathbf{getLabel}\\
&\quad\quad n \leftarrow \mathbf{newRef}_{\mathrm{FI}}\ (l_{\mathrm{cur}} \sqcup (l \sqcap \mathbf{labelOf}\ i))\ \bot\\
&\quad\quad \mathbf{writeRef}_{\mathrm{FI}}\ o\ n\\
&\quad \mathbf{return}\ ()\\
&[\![\mathbf{withRefs}_{\mathrm{FS}}\ v\ t]\!]^{(l_{\mathrm{cur}},\mu_{\mathrm{FI}},\mu_{\mathrm{FS}})}_{\mathrm{FI}} \triangleq [\![t]\!]^{(l_{\mathrm{cur}},\mu_{\mathrm{FI}},\mu'_{\mathrm{FS}})}_{\mathrm{FI}}\\
&\quad \mathbf{where}\\
&\quad\quad \mu'_{\mathrm{FS}} = \{\, a \mapsto \mu_{\mathrm{FS}}\ (a) \mid a \in \mathrm{dom}\ \mu_{\mathrm{FS}} \cap (\mathit{addrs}^{+}_{\mu_{\mathrm{FS}}}(v)) \,\}
\end{aligned}
$$

Fig. 14: Implementation mapping for flow-sensitive references. For all other terms, the function is applied homomorphically.

and termination-sensitive non-interference (TSNI) in the concurrent setting. By using the embedding theorem we can extend these results for LIO with flow-sensitive references.

For completeness, we now present our non-interference theorems, as straightforward applications of the theorems in previous work. Our security results rely on the notion of l-equivalence for terms and configurations, which captures the idea of terms that cannot be distinguished by an attacker which can observe data at level l. A pair of terms t_1, t_2 is said to be l-equivalent (written $t_1 \approx_l t_2$) if, after erasing all the information more sensitive than l from t_1 and t_2, we obtain syntactically equivalent terms. This definition extends naturally to configurations.

Intuitively, non-interference means that an attacker at level l cannot distinguish among different runs of a program with l-equivalent initial configurations.

Theorem 2 (**TINI for $\lambda^{\mathsf{LIO}}_{\ell,\mathbf{fs}}$**). *Consider two well-typed terms t_1 and t_2 in $\lambda^{LIO}_{\ell,\mathrm{FS}}$ which do not contain any $\cdot^{TCB}$ syntax nodes, such that $t_1 \approx_l t_2$, where l is the attacker observation level. Let Σ be an initial environment, and let*

$$\langle \Sigma | t_1 \rangle \longrightarrow^* \langle \Sigma_1 | v_1 \rangle \text{ and } \langle \Sigma | t_2 \rangle \longrightarrow^* \langle \Sigma_2 | v_2 \rangle$$

Then, we have that $\langle \Sigma_1 | v_1 \rangle \approx_l \langle \Sigma_2 | v_2 \rangle$.

Proof. By expanding all the flow-sensitive operations in t_1 and t_2 using their definition given in Figure 14, we get terms in $\lambda^{\mathsf{LIO}}_{\ell}$, which by Theorem 1 has equivalent semantics. Therefore, the result follows from the $\lambda^{\mathsf{LIO}}_{\ell}$ TINI result of [Stefan et al., 2011].

Corollary 1 (**TINI for $\lambda^{\mathsf{LIO}}_{\ell,\mathbf{fs+au}}$**). *The previous non-interference result can be easily extended to $\lambda^{LIO}_{\ell,\mathrm{FS+AU}}$. In $\lambda^{LIO}_{\ell,\mathrm{FS+AU}}$, the* **unlabel** *operation triggers the automatic upgrades mechanism, which performs the* **upgrade** *operation for every flow-sensitive reference in scope before actually raising the current label. Regardless of how* **unlabel** *is used, we note that the resulting term (after inserting the necessary* **upgrade**s*), is just an $\lambda^{LIO}_{\ell,\mathrm{FS}}$ term. Therefore, the main TINI result for $\lambda^{LIO}_{\ell,\mathrm{FS}}$ applies.*

For the concurrent result, we need a supporting lemma which states that the current label is always at least as sensitive as the label on the label of every reference in scope. Formally,

Lemma 1. *Consider the predicate*

$$P(ks, \mu_{\mathrm{FS}}) = \forall \langle l, v, t \rangle \in ks. \forall a \in (addrs(v) \cap dom\ \mu_{\mathrm{FS}}). \mathbf{labelOf}\ (\mu_{\mathrm{FS}}\ a) \sqsubseteq l,$$

where ks is a well-typed thread pool in $\lambda^{\|\text{-}LIO}_{\ell}$ and μ_{FS} is a flow-sensitive store. Then, P is invariant *under $\longrightarrow$, that is*

$$P(ks, \mu_{\mathrm{FS}}) \vee \{\mu_{\mathrm{FI}}, \mu_{\mathrm{FS}} | ks\} \longrightarrow \{\mu'_{\mathrm{FI}}, \mu'_{\mathrm{FS}} | ks'\} \Rightarrow P(ks', \mu'_{\mathrm{FS}})$$

Proof. We consider cases on the rule used to derive $\{\mu_{\text{FI}}, \mu_{\text{FS}} | ks\} \longrightarrow \{\mu'_{\text{FI}}, \mu'_{\text{FS}} | ks'\}$.

- **Cases** (T-DONE) and (T-STUCK). Trivial.
- **Case** (T-FORK). We conclude that t must be a **forkLIO** operation, therefore $l'_{\text{cur}} = l_{\text{cur}}$, $v = v'$ and $\mu'_{\text{FS}} = \mu_{\text{FS}}$. Hence P must hold for the thread pool $ks, \langle l'_{\text{cur}}, v', t'' \rangle, k_{\text{new}}$.
- **Case** (T-STEP). By inspection of all reduction rules, we see that no operation changes the label on the label of a reference after it's been created, which means that **labelOf** $(\mu_{\text{FS}}\ a) =$ **labelOf** $(\mu'_{\text{FS}}\ a)$ for all $a \in \text{dom}\ \mu_{\text{FS}} \cap \text{dom}\ \mu'_{\text{FS}}$. Therefore $P(ks, \mu'_{\text{FS}})$ still holds from the hypothesis that $P(ks, \mu_{\text{FS}})$ holds.
  We only need to prove that $\forall a \in v'.$**labelOf** $(\mu'_{\text{FS}}\ a) \sqsubseteq l'_{\text{cur}}$. We consider two cases:
  - **Case** $l'_{\text{cur}} \neq l_{\text{cur}}$. The only reduction rules that modify l_{cur} at all are the read/unlabel rules, so t must be a **readRef** or **unlabel** operation. This means that $l_{\text{cur}} \sqsubseteq l'_{\text{cur}}$ and $v' = v$, so the invariant holds.
  - **Case** $l'_{\text{cur}} = l_{\text{cur}}$ but $v' \neq v$. The only operation that can affect the domain of μ_{FS} is **newRef**., so this means there is a new address a' in dom μ'_{FS} for which we need to check that **labelOf** $(\mu'_{\text{FS}}\ a) \sqsubseteq l'_{\text{cur}}$. The **newRef**. operation will set **labelOf** $(\mu'_{\text{FS}}\ a')$ to l_{cur} and since this is exactly l'_{cur} we have that the invariant holds for a', and it trivially holds for all other addresses in $addrs(v)$ by hypothesis.

We now prove our non-interference theorem for $\lambda_\ell^{\parallel\text{-LIO}}$. This result is stronger than TINI, since it implies that there can be no termination or internal timing leaks.

Theorem 3 (TSNI for $\lambda_\ell^{\parallel\text{-LIO}}$). *Consider two well-typed terms t_1 and t_2 in $\lambda_\ell^{\parallel\text{-LIO}}$ which do not contain any $\cdot^{\text{TCB}}$ syntax nodes, such that $t_1 \approx_l t_2$, where l is the attacker observation level. Let $\Sigma = (l_{cur}, \mu_{\text{FI}}, \mu_{\text{FS}})$ be an initial environment, and let*

$$\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{cur}, addrs^{-1}(dom\ \mu_{\text{FS}}), t_1 \rangle\} \longrightarrow^* M_1$$

Then, there exists some configuration M_2 such that

$$\{\mu_{\text{FI}}, \mu_{\text{FS}} | \langle l_{cur}, addrs^{-1}(dom\ \mu_{\text{FS}}), t_2 \rangle\} \longrightarrow^* M_2$$

and $M_1 \approx_l M_2$.

Proof. We note that the initial stores are empty, so the predicate in Lemma 1 trivially holds and therefore it holds for all subsequent configurations by induction. From Lemma 1 and looking at the embeddings of $\textbf{writeRef}_{\text{FS}}$ and $\textbf{upgrade}_{\text{FS}}$, we note that the first $\textbf{readRef}_{\text{FI}}$ operation in each **toLabeled** block will be trying to raise the current label to l. However, since $l \sqsubseteq l_{\text{cur}}$, these operations will never effectively raise the current label. This means

that using **toLabeled** is not necessary to preserve the semantics, because there is no need to restore the current label afterwards. As a result, and after removing **toLabeled** in these two cases, we note that the embedding produces valid concurrent $\lambda_\ell^{\mathsf{LIO}}$ terms (which does not have **toLabeled**).

Finally, by expanding all the flow-sensitive operations in t_1 and t_2 using their definition given in Figure 14, we get terms in concurrent $\lambda_\ell^{\mathsf{LIO}}$. Therefore, the result follows from the termination-sensitive non-interference of concurrent $\lambda_\ell^{\mathsf{LIO}}$ [Stefan et al., 2012a].

We lastly remark a limitation: while we preserve non-interference when embedding the flow-sensitive calculus in the original LIO, our embedding includes no synchronization to ensure atomicity of the flow-sensitive operations, so certain interleaving that break semantic equivalence are possible.

8.5.3 Permissiveness

In Section 8.7 we compare the permissiveness of our system with previous flow-sensitive IFC systems. Here, we solely remark that the above results imply that our flow-sensitive calculus is as permissive as flow-insensitive LIO. In particular, any flow-insensitive LIO program can be trivially converted to a flow-sensitive LIO program (without auto-upgrades) by using flow-sensitive references instead of flow-insensitive ones. Since these references would never be upgraded, they will behave just like their flow-insensitive counterparts. This means that all existing LIO programs can be run in our flow-sensitive monitor. This includes Hails [Giffin et al., 2012], a web framework using LIO, on top of which a number of applications have been built (e.g., GitStar[7], a code-hosting web platform, LearnByHacking[8], a blog/tutorial platform similar to School of Haskell, and LambdaChair [Stefan et al., 2012c], an EasyChair-like conference review system).

8.6 Comparison with other policies for label change

In this section, we compare our enforcement mechanism with two policies for label change: *no-sensitive-upgrade* (which was originally proposed by Zdancewic [Zdancewic, 2002]) and *permissive-upgrade*, a more permissive version of the former by Austin and Flanagan [Austin and Flanagan, 2009, 2010]. We introduce a simple imperative language to simplify our comparison with the languages implementing the aforementioned policies. This simple language has variables, **if**-statements, a **skip** command that does nothing, and an **output** command that is used to produce public outputs. This language is easily implemented in $\lambda_{\ell,\mathrm{FS}}^{\mathsf{LIO}}$ as syntactic sugar. For example, a conditional statement $\mathbf{if}\ \ C; A\ \ \mathbf{else}\ \ B$ is desugared to

[7] `www.gitstar.com`

[8] `www.learnbyhacking.org`

toLabeled H (**do** $b \leftarrow$ **unlabel** C; **if** b **then** A **else** B). (Here, **toLabeled** is used to ensure that the current label is restored after leaving the **if**-statement.)

8.6.1 No-sensitive-upgrade

The no-sensitive-upgrade discipline stops execution on any attempt to change the label of a public variable inside a secret context. Our $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$ calculus essentially implements this discipline as well—see (UPGRADEREF) in Figure 8. The original presentation of no-sensitive-upgrade allows for variables with a secret label to be downgraded, as long as the original contents are discarded. Our $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$ calculus similarly allows for this with the **downgrade** operation. Our approach differs in also allowing code to explicitly upgrade a variable before entering a secret context, permissively allowing writes to originally-public variables in secret contexts.

8.6.2 Permissive-upgrade

```
upgrade x H
if h
  x := True
if x
  skip
```

Fig. 15: A secure program that $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$ accepts.

The permissive-upgrade policy differs from no-sensitive-upgrade in allowing code to change the label of a public variables in secret contexts, but subsequently disallowing branches on such permissively-upgraded, or "marked," variables. When upgrading a public variable in a secret context, the security label of the variable is changed to **P** where $\mathbf{L} \sqsubseteq \mathbf{H} \sqsubseteq \mathbf{P}$. In general, the permissiveness of our approach is incomparable with that of permissive-upgrade. For example, without the **upgrade** operation, $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$ is as expressive as no-sensitive-upgrade, and thus less permissive than permissive-upgrade. But, with **upgrade** we can write programs in $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$ that would be rejected by a permissive-upgrade monitor. Figure 15 shows an example of one such case. In the example, the **upgrade** operation is used to ensure that reference x, which would be marked **P** by permissive-upgrade, ends up as **H** in all runs; without the **upgrade**, a permissive-upgrade monitor would reject the branch on x. By inserting **upgrade** operations in the "right" places, our approach can become more flexible than permissive-upgrade.

Of course, the challenge lies in upgrading references in a permissive fashion. And automatically upgrading references whenever the current label is raised is not necessarily more permissive than a permissive-upgrade monitor. Indeed, the permissiveness of permissive-upgrade and $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS+AU}}$ are also incomparable.

Figure 16 shows a program that is rejected by permissive-upgrade but accepted by $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS+AU}}$. With permissive-upgrade, the first branch on

h upgrades x to **P**, which leads to a failure when subsequently trying to branch x. With $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$, on the other hand, reference x would be upgraded to **H**, permissively allowing the second branch.

```
if h
  x := True
if x
  skip
```

Fig. 16: A secure program that permissive-upgrade rejects and $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ accepts.

Conversely, Figure 17 shows a secure program that is accepted by permissive-upgrade but rejected by $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$. In this program, there are two variables in scope: x and y, both initially public. In the first secret conditional block we assign to y, which with permissive-upgrade only upgrades variable y to **P**; x remains **L** and thus the second branch is executed, producing a public output. With $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$, however, unlabeling h (an operation implicit in the first conditional, which inspects h) auto-upgrades both x and y to **H**. As a result, the current label at the point of the **output** is **H**, causing a failure.

One way to address the permissiveness issues of $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ is by using $\textbf{withRefs}_{\text{FS}}$ to delimit the scope of the upgrades. Figure 18 shows a modified version of the previous example, where y is explicitly marked as the only variable that should be upgraded in the first branch. As a consequence, x does not get upgraded and the program does not fail—the **output** operation is allowed. More generally, if there is enough static information to guide the use of $\textbf{withRefs}_{\text{FS}}$, we believe that $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ could match (or exceed) the permissiveness of permissive-upgrade.

8.7 Related work

The *label on the label* could be seen as a fixed label that dictates which principals can read or modify the policy (inner label) of a flow-sensitive entity. In a different setting, trust management frameworks have explored this idea [Bandhakavi et al., 2008], where role-based rules are labeled to restrict the view on policies—the mere presence of certain policies could become inappropriate conduits of information.

```
x, y := True
if h
  y := False
if x
  output (1)
```

Fig. 17: A secure program that permissive-upgrade accepts and $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ rejects.

```
x, y := False
withRefs_FS (y) {
  if h
    y := True
}
if x
  output (1)
```

Fig. 18: A secure program that permissive-upgrade and $\lambda^{\text{LIO}}_{\ell,\text{FS+AU}}$ with $\textbf{withRefs}_{\text{FS}}$ accept.

Several authors propose an *existence security label* to remove leaks due to the termination covert channel [Rafnsson et al., 2012, Rafnsson and Sabelfeld, 2013] or certain behaviors in dynamic nested data structures [Russo et al., 2009, Hedin and Sabelfeld, 2012]. While the existence security label and the label on the label are structurally isomorphic, they are used for different purposes and in different scenarios, e.g., inspecting labels is not allowed in [Russo et al., 2009, Hedin and Sabelfeld, 2012, Rafnsson et al., 2012, Rafnsson and Sabelfeld, 2013].

Hunt and Sands [Hunt and Sands, 2006] show the equivalence (modulo code transformation) between flow-sensitive and flow-insensitive type-systems. Russo and Sabelfeld [Russo and Sabelfeld, 2010] formally pin down the challenge of mutable labels when using purely dynamic monitors. They prove that monitors require static analysis in order to be more permissive than traditional flow-sensitive type-systems. Austin and Flanagan propose a *privatization* operation to boost the permissiveness of permissive-upgrade. This technique has been recently generalized to arbitrary lattices [Bichhawat et al., 2014]. Moreover, the privatization operation can only enforce non-interference when outputs are suppressed after branching on a marked flow-sensitive reference. Unfortunately, none of the mentioned work consider flow-sensitive in the presence of concurrency. In fact, the notion of permissive-upgrade does not easily generalize to the concurrent setting, as this would require tracking occurrences of branches across threads.

Recently, Hritcu et al. [Hritcu et al., 2013] propose a floating-label system called Breeze. Like LIO, Breeze allows changes in the current context label (i.e., *pc*) and only considered values with flow-insensitive labels. Given the design similarities with LIO [Stefan et al., 2011], we believe that our results could be easily adapted to Breeze.

Hedin et al. [Hedin et al., 2014] recently developed JSFlow, an IFC flow-sensitive monitor for JavaScript. The monitor uses the no-sensitive-upgrade label changing policy. To overcome some of the restrictions imposed by this discipline, the primitive **upgrade** is introduced to explicitly change labels. Our upgrade operation resembles that proposed by Hedin et al. Moreover, the extension to **unlabel** as described Section 8.3 can be seen as an automatic application of **upgrade** every time that the current label gets raised. Using testing, Birgisson et al. [Birgisson et al., 2012] automatically insert **upgrade** instructions to boost the permissiveness of no-sensitive-upgrade. We further extend this concept of (automatic) **upgrade**s to a concurrent setting.

The Operating System IFC community has also treated the mutable label problem in the presence of purely dynamic monitors. Specifically, IFC OSes such as Asbestos [Efstathopoulos et al., 2005], HiStar [Zeldovich et al., 2006], and Flume [Krohn et al., 2007] distinguish between subjects (processes), and objects (files, sockets, etc.) such that the security labels for objects are immutable, while subject labels change according to the

sensitivity of data being read. As in language-based IFC systems, changing the label of subjects and object can become a covert channel, if not handled appropriated. Hence, HiStar and Flume require that the label of a subject be done explicitly by the subject code. Asbestos, on the other hand, allowed (unsafe) changes to labels as the result of receiving messages under specific and safe conditions. Our work extends these concepts with a level of indirection to allow for changes in object labels.

Coarse-grained IFC enforcements, similar to the ones found in IFC OS work, have been applied to web browsers. e.g, BFlow [Yip et al., 2009] and COWL [Stefan et al., 2014] track the flow of information at the granularity of protection zones and context, respectively. Both can be understood as tracking IFC at the iframe-level granularity. As in LIO, an iframe's label, i.e., a subject's label, must be explicitly updated. While our techniques can be applied to COWL, BFlow does not have fine-grained labeled objects; hence the flow-sensitivity result is only applicable at the protection zone level. By taking a more fine grained approach, the DOM-tree could be thought of as being composed of flow-sensitive objects, whose security labels change according to the dynamic behavior of the web page, as done in [Russo et al., 2009].

Hoare-like logics for IFC are often flow-sensitive [e.g. Amtoft et al., 2006, Nanevski et al., 2011]. Different from dynamic approaches, these logics have the ability to observe all the execution paths and safely approximate label changes. As a result, no leaks due to label changes are present in provably secure programs. Le Guernic et al. [Le Guernic et al., 2006, Le Guernic, 2007] combine dynamic and static checks in a flow-sensitive execution monitor. For a flow-sensitive type-system, Foster et al. [Foster et al., 2002] propose a **restrict** primitive that limits the use of variables' aliases in a block of code. Our $\mathbf{withRefs}_{\mathrm{FS}}$ is similar to **restrict** in being used to increase the permissiveness of the analysis.

8.8 Conclusions

We presented an extension of LIO with flow-sensitive references. As in previous flow-sensitive work, our approach allows secure label changes using **upgrade** and **downgrade** operations, as a way to boost the permissiveness of the IFC system, i.e., **upgrade** can be used to allow for the encoding of programs that would otherwise be rejected by the IFC monitor. Since manually inserting **upgrade** operations can be cumbersome, we extend the calculus to automatically insert upgrades whenever the current label is raised, while still giving programmers fine-grained control over which references untrusted code can upgrade. Importantly, our approach extends to a concurrent setting. To the best of our knowledge, this is the first work to address the problem of flow-sensitive label changes for a concurrent, dynamic IFC language. A further insight of this work was to show that, by lever-

aging nested labeled objects, both the sequential and concurrent calculi with flow-sensitive references can be encoded using only flow-insensitive constructs. As a consequence, our soundness proof can be reduced to an invocation of previous results for LIO.

Acknowledgments

We thank our colleagues in the ProSec group at Chalmers, Stefan Heule, David Mazières, and Edward Z. Yang for the useful discussions. We thank the anonymous reviewers for constructive feedback on an earlier version of this work. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, the Swedish research agency VR, and a grant from Mozilla. Deian Stefan was supported by the DoD through the NDSEG Fellowship Program.

BIBLIOGRAPHY

Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06. ACM, 2006.

T. H. Austin and C. Flanagan. Efficient Purely-Dynamic Information Flow Analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10. ACM, 2010.

S. Bandhakavi, W. Winsborough, and M. Winslett. A trust management approach for flexible policy management in security-typed languages. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, June 2008.

Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, PLAS'14. ACM, 2014.

Arnar Birgisson, Daniel Hedin, and Andrei Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. In *Proc. European Symp. on Research in Computer Security*, 2012.

P. Buiras, D. Stefan, and A. Russo. On Dynamic Flow-sensitive Floating-Label Systems. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE, July 2014.

Winnie Cheng, Dan RK Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information

flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12. ACM, 2012.

Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proc. of the twentieth ACM symp. on Operating systems principles*, SOSP '05. ACM, 2005.

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10. USENIX Association, 2010.

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, 103(2):235–271, 1992.

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02. ACM, 2002.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, October 2012.

J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

Daniel Hedin and Andrei Sabelfeld. Information-Flow Security for a Core of JavaScript. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, 2012.

Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. ACM Symposium on Applied Computing (SAC)*. ACM, March 2014.

Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All Your IFCException Are Belong to Us. *2012 IEEE Symposium on Security and Privacy*, 0, 2013.

Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, POPL '06, pages 79–90. ACM, 2006.

Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Runtime enforcement of information-flow properties on Android (extended abstract). In *Computer Security—ESORICS 2013: 18th European Symposium on Research in Computer Security*. Springer, September 2013.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st Symp. on Operating Systems Principles*, October 2007.

Gurvan Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, CSF '07. IEEE Computer Society, 2007.

Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06. Springer-Verlag, 2006.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.

Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11. IEEE Computer Society, 2011.

Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 33–48. IEEE, 2013.

Willard Rafnsson, Daniel Hedin, and Andrei Sabelfeld. Securing Interactive Programs. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, 2012.

Alejandro Russo and Andrei Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proc. of the 2010 23rd IEEE Computer Security Foundations Symp.*, CSF '10, pages 186–199. IEEE Computer Society, 2010.

Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information flow in dynamic tree structures. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09. Springer-Verlag, 2009.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

V. Simonet. The Flow Caml System. Software release at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible Dynamic Information Flow Control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the 17th ACM SIGPLAN International Conference on Functional Programming*, Sep. 2012a.

Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications*, NordSec'11. Springer-Verlag, 2012b.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in the presence of exceptions. `http://www.cse.chalmers.se/~russo/jfp15.pdf`, 2012c.

Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, October 2014.

Philip Wadler. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences*. Springer Verlag, August 1992.

Edward Yang, Deian Stefan, John Mitchell, David Mazières, Petr Marchenko, and Brad Karp. Toward principled browser security. In *The 14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. USENIX, 2013.

Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. Privacy-preserving browser-side scripting with bflow. In *Proc. of the 4th ACM European Conference on Computer Systems*, EuroSys '09. ACM, 2009.

Stephan Zdancewic. PhD thesis: Programming languages for information security. Technical report, Cornell University, 2002.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

$$\textsc{app}\quad \frac{}{E\,[(\lambda x.t_1)\ t_2] \longrightarrow E\,[\{\,t_2\ /\ x\,\}\ t_1]} \qquad \textsc{fix}\quad \frac{}{E\,[\mathbf{fix}\ (\lambda x.t)] \longrightarrow E\,[\{\,\mathbf{fix}\ (\lambda x.t)\ /\ x\,\}\ t]}$$

$$\textsc{ifTrue}\quad \frac{}{E\,[\mathbf{if}\ \mathit{True}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3] \longrightarrow E\,[t_2]}$$

$$\textsc{ifFalse}\quad \frac{}{E\,[\mathbf{if}\ \mathit{False}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3] \longrightarrow E\,[t_3]} \qquad \textsc{labelOp}\quad \frac{v = [\![l_1\ \otimes\ l_2]\!]_\ell}{E\,[l_1\ \otimes\ l_2] \longrightarrow E\,[v]}$$

$$\textsc{return}\quad \frac{}{\langle \Sigma | \mathbf{E}\,[\mathbf{return}\ t]\rangle \longrightarrow \langle \Sigma | \mathbf{E}\,[\mathit{LIO}^{\mathsf{TCB}}\ t]\rangle}$$

$$\textsc{bind}\quad \frac{}{\langle \Sigma | \mathbf{E}\,[(\mathit{LIO}^{\mathsf{TCB}}\ t_1) \gg\!= t_2]\rangle \longrightarrow \langle \Sigma | \mathbf{E}\,[t_2\ t_1]\rangle}$$

Fig. 19: Reduction rules for standard $\lambda^{\mathsf{LIO}}_{\ell}$ terms.

A Semantics for the base calculus

The reduction rules for pure and monadic terms are given in Figure 19. We define substitution $\{\,t_2\ /\ x\,\}\ t_1$ in the usual way: homomorphic on all operators and renaming bound names to avoid captures. Our label operations $\sqcup$, $\sqcap$, and $\sqsubseteq$ rely on the label-specific implementation of these lattice operators, as used in the premise of rule (LABELOP); we use the meta-level partial function $[\![\cdot]\!]_\ell$, which maps terms to values, to precisely capture this implementation detail.

The reduction rules for pure terms are standard. For instance, in rule (IFTRUE), when the branch has a true condition, i.e., $E\,[\mathbf{if}\ \mathit{True}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3]$, it reduces to the then branch ($E\,[t_2]$). The rest are self-explanatory and we do not discuss them any further.

Since all the IFC checks are performed by individual LIO terms, the definition for **return** and ($\gg\!=$) are trivial. The former simply reduces to a monadic value by wrapping the term with the $\mathit{LIO}^{\mathsf{TCB}}$ constructor, while the latter evaluates the left-hand term and supplies the result to the right-hand term, as usual.

B Attack on naive flow-sensitive references

As in the attack of Figure 7, the *leakRef* of Figure 20 can be used to leak the value stored in a **H** reference *href*, while keeping the current label **L**, without using **labelOf**. Internally, the value is leaked into public reference *lref* by leveraging the fact that, based on a secret value, the label of a public

```
leakRef :: Ref_FS Bool → LIO Bool
leakRef href = do
  lref ← newRef L True
  tmp ← newRef L False
  toLabeled H $ do h ← readRef href
                   when h $ writeRef tmp True
  toLabeled H $ do t ← readRef tmp
                   when (¬ t) $ writeRef lref False
  readRef lref
```

Fig. 20: An attack in LIO with naive flow-sensitive reference extension without **labelOf**.

reference (*tmp*) can be changed (or not). In the first **toLabeled** block, if $h \equiv \mathit{True}$, then the label of *tmp* is raised to **H** and its value is set to *True*. In the second **toLabeled** block, we read *tmp*, which may raise the current label to **H** if the secret is *True* (and thus *tmp* was upgraded). Indeed, if the secret is *True* (and thus $t \equiv \mathit{True}$) we leave the public reference intact: *True*. However, if the secret is *False*, the *tmp* reference is not modified in the first **toLabeled** block and thus when reading it in the second **toLabeled** block, the current label remains **L**, and since $t \equiv \mathit{False}$, we write *False* into the public reference. In both cases the value stored in *lref* corresponds to that of *href*, yet leaving the current label and the label of *lref* intact (**L**).

C Embedding Theorem

In this section we prove that the embedding from $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$ into $\lambda^{\mathsf{LIO}}_{\ell}$ preserves semantics. We define the notation $\mu \gg n$ as a shorthand for $\mu \gg\!= \lambda x \to n$, which is typically used when n ignores the result of μ.

We will use the following lemma for single $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$ steps:

Lemma 2 (Single-step embedding). *Let t be a well-typed term in $\lambda^{\mathsf{LIO}}_{\ell,\mathrm{FS}}$. Then if $\langle \Sigma | t \rangle \longrightarrow \langle \Sigma' | t' \rangle$, then there is a configuration Y such that $\langle [\![\Sigma]\!]_{\mathrm{FI}} | [\![t]\!]^{\Sigma}_{\mathrm{FI}} \rangle \longrightarrow^* Y$ and $\langle [\![\Sigma']\!]_{\mathrm{FI}} | [\![t']\!]^{\Sigma}_{\mathrm{FI}} \rangle \longrightarrow^* Y$, i.e. $\langle [\![\Sigma]\!]_{\mathrm{FI}} | [\![t]\!]^{\Sigma}_{\mathrm{FI}} \rangle$ and $\langle [\![\Sigma']\!]_{\mathrm{FI}} | [\![t']\!]^{\Sigma}_{\mathrm{FI}} \rangle$ are β-equivalent.*

Proof. Case analysis on the next redex in t. Most cases show a stronger version of the lemma, i.e. that $\langle \Sigma | t \rangle \longrightarrow \langle \Sigma' | t' \rangle$ implies $\langle [\![\Sigma]\!]_{\mathrm{FI}} | [\![t]\!]^{\Sigma}_{\mathrm{FI}} \rangle \longrightarrow^* \langle [\![\Sigma']\!]_{\mathrm{FI}} | [\![t']\!]^{\Sigma}_{\mathrm{FI}} \rangle$.

Case E $[\mathbf{newRef}_{\mathrm{FS}}\ l\ t]$.

We have $\langle \Sigma | \mathbf{E}\ [\mathbf{newRef}_{\mathrm{FS}}\ l\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ (\mathit{Ref}^{\mathsf{TCB}}_{\mathrm{FS}}\ a)] \rangle$, where $\Sigma' = \Sigma\ [\mu_{\mathrm{FS}} \mapsto \Sigma.\mu_{\mathrm{FS}}\ [a \mapsto \mathit{Lb}^{\mathsf{TCB}}\ \Sigma.l_{\mathrm{cur}}\ (\mathit{Lb}^{\mathsf{TCB}}\ l\ t)]]$, and we know that $\Sigma.l_{\mathrm{cur}} \sqsubseteq l$.

Let $\Sigma_1 = [\![\Sigma]\!]_{\mathrm{FI}}$. We argue

$$
\begin{array}{l}
\quad \langle \Sigma_1 | [\![\mathbf{E}\ [\mathbf{newRef}_{\mathrm{FS}}\ l\ t]]\!]^{\Sigma}_{\mathrm{FI}} \rangle \\
\longrightarrow \langle \Sigma_1' | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{do}\ l_{\mathsf{cur}} \leftarrow \mathbf{getLabel}; o \leftarrow \mathbf{newRef}_{\mathrm{FI}}\ l_{\mathsf{cur}}\ i; \ldots] \rangle \\
\quad (\Sigma_1' = \Sigma_1\ [\mu_{\mathrm{FI}} \mapsto \Sigma_1.\mu_{\mathrm{FI}}\ [i \mapsto Lb^{\mathsf{TCB}}\ \Sigma_1.lbl\ t]]) \\
\longrightarrow \langle \Sigma_1' | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{do}\ o \leftarrow \mathbf{newRef}_{\mathrm{FI}}\ \Sigma_1.lbl\ i; \mathbf{return}\ (wrap\ o)] \rangle \\
\quad (\Sigma_1' = \Sigma_1\ [\mu_{\mathrm{FI}} \mapsto \Sigma_1.\mu_{\mathrm{FI}}\ [i \mapsto Lb^{\mathsf{TCB}}\ \Sigma_1.lbl\ t]]) \\
\longrightarrow \langle \Sigma_1'' | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{return}\ (wrap\ (Ref^{\mathsf{TCB}}_{\mathrm{FI}}\ \Sigma_1'.lbl\ o))] \rangle \\
\quad (\Sigma_1'' = \Sigma_1\ [\mu_{\mathrm{FI}} \mapsto \Sigma_1.\mu_{\mathrm{FI}}\ [i \mapsto Lb^{\mathsf{TCB}}\ \Sigma_1.lbl\ t; \\
\qquad\qquad\qquad\qquad o \mapsto Lb^{\mathsf{TCB}}\ \Sigma_1'.lbl\ (Ref^{\mathsf{TCB}}_{\mathrm{FI}}\ \Sigma_1.lbl\ i)]])
\end{array}
$$

We now have to check that $[\![\mathbf{return}\ (Ref^{\mathsf{TCB}}_{\mathrm{FS}}\ a)]\!]^{\Sigma'}_{\mathrm{FI}} = \mathbf{return}\ (wrap\ (Ref^{\mathsf{TCB}}_{\mathrm{FI}}\ \Sigma_1'.lbl\ o))$ and $[\![\Sigma']\!]_{\mathrm{FI}} = \Sigma_1''$, which follow directly from the definition of $[\![\cdot]\!]_{\mathrm{FI}}$ for references and states.

Case $\mathbf{E}\ [\mathbf{readRef}_{\mathrm{FS}}\ (Ref^{\mathsf{TCB}}_{\mathrm{FS}}\ a)]$.

We have $\langle \Sigma | \mathbf{E}\ [\mathbf{readRef}_{\mathrm{FS}}\ (Ref^{\mathsf{TCB}}_{\mathrm{FS}}\ a)] \rangle \longrightarrow \langle \Sigma | \mathbf{E}\ [\mathbf{unlabel}\ (Lb^{\mathsf{TCB}}\ (l \sqcup l')\ t)] \rangle$, where $\Sigma.\mu_{\mathrm{FS}}\ (a) = Lb^{\mathsf{TCB}}\ l\ (Lb^{\mathsf{TCB}}\ l'\ t)$.

Let $\Sigma_1 = [\![\Sigma]\!]_{\mathrm{FI}}$. We argue

$$
\begin{array}{l}
\quad \langle \Sigma_1 | [\![\mathbf{E}\ [\mathbf{readRef}_{\mathrm{FS}}\ (Ref^{\mathsf{TCB}}_{\mathrm{FS}}\ a)]]\!]^{\Sigma}_{\mathrm{FI}} \rangle \\
\longrightarrow \langle \Sigma_1 | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{readRef}_{\mathrm{FI}}\ ([\![Ref^{\mathsf{TCB}}_{\mathrm{FS}}\ a]\!]^{\cdot}_{\mathrm{FI}}) \gg\!= \mathbf{readRef}_{\mathrm{FI}}] \rangle \\
\longrightarrow \langle \Sigma_1 | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{unlabel}\ (\Sigma_1.\mu_{\mathrm{FI}}\ (a)) \gg\!= \mathbf{readRef}_{\mathrm{FI}}] \rangle \\
\longrightarrow \langle \Sigma_1' | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{return}\ (Ref^{\mathsf{TCB}}_{\mathrm{FI}}\ l'\ i) \gg\!= \mathbf{readRef}_{\mathrm{FI}}] \rangle \\
\quad (\Sigma_1' = \Sigma_1\ [l_{\mathsf{cur}} \mapsto \Sigma_1.l_{\mathsf{cur}} \sqcup l]) \\
\longrightarrow \langle \Sigma_1' | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{readRef}_{\mathrm{FI}}\ (Ref^{\mathsf{TCB}}_{\mathrm{FI}}\ l'\ i)] \rangle \\
\longrightarrow \langle \Sigma_1' | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{unlabel}\ (\Sigma_1'.\mu_{\mathrm{FI}}\ (i))] \rangle \\
\longrightarrow \langle \Sigma_1'' | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{return}\ t] \rangle \\
\quad (\Sigma_1'' = \Sigma_1'\ [l_{\mathsf{cur}} \mapsto \Sigma_1'.l_{\mathsf{cur}} \sqcup l'])
\end{array}
$$

Now if we consider $\langle [\![\Sigma]\!]_{\mathrm{FI}} | [\![\mathbf{E}\ [\mathbf{unlabel}\ (Lb^{\mathsf{TCB}}\ (l \sqcup l')\ t)]]\!]^{\cdot}_{\mathrm{FI}} \rangle$, we have

$$
\begin{array}{l}
\quad \langle [\![\Sigma]\!]_{\mathrm{FI}} | [\![\mathbf{E}\ [\mathbf{unlabel}\ (Lb^{\mathsf{TCB}}\ (l \sqcup l')\ t)]]\!]^{\cdot}_{\mathrm{FI}} \rangle \\
\longrightarrow \langle \Sigma_2 | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{return}\ t] \\
\quad (\Sigma_2 = \Sigma_2\ [l_{\mathsf{cur}} \mapsto ([\![\Sigma]\!]_{\mathrm{FI}}).l_{\mathsf{cur}} \sqcup l \sqcup l']) \rangle
\end{array}
$$

Note that $\Sigma_1''.l_{\mathsf{cur}} = ([\![\Sigma]\!]_{\mathrm{FI}}).l_{\mathsf{cur}} \sqcup l \sqcup l' = \Sigma_2.l_{\mathsf{cur}}$.

Case $\mathbf{E}\ [\mathbf{writeRef}_{\mathrm{FS}}\ (Ref^{\mathsf{TCB}}_{\mathrm{FS}}\ a)\ t]$.

We have $\langle \Sigma | \mathbf{E}\ [\mathbf{writeRef}_{\mathrm{FS}}\ (Ref^{\mathsf{TCB}}_{\mathrm{FS}}\ a)\ t] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ ()] \rangle$, where $\Sigma.\mu_{\mathrm{FS}}\ (a) = Lb^{\mathsf{TCB}}\ l\ (Lb^{\mathsf{TCB}}\ l'\ v)$, $\Sigma' = \Sigma\ [\mu_{\mathrm{FS}} \mapsto \Sigma.\mu_{\mathrm{FS}}\ [a \mapsto Lb^{\mathsf{TCB}}\ l\ (Lb^{\mathsf{TCB}}\ l'\ v)]]$, and we know that $\Sigma.l_{\mathsf{cur}} \sqsubseteq l \sqcup l'$.

Let $\Sigma_1 = [\![\Sigma]\!]_{\mathrm{FI}}$. Then there exists a function μ such that:

$$
\begin{array}{l}
\quad \langle \Sigma_1 | [\![\mathbf{E}\ [\mathbf{upgrade}_{\mathrm{FS}}\ (Ref^{\mathsf{TCB}}_{\mathrm{FS}}\ a)\ l']]\!]^{\Sigma}_{\mathrm{FI}} \rangle \\
\longrightarrow \langle \Sigma_1 | ([\![\mathbf{E}]\!]^{\Sigma}_{\mathrm{FI}})\ [\mathbf{toLabeled}\ (\Sigma_1.l_{\mathsf{cur}} \sqcup l)\ (\mu\ \Sigma_1.l_{\mathsf{cur}}) \gg \mathbf{return}\ ()] \rangle
\end{array}
$$

We now step through the evaluation of $\langle \Sigma_1 | \mu\ \Sigma_1.l_{\mathsf{cur}} \rangle$, as follows:

$$
\begin{array}{l}
\quad \langle \Sigma_1 | \mu\ \Sigma_1.l_{\text{cur}} \rangle \\
\longrightarrow \langle \Sigma_1 |\ \mathbf{do}\ i \leftarrow \mathbf{readRef}_{\text{FI}}\ (\llbracket Ref_{\text{FS}}^{\text{TCB}}\ a \rrbracket_{\text{FI}}^{\cdot});\ \mathbf{writeRef}_{\text{FI}}\ i\ t \rangle \\
\longrightarrow \langle \Sigma_1' | \mathbf{writeRef}_{\text{FI}}\ i\ t \rangle \\
\quad (\Sigma_1' = \Sigma_1\ [l_{\text{cur}} \mapsto \Sigma_1.l_{\text{cur}} \sqcup l]) \\
\longrightarrow \langle \Sigma_1'' | \mathbf{return}\ () \rangle \\
\quad (\Sigma_1'' = \Sigma_1'\ [\mu_{\text{FI}} \mapsto \Sigma_1'.\mu_{\text{FI}}\ [i \mapsto Lb^{\text{TCB}}\ l'\ t]])
\end{array}
$$

Finally, this allows us to conclude (from the rule for **toLabeled** and the definition of $(\gg)$), that

$$
\begin{array}{l}
\langle \Sigma_1 | (\llbracket \mathbf{E} \rrbracket_{\text{FI}}^{\Sigma})\ [\mathbf{toLabeled}\ (\Sigma_1.l_{\text{cur}} \sqcup l)\ (\mu\ \Sigma_1.l_{\text{cur}}) \gg \mathbf{return}\ ()] \rangle \\
\longrightarrow^* \langle \Sigma_2 | (\llbracket \mathbf{E} \rrbracket_{\text{FI}}^{\Sigma})\ [\mathbf{return}\ ()] \rangle
\end{array}
$$

where $\Sigma_2 = (\Sigma_1.l_{\text{cur}}, \Sigma_1''.\mu_{\text{FI}})$. Now we can check that $\llbracket \Sigma' \rrbracket_{\text{FI}} = \Sigma_2$ from the definition of $\llbracket \cdot \rrbracket_{\text{FI}}$ for states.

Case E $[\mathbf{upgrade}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)\ l']$.

We have $\langle \Sigma | \mathbf{E}\ [\mathbf{upgrade}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)\ l'] \rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ ()] \rangle$, where $\Sigma.\mu_{\text{FS}}\ (a) = Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ l''\ v)$, $\Sigma' = \Sigma\ [\mu_{\text{FS}} \mapsto \Sigma.\mu_{\text{FS}}\ [a \mapsto Lb^{\text{TCB}}\ l\ (Lb^{\text{TCB}}\ (l \sqcup l'' \sqcup l')\ v)]]$, and we know that $\Sigma.l_{\text{cur}} \sqsubseteq l$.

Let $\Sigma_1 = \llbracket \Sigma \rrbracket_{\text{FI}}$. Then there exists a function μ such that:

$$
\begin{array}{l}
\quad \langle \Sigma_1 | \llbracket \mathbf{E}\ [\mathbf{upgrade}_{\text{FS}}\ (Ref_{\text{FS}}^{\text{TCB}}\ a)\ l'] \rrbracket_{\text{FI}}^{\Sigma} \rangle \\
\longrightarrow \langle \Sigma_1 | (\llbracket \mathbf{E} \rrbracket_{\text{FI}}^{\Sigma})\ [\mathbf{toLabeled}\ (\Sigma_1.l_{\text{cur}} \sqcup l)\ (\mu\ \Sigma_1.l_{\text{cur}}) \gg \mathbf{return}\ ()] \rangle
\end{array}
$$

We now step through the evaluation of $\langle \Sigma_1 | \mu\ \Sigma_1.l_{\text{cur}} \rangle$, as follows:

$$
\begin{array}{l}
\quad \langle \Sigma_1 | \mu\ \Sigma_1.l_{\text{cur}} \rangle \\
\longrightarrow \langle \Sigma_1 |\ \mathbf{do}\ i \leftarrow \mathbf{readRef}_{\llbracket Ref_{\text{FS}}^{\text{TCB}}\ a \rrbracket_{\text{FI}}^{\cdot}};\ lc \leftarrow \mathbf{getLabel};\ldots \rangle \\
\longrightarrow \langle \Sigma_1' |\ \mathbf{do}\ lc \leftarrow \mathbf{getLabel};\ n \leftarrow \mathbf{newRef}_{\text{FI}}\ (lc \sqcup (l' \sqcup l))\ \bot;\ldots \rangle \\
\quad (\Sigma_1' = \Sigma_1\ [l_{\text{cur}} \mapsto \Sigma_1.l_{\text{cur}} \sqcup l]) \\
\longrightarrow \langle \Sigma_1' |\ \mathbf{do}\ n \leftarrow \mathbf{newRef}_{\text{FI}}\ (lc \sqcup (l' \sqcup l))\ \bot;\ \mathbf{copyRef}\ i\ n;\ldots \rangle \\
\longrightarrow \langle \Sigma_1'' |\ \mathbf{do}\ \mathbf{copyRef}\ i\ n;\ \mathbf{writeRef}_{\text{FI}}\ (\llbracket Ref_{\text{FS}}^{\text{TCB}}\ a \rrbracket_{\text{FI}}^{\cdot})\ n \rangle \\
\quad (\Sigma_1'' = \Sigma_1'\ [\mu_{\text{FI}} \mapsto \Sigma_1'.\mu_{\text{FI}}\ [n \mapsto Lb^{\text{TCB}}\ (lc \sqcup (l' \sqcup l))\ \bot]]) \\
\longrightarrow \langle \Sigma_1''' | \mathbf{writeRef}_{\text{FI}}\ (\llbracket Ref_{\text{FS}}^{\text{TCB}}\ a \rrbracket_{\text{FI}}^{\cdot})\ n \rangle \\
\quad (\Sigma_1''' = \Sigma_1''\ [\mu_{\text{FI}} \mapsto \Sigma_1''.\mu_{\text{FI}}\ [n \mapsto Lb^{\text{TCB}}\ (lc \sqcup (l' \sqcup l))\ v]]) \\
\longrightarrow \langle \Sigma_1'''' | \mathbf{return}\ () \rangle \\
\quad (\Sigma_1'''' = \Sigma_1'''\ [\mu_{\text{FI}} \mapsto \Sigma_1'''.\mu_{\text{FI}}\ [a \mapsto Lb^{\text{TCB}}\ l\ (Ref_{\text{FI}}^{\text{TCB}}\ (lc \sqcup l' \sqcup l)\ n)]])
\end{array}
$$

Finally, this allows us to conclude (from the rule for **toLabeled** and the definition of $(\gg)$), that

$$
\begin{array}{l}
\langle \Sigma_1 | (\llbracket \mathbf{E} \rrbracket_{\text{FI}}^{\Sigma})\ [\mathbf{toLabeled}\ l'\ (\mu\ \Sigma_1.l_{\text{cur}}) \gg \mathbf{return}\ ()] \rangle \\
\longrightarrow^* \langle \Sigma_2 | (\llbracket \mathbf{E} \rrbracket_{\text{FI}}^{\Sigma})\ [\mathbf{return}\ ()] \rangle
\end{array}
$$

where $\Sigma_2 = (\Sigma_1.l_{\text{cur}}, \Sigma_1''''.\mu_{\text{FI}})$. Now we can check that $\llbracket \Sigma' \rrbracket_{\text{FI}} = \Sigma_2$ from the definition of $\llbracket \cdot \rrbracket_{\text{FI}}$ for states.

Case E $[\mathbf{downgrade}_{\mathrm{FS}}\ (Ref_{\mathrm{FS}}^{\mathsf{TCB}}\ a)\ l']$.

We have $\langle \Sigma | \mathbf{E}\ [\mathbf{downgrade}_{\mathrm{FS}}\ (Ref_{\mathrm{FS}}^{\mathsf{TCB}}\ a)\ l']\rangle \longrightarrow \langle \Sigma' | \mathbf{E}\ [\mathbf{return}\ ()]\rangle$, where $\Sigma.\mu_{\mathrm{FS}}\ (a) = Lb^{\mathsf{TCB}}\ l\ (Lb^{\mathsf{TCB}}\ l''\ v)$, $\Sigma' = \Sigma\ [\mu_{\mathrm{FS}} \mapsto \Sigma.\mu_{\mathrm{FS}}\ [a \mapsto Lb^{\mathsf{TCB}}\ l\ (Lb^{\mathsf{TCB}}\ (l \sqcup l'' \sqcap l')\ \bot)]]$, and we know that $\Sigma.l_{\mathrm{cur}} \sqsubseteq l$.

Let $\Sigma_1 = [\![\Sigma]\!]_{\mathrm{FI}}$. Then there exists a function μ such that:

$$\begin{array}{l}\quad \langle \Sigma_1 | [\![\mathbf{E}\ [\mathbf{downgrade}_{\mathrm{FS}}\ (Ref_{\mathrm{FS}}^{\mathsf{TCB}}\ a)\ l']]\!]_{\mathrm{FI}}^{\Sigma}\rangle \\ \longrightarrow \langle \Sigma_1 | ([\![\mathbf{E}]\!]_{\mathrm{FI}}^{\Sigma})\ [\mathbf{toLabeled}\ (\Sigma_1.l_{\mathrm{cur}} \sqcup l)\ (\mu\ \Sigma_1.l_{\mathrm{cur}}) \gg \mathbf{return}\ ()]\rangle\end{array}$$

We now step through the evaluation of $\langle \Sigma_1 | \mu\ \Sigma_1.l_{\mathrm{cur}}\rangle$, as follows:

$$\begin{array}{l}\quad \langle \Sigma_1 | \mu\ \Sigma_1.l_{\mathrm{cur}}\rangle \\ \longrightarrow \langle \Sigma_1 |\ \mathbf{do}\ i \leftarrow \mathbf{readRef}_{[\![Ref_{\mathrm{FS}}^{\mathsf{TCB}}\ a]\!]_{\mathrm{FI}}^{\cdot}};\ lc \leftarrow \mathit{getlabel}; \ldots\rangle \\ \longrightarrow \langle \Sigma_1' |\ \mathbf{do}\ lc \leftarrow \mathbf{getLabel};\ n \leftarrow \mathbf{newRef}_{\mathrm{FI}}\ (lc \sqcup (l' \sqcap l))\ \bot; \ldots\rangle \\ \quad (\Sigma_1' = \Sigma_1\ [l_{\mathrm{cur}} \mapsto \Sigma_1.l_{\mathrm{cur}} \sqcup l]) \\ \longrightarrow \langle \Sigma_1' |\ \mathbf{do}\ n \leftarrow \mathbf{newRef}_{\mathrm{FI}}\ (lc \sqcup (l' \sqcap l))\ \bot; \mathbf{writeRef}_{\mathrm{FI}}\ ([\![Ref_{\mathrm{FS}}^{\mathsf{TCB}}\ a]\!]_{\mathrm{FI}}^{\cdot})\ n\rangle \\ \longrightarrow \langle \Sigma_1'' | \mathbf{writeRef}_{\mathrm{FI}}\ ([\![Ref_{\mathrm{FS}}^{\mathsf{TCB}}\ a]\!]_{\mathrm{FI}}^{\cdot})\ n\rangle \\ \quad (\Sigma_1'' = \Sigma_1'\ [\mu_{\mathrm{FI}} \mapsto \Sigma_1'.\mu_{\mathrm{FI}}\ [n \mapsto Lb^{\mathsf{TCB}}\ (lc \sqcup (l' \sqcap l))\ \bot]]) \\ \longrightarrow \langle \Sigma_1''' | \mathbf{return}\ ()\rangle\end{array}$$

Finally, this allows us to conclude (from the rule for **toLabeled** and the definition of $(\gg)$), that

$$\begin{array}{l}\langle \Sigma_1 | ([\![\mathbf{E}]\!]_{\mathrm{FI}}^{\Sigma})\ [\mathbf{toLabeled}\ l'\ (\mu\ \Sigma_1.l_{\mathrm{cur}}) \gg \mathbf{return}\ ()]\rangle \\ \longrightarrow^* \langle \Sigma_2 | ([\![\mathbf{E}]\!]_{\mathrm{FI}}^{\Sigma})\ [\mathbf{return}\ ()]\rangle\end{array}$$

where $\Sigma_2 = (\Sigma_1.l_{\mathrm{cur}}, \Sigma_1'''.\mu_{\mathrm{FI}})$. Now we can check that $[\![\Sigma']\!]_{\mathrm{FI}} = \Sigma_2$ from the definition of $[\![\cdot]\!]_{\mathrm{FI}}$ for states.

Now we can state the main theorem of this section.

Theorem. [Embedding $\lambda_{\ell,\mathrm{FS}}^{\mathsf{UIO}}$ in $\lambda_{\ell}^{\mathsf{UIO}}$] Let t be a well-typed term in $\lambda_{\ell,\mathrm{FS}}^{\mathsf{UIO}}$. Then if $\langle \Sigma | t\rangle \longrightarrow^* \langle \Sigma' | v\rangle$, we have $\langle [\![\Sigma]\!]_{\mathrm{FI}} | [\![t]\!]_{\mathrm{FI}}^{\Sigma}\rangle \longrightarrow^* \langle [\![\Sigma']\!]_{\mathrm{FI}} | [\![v]\!]_{\mathrm{FI}}^{\Sigma}\rangle$, and if $\langle \Sigma | t\rangle \longrightarrow^* \langle \Sigma' | \Uparrow\rangle$, then $\langle [\![\Sigma]\!]_{\mathrm{FI}} | [\![t]\!]_{\mathrm{FI}}^{\Sigma}\rangle \longrightarrow^* \langle [\![\Sigma']\!]_{\mathrm{FI}} | \Uparrow\rangle$.

Proof. By induction on the number of steps in $\langle \Sigma | t\rangle \longrightarrow^* \langle \Sigma' | v\rangle$, using Lemma 2 and uniqueness of normal forms in $\lambda_{\ell}^{\mathsf{UIO}}$.

CHAPTER
NINE

HLIO: MIXING STATIC AND DYNAMIC TYPING FOR INFORMATION-FLOW CONTROL IN HASKELL

Pablo Buiras, Dimitrios Vytiniotis, Alejandro Russo

Abstract. Information-Flow Control (IFC) is a well-established approach for allowing untrusted code to manipulate sensitive data without disclosing it. IFC is typically enforced via type systems and static analyses or via dynamic execution monitors. The LIO Haskell library, originating in operating systems research, implements a purely dynamic monitor of the sensitivity level of a computation, particularly suitable when data sensitivity levels are only known at runtime. In this paper, we show how to give programmers the flexibility of deferring IFC checks to runtime (as in LIO), while also providing static guarantees—and the absence of runtime checks—for parts of their programs that can be statically verified (unlike LIO). We present the design and implementation of our approach, HLIO (Hybrid LIO), as an embedding in Haskell that uses a novel technique for deferring IFC checks based on singleton types and constraint polymorphism. We formalize HLIO, prove non-interference, and show how interesting IFC examples can be programmed. Although our motivation is IFC, our technique for deferring constraints goes well beyond and offers a methodology for programmer-controlled hybrid type checking in Haskell.

9.1 Introduction

Preserving confidentiality of data has become of extreme importance, particularly in complex systems where *untrusted* components require access to sensitive information (e.g. text messages, contact lists, pictures, etc.) in order to provide their functionality. Information-Flow Control (IFC) is a well-established approach for allowing untrusted code to manipulate sensitive data without *disclosing* it [Sabelfeld and Myers, 2003]. IFC essentially scrutinizes source code to track how data of different sensitivity levels flows within a program, where security alarms are raised when confidentiality might be at stake. IFC research has produced three mature compilers for secure programs: *Jif* [Myers and Liskov, 2000] (based on Java), *FlowCaml* [Simonet, 2003] (based on Caml and not developed any more), and *Paragon* [Broberg et al., 2013] (based on Java). Alternatively, IFC can be provided via simple libraries in Haskell where concepts like arrows and monads are repurposed to protect confidentiality [Li and Zdancewic, 2006, Russo et al., 2008].

There exists a broad spectrum of enforcement mechanisms for IFC, ranging from fully dynamic ones, e.g., in the form of execution monitors [Austin and Flanagan, 2009, Askarov and Sabelfeld, 2009], to static ones, e.g., in the form of type systems [Volpano et al., 1996]. Although dynamic and static techniques provide similar security guarantees [Sabelfeld and Russo, 2009], there are many arguments for choosing dynamic over static approaches and vice versa. Several of these arguments have their roots in the long-term dispute between dynamic and static analyses, e.g., overhead vs. performance, enforcing properties for a program once and for all vs. monitoring properties in every run of a program, etc.

From the security point of view, specifically, there are good reasons to prefer dynamic over static approaches. Code statically verified to preserve confidentiality clearly adheres to data sensitivity levels and policies valid *at compile* time. However, data sensitivity levels may be entirely dynamic (e.g. we may read data from a trusted or a non-trusted domain at runtime) and even policies may change at runtime (e.g. principals (users) can change the set of principals they share data with by—for instance—altering their list of friends). In situations like this, the statically verified code has to be restructured to perform runtime checks in ways that the static analysis or the type system can understand and exploit to verify the program (we will see an example of that in Section 9.3). Alternatively, programs have to be written in a way that can statically deal with *all possible* sensitivity levels or policies that they could potentially encounter at runtime; this in turn may limit the set of useful side-effects programs can perform.

The LIO library [Stefan et al., 2011b] for Haskell offers a way of tackling this problem by providing a monad that dynamically enforces IFC. Borrowing ideas from operating systems research [VanDeBogart et al., 2007, Zeldovich et al., 2006], the LIO monad implements an execution monitor

that keeps track of a *current* label to indicate the sensitivity level of the computation. The current label may get raised, or *tainted*, when the computation depends on sensitive data. Furthermore, sensitive computations are prevented from writing into public channels. In practice, LIO has proven suitable for building production secure web systems [Giffin et al., 2012].

There are plenty of opportunities to optimize away LIO runtime security checks. For example, it is enough to perform a *single check* for computations that, within a long loop, attempt to write to the same channel without affecting the current label. Ideally, runtime checks should only be applied to those parts of the program where sensitive labels are unknown at compile time or susceptible to changes at runtime. Although a state-of-the-art tool, LIO does not support mixing static and dynamic IFC. In this work, we address this shortcoming.

We present HLIO, a Hybrid IFC library which combines the best of both approaches. HLIO statically protects confidentiality while allowing the programmers to *defer* selected checks to be done at runtime. In that manner, security checks involving statically-unknown or prone-to-change labels can be performed at runtime, while providing static guarantees for the rest of the code. Existing LIO code can easily be embedded in HLIO. Furthermore, HLIO provides a very similar interface to LIO. As a result, existing LIO code can also be incrementally refactored to work in HLIO so that programmers can obtain static guarantees where possible. The main purpose of HLIO is making a LIO-like IFC analysis hybrid rather than making LIO better in the kind of leaks it prevents. Specifically, our contributions with this paper are:

- We design and implement HLIO, a hybrid approach to IFC that allows programmers to defer IFC constraints to runtime. (Section 9.4)
- We present a novel technique for embedding HLIO as a library in Haskell. Our technique makes essential use of advanced features of the GHC type system and type inference, namely (a) singleton types [Eisenberg and Weirich, 2012], (b) data promotion [Yorgey et al., 2012], and (c) constraint polymorphism[1], i.e., data types that can be parameterized over type class constraints, to enable deferring IFC checks to runtime. We remark that it is not necessary to understand these advanced type system features in order to use our library. (Sections 9.5 and 9.6)
- We formalize the core features of HLIO in a calculus that allows us to establish a simulation with LIO, thereby showing that HLIO cannot leak secrets, i.e., that it satisfies termination-insensitive noninterference. (Section 9.7)
- As an overall contribution, we describe a general-purpose mechanism for deferring static constraints without any compiler or language modifications. Those constraints can go well beyond IFC, and can even

[1] GHC 7.8.1 manual, Section 7.12

include ordinary type equalities emitted by GHC's type inference engine (see Section 9.8). We thus make it easier for programmers to move across the static/dynamic boundary, following the mantra of Meijer and Drayton "Static typing where possible, dynamic typing when needed!" [Meijer and Drayton, 2005].

9.2 LIO: Flexible Dynamic IFC for Haskell

In this section, we briefly review LIO and its mechanism for dynamically protecting confidentiality of data.

Security Lattices In an IFC system, data gets classified according to its sensitivity degree, which is often denoted by a security label (from now on, just labels). Formally, labels form a lattice *Label* to indicate the allowed flows of information within a program. Data associated with label ℓ_1 can flow into entities labeled as ℓ_2 provided that they respect the order relationship of the lattice, i.e., $\ell_1 \sqsubseteq \ell_2$. The encoding of security lattices can be given as a type class, providing join ($\sqcup$), and the order relationship ($\sqsubseteq$)—see Figure 1. In LIO, this type class also includes a meet ($\sqcap$) operation, but we exclude it from our definition since it is not important for our purposes.[2] Our running example is the classical two-point security lattice, *Label*, that introduces labels L (low) and H (high) to classify data as public and secret, respectively.

$$\begin{aligned}&\textbf{class } \mathit{Lattice}\ \alpha\ \textbf{where}\\ &\quad \sqcup\ :: \alpha \to \alpha \to \alpha\\ &\quad \sqsubseteq\ :: \alpha \to \alpha \to \mathit{Bool}\end{aligned}$$

Fig. 1. Security lattice

$$\begin{aligned}&\textbf{data } \mathit{Label} = L \mid H\\ &\textbf{instance } \mathit{Lattice}\ \mathit{Label}\end{aligned}$$

The *Label* lattice implementation is what one expects; public data can flow into secret entities, i.e., $L \sqsubseteq H$, but not vice versa, i.e., $H \not\sqsubseteq L$.

The LIO Monad LIO provides the *LIO* monad to guarantee that computations manipulate data according to the security lattice—see Figure 2. In [Stefan et al., 2012b], this monad is parametric on the security lattice being considered, but we consider this lattice to be fixed to type *Label* to simplify exposition.

$$\begin{aligned}&\textbf{data } \mathit{LIO}\ a\\ &\textbf{instance } \mathit{Monad}\ \mathit{LIO}\\ &\mathit{getLabel} :: \mathit{LIO}\ \mathit{Label}\\ &\mathit{runLIO} :: \mathit{Label} \to \mathit{LIO}\ a \to \mathit{IO}\ a\end{aligned}$$

Fig. 2. *LIO* interface

[2] Meet is normally used for tracking integrity, e.g. for checking that data has not been corrupted by untrusted parties.

It is expected that untrusted code is written using this monad (and not directly in the IO monad) in order to have some guarantees about its behavior—this can be enforced using other mechanisms [Terei et al., 2012]. *LIO* encapsulates IO actions so that they are only executed when confidentiality is not compromised. To achieve that, the monad keeps track of a label $\ell_{\text{cur}} :: \mathit{Label}$, called the *current floating label* (or current label for short), which can be retrieved at any time by the function $\mathit{getLabel}$. The role of the current label is two-fold. Firstly, it implicitly labels all the data in scope. Secondly, it only allows computations to write to channels that are labeled with $\ell :: \mathit{Label}$ such that $\ell_{\text{cur}} \sqsubseteq \ell$; otherwise, *LIO* aborts execution. For instance, a computation $m :: \mathit{LIO}\ a$ with $\ell_{\text{cur}} = H$ indicates that a secret has already been observed by m—thus, m cannot subsequently write to public channels.

LIO computations have the flexibility to read sensitive data above the current label, but at the cost of raising the current label and thus being more restrictive in subsequent computations. More specifically, when reading data with sensitivity $\ell :: \mathit{Label}$, the current label ℓ_{cur} is raised to $\ell'_{\text{cur}} = \ell_{\text{cur}} \sqcup \ell$—in the LIO terminology, the new current label *floats above the observed data*. Consequently, the current label protects *all the data* that have been observed.

Labeled Expressions As in many other IFC systems, *LIO* provides abstractions to label data with different sensitivity degrees in a fine-grained manner—see Figure 3. Data type $\mathit{Labeled}\ a$ associates an expression of type a with a label in Label. The pure function $\mathit{labelOf}$ can retrieve the label associated with a labeled expression. The functions label and $\mathit{unlabel}$ are used to respectively create and destroy elements of this data type.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Labeled}\ a\\
&\mathit{labelOf} :: \mathit{Labeled}\ a \to \mathit{Label}\\
&\mathit{label} :: \mathit{Label} \to a\\
&\quad \to \mathit{LIO}\ (\mathit{Labeled}\ a)\\
&\mathit{unlabel} :: \mathit{Labeled}\ a \to \mathit{LIO}\ a\\
&\mathit{toLabeled} :: \mathit{Label} \to \mathit{LIO}\ a\\
&\quad \to \mathit{LIO}\ (\mathit{Labeled}\ a)
\end{aligned}
$$

Fig. 3. Labeled expressions

Term $\mathit{label}\ \ell\ x$ creates a labeled expression which associates label ℓ with expression x, only if $\ell_{\text{cur}} \sqsubseteq \ell$. This constraint ensures that *LIO* computations do not allocate data below the current label, which could potentially be returned and read by lower-labeled computations.

Term $\mathit{unlabel}\ x$ never fails; it extracts the data inside a labeled expression x but taints (as a side-effect) the current label by joining it ($\sqcup$) with the label of the expression. From a security point of view, creating a labeled expression with label ℓ can be regarded as writing into a channel at security level ℓ. Similarly, observing (i.e., unlabeling) a labeled expression is analogous to reading from a channel with the same security label. For simplicity, we only consider labeled expressions in this paper—they are

the simplest examples of labeled entities. Nevertheless, LIO does support labeled mutable references [Stefan et al., 2011b], exceptions [Stefan et al., 2012b], and synchronization variables [Stefan et al., 2012a], which could be orthogonally added.

Example 1 *(Tainting ℓ_{cur})* An *LIO* computation only raises its current label when observing (unlabeling) labeled expressions, as the secure string concatenation example below shows:

```
lconcat :: Labeled String → Labeled String → LIO String
lconcat lstr1 lstr2 = do      – Initial current label ℓcur
  str1 ← unlabel lstr1        – ℓ'cur = ℓcur ⊔ (labelOf lstr1)
  str2 ← unlabel lstr2        – ℓ''cur = ℓ'cur ⊔ (labelOf lstr2)
  return (str1 ++ str2)       – Final current label ℓ''cur
```

Label Creep *Label creep* is the problem of raising the current label to such a point where computations are no longer capable of performing useful side-effects [Sabelfeld and Myers, 2003], i.e., the current label becomes "too high, too soon." To address this problem, LIO provides the primitive *toLabeled* (Figure 3) to allow computations to only temporarily raise their current label. Specifically, *toLabeled* ℓ m executes m with the current label ℓ_{cur} at the time of executing this action. It first ensures that $\ell_{\text{cur}} \sqsubseteq \ell$ since it would attach ℓ to the result of m—after all, it is creating a labeled value. Computation m can in turn raise the current label during its execution, to a new ℓ'_{cur}. After m terminates, *toLabeled* checks that $\ell'_{\text{cur}} \sqsubseteq \ell$, and if that is the case, label ℓ is used to protect the sensitivity of the result (in the return value of type *Labeled* a).

In *toLabeled* ℓ m, ℓ is an upper bound on the final current label of m. The reason for that is to avoid leaks by manipulating the current label inside m [Stefan et al., 2011b]. Imagine that the labeled value is instead wrapped with the final current label of m, and that the current label before executing *toLabeled* is set to L. It could happen that in one run, the current label in m is ℓ_1, where $L \sqsubset \ell_1$, and depending on information at that level, it decides to unlabel a piece of data which takes the current label to ℓ_2 ($\ell_2 \not\equiv \ell_1$). After *toLabeled* gets executed, the next instruction simply reads the label of the returned value (*labelOf*), which returns either ℓ_1 or ℓ_2 without raising the current label. In that manner, code with current label L can learn from data at level ℓ_1—an information leak!

Example 2 *(Avoiding label creep)* With *toLabeled* in place, we can provide a more flexible version of *lconcat* as follows.

```
lconcat' :: Labeled String → Labeled String
            → LIO (Labeled String)
lconcat' lstr1 lstr2 = do     – Initial current label ℓcur
```

```
  let lab = labelOf lstr1 ⊔ labelOf lstr2
  lresult ← toLabeled lab (lconcat lstr1 lstr2)
  return lresult   – Final current label ℓcur
```

Observe that *lconcat′*, in contrast with *lconcat*, can concatenate secret strings without raising the current label.

Running LIO Actions without Leaking Secrets Function *runLIO* uses its first argument to initialize the current label and executes the *LIO* action given as its second argument. It returns an *IO* action which is IFC-compliant, i.e., where side-effects do not leak sensitive information with respect to that label.

Example 3 *(Preventing secret leaking)* We describe below a function which runs untrusted code and publishes a returned string value in a public web site.

```
publish :: LIO String → IO String
publish m = do {r ← runLIO L action; report r}
  where
    action = do
      x ← m
      lx ← label L x   – succeeds if ℓcur ⊑ L
      unlabel lx   – ℓcur is not modified
    report s = wget ("http://reports/str=" ++ s) [] []
```

Function *wget* sends an HTTP request to the URL given as argument. The *action* computation runs the untrusted code m but guards the result x with L by calling *label* L. This call only succeeds when the final label of m is less than or equal to L.

Dynamically Labeled Values As mentioned in the introduction, runtime IFC enforcement is particularly useful in systems where values get classified based on runtime information. For instance one can assume (or implement) a primitive that reads a remote labeled value from the network:

```
readRemote :: URI → LIO (Labeled String)
```

The primitive does not necessarily increase the current label as sensitive data can be encapsulated in the labeled value we return. A more realistic example of such a primitive can be found in Appendix A.

Untrusted scripts can freely call *readRemote* without compromising confidentiality since, in order to observe the returned value, they would have to have their current label tainted and thus would be restricted from performing unsafe side-effects. While not a problem for dynamic LIO, we will see in the next section how dynamically labeled data complicates the programming model in a statically-typed IFC discipline.

9.3 SLIO: Static IFC for Haskell

LIO performs information-flow checks at runtime, and hence the ability to discharge those statically is certainly appealing.

Security Labels at the Type Level The first step towards a statically typed version of LIO in Haskell is to transport labels and lattice operations over labels to the type level. We illustrate how this can be done in Haskell for the familiar 2-point lattice:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Label} = \mathit{L} \mid \mathit{H} \\
\textbf{class}\ \mathit{Flows}\ (\ell_1 :: \mathit{Label})\ (\ell_2 :: \mathit{Label}) \\
\textbf{instance}\ \mathit{Flows}\ \mathit{L}\ \mathit{L} \\
\textbf{instance}\ \mathit{Flows}\ \mathit{L}\ \mathit{H} \\
\textbf{instance}\ \mathit{Flows}\ \mathit{H}\ \mathit{H} \\
\textbf{type}\ \mathit{family}\ \mathit{Join}\ (\ell_1 :: \mathit{Label})\ (\ell_2 :: \mathit{Label}) :: \mathit{Label}\ \textbf{where} \\
\quad \mathit{Join}\ \mathit{L}\ \mathit{L} = \mathit{L} \\
\quad \mathit{Join}\ \mathit{L}\ \mathit{H} = \mathit{H} \\
\quad \mathit{Join}\ \mathit{H}\ \mathit{L} = \mathit{H} \\
\quad \mathit{Join}\ \mathit{H}\ \mathit{H} = \mathit{H}
\end{array}
$$

The *Label* datatype constructors will be used at the type-level. In Haskell terminology, *Label* will be a *promoted* datatype [Yorgey et al., 2012]. Moreover, we can represent $\sqsubseteq$-constraints at the type level using the type class $\mathit{Flows}\ (\ell_1 :: \mathit{Label})\ (\ell_2 :: \mathit{Label})$ over labels. The instances of the type class encode specific cases of the $\sqsubseteq$-relationship. [3] We also use a *closed type family* [Eisenberg et al., 2014] *Join* to express the $\sqcup$ computation at the type level.

Ordinary term-level labels can now be indexed by type-level labels, i.e., they can be defined as *singleton types* in the dependent type theory jargon—see Figure 4. In a proper dependently typed language, such as Agda or F*, there would be no need for duplication of labels and lattice functionality at the type level and, in fact, our formal treatment (Section 9.7) does away with the duplication.

$$
\begin{array}{l}
\textbf{data}\ \mathit{SLabel}\ (\ell :: \mathit{Label})\ \textbf{where} \\
\quad \mathit{L} :: \mathit{SLabel}\ \mathit{L} \\
\quad \mathit{H} :: \mathit{SLabel}\ \mathit{H}
\end{array}
$$

Fig. 4. Singleton labels

Although our running example is the 2-point lattice, we have successfully applied similar techniques to implement a more complicated type-level

[3] Although type classes in Haskell are *open*, we can prevent malicious users from introducing bogus instances by employing superclasses and Haskell's export mechanism.

lattice, namely DC-labels [Stefan et al., 2011a], a decentralized security label model for IFC that can express security concerns from different actors in a mutual distrust environment. As far as we know, this is the first implementation of DC-labels at the type level.

An LIO Hoare State Monad Once the type-level machinery is in place, we replace our dynamic *LIO* monad with a *Hoare state monad* [Nanevski et al., 2006], indexed by the initial label of a computation (analogous to a pre-condition) and the final label of a computation (analogous to a post-condition):

$$\textbf{data}\ \mathit{SLIO}\ (\ell_i :: \mathit{Label})\ (\ell_o :: \mathit{Label})\ a$$
$$\mathit{runSLIO} :: \mathit{SLabel}\ \ell_i \to \mathit{SLIO}\ \ell_i\ \ell_o\ a \to \mathit{IO}\ a$$

SLIO is just an intermediate step towards our final solution, but readers can assume a very similar implementation as that of *LIO*: a state monad over the current label.

$$\textbf{type}\ \mathit{SLIO}\ \ell_i\ \ell_o\ a = \mathit{SLabel}\ \ell_i \to \mathit{IO}\ (a, \mathit{SLabel}\ \ell_o)$$

Due to its more expressive type, *SLIO* is not a Haskell monad. Nevertheless, it is a parameterized monad [Atkey, 2006] and more generally a polymonad [Hicks et al., 2014]. As a consequence, it is possible to define meaningful $(\gg\!=)$ and *return* operators that satisfy the usual monad laws:

$$(\gg\!=) :: \mathit{SLIO}\ \ell_1\ \ell_2\ a \to (a \to \mathit{SLIO}\ \ell_2\ \ell_3\ b) \to \mathit{SLIO}\ \ell_1\ \ell_3\ b$$
$$\mathit{return} :: a \to \mathit{SLIO}\ \ell\ \ell\ a$$

It is easy to see how these functions are implemented.

A Statically Typed API for IFC *SLIO* so far seems like a more precise typing of *LIO*. However, the ability to express labels and their operations at the type level immediately opens up the possibility for converting the *dynamic checks* of LIO to *static proof requirements*. We do this below by simply rewriting the dynamic API to use static constraints instead:

$$\textbf{data}\ \mathit{SLabeled}\ (\ell :: \mathit{Label})\ a = \mathit{SLabeled}\ (\mathit{SLabel}\ \ell)\ a$$
$$\mathit{getLabel} :: \mathit{SLIO}\ \ell_i\ \ell_i\ (\mathit{SLabel}\ \ell_i)$$
$$\mathit{labelOf} :: \mathit{SLabeled}\ (\ell :: \mathit{Label})\ a \to \mathit{SLabel}\ \ell$$
$$\mathit{label} :: \mathit{Flows}\ \ell_i\ \ell \Rightarrow \mathit{SLabel}\ \ell \to a$$
$$\quad \to \mathit{SLIO}\ \ell_i\ \ell_i\ (\mathit{SLabeled}\ \ell\ a)$$
$$\mathit{unlabel} :: \mathit{SLabeled}\ \ell\ a \to \mathit{SLIO}\ \ell_i\ (\mathit{Join}\ \ell_i\ \ell)\ a$$
$$\mathit{toLabeled} :: \mathit{SLIO}\ \ell_i\ \ell_o\ a \to \mathit{SLIO}\ \ell_i\ \ell_i\ (\mathit{SLabeled}\ \ell_o\ a)$$

Function *getLabel* returns the current label without affecting it. Function *labelOf* returns the singleton type corresponding to the initial label of the

computation. Function *label* creates a labeled value with label ℓ without modifying the current label ℓ_i, provided that $\ell_i \sqsubseteq \ell$, expressed this time as a static proof obligation $Flows\ \ell_i\ \ell$. Function *unlabel*, on the other hand, *taints* the current label with the value of the labeled expression. Function *toLabeled* has a very simple type: just encapsulate the output label in the labeled value that we return. The careful reader may observe a small disconnect between the static and dynamic versions of *toLabeled*—this is due to a significant simplification that the static world enables, a point we discuss in detail in Section 9.8.

Finally, in order to give a valid type to primitives such as *readRemote*, it is often convenient to hide the label of a labeled value with an existential type, so that it no longer appears in the type. Haskell does not support first-class existential types, so we encode this with a datatype definition:

```
data LabeledX a where
  LabeledX :: (SLabeled (ℓ :: Label) a) → LabeledX a
```

Problems When Programming in SLIO Let us consider how one can program using the *SLIO* primitives. Suppose that we have a function *report* with type

$$report :: Flows\ \ell_i\ L \Rightarrow String \rightarrow SLIO\ \ell_i\ \ell_i\ ()$$

that sends a given *String* to a public server and publishes it on the Internet. This function has a *Flows* type class constraint which specifies that the current label at the time when *report* is run should not exceed L, i.e., the public label. For the simple lattice that we consider in this paper, *report* can effectively be called only when ℓ_i is L. One could imagine more complex situations with a richer label hierarchy, where more than one label is allowed to report or when the label associated with the public server is not fixed to L in advance but is rather dynamically obtained. Such situations would amplify our arguments in the rest of this section, but the simpler *report* above is sufficient for our presentation.

Figure 5 considers the secure string concatenation example *lconcat* (from the previous section), except that we instead use the statically typed counterparts to the LIO operations, and we incorporate a call to *report* in order to publish the result of the concatenation. This function, called *lReport2*, is a perfectly well-typed program with type

```
lReport2 ::
  Flows (Join (Join ℓ_i ℓ_1) ℓ_2) L
```

```
lReport2 lstr_1 lstr_2 =
do v_1 ← unlabel lstr_1
   v_2 ← unlabel lstr_2
   let result = v_1 ++ v_2
   report result
   return result
```

Fig. 5. Static *lReport2*

$$\Rightarrow \mathit{SLabeled}\ \ell_1\ \mathit{String} \rightarrow \mathit{SLabeled}\ \ell_2\ \mathit{String}$$
$$\rightarrow \mathit{SLIO}\ \ell_i\ (\mathit{Join}\ (\mathit{Join}\ \ell_i\ \ell_1)\ \ell_2)\ \mathit{String}$$

Client scripts can call *lReport2* provided that they can satisfy the constraint, which enforces that both strings should be public, i.e., labeled with L. For instance, let us assume that we have $lv_1 :: \mathit{SLabeled}\ L\ \mathit{String}$, $lv_2 :: \mathit{SLabeled}\ L\ \mathit{String}$, and code

$$\mathit{foo} :: \mathit{SLIO}\ L\ L\ \mathit{String}$$
$$\mathit{foo} = \mathit{lReport2}\ lv_1\ lv_2$$

All labels are statically resolved, and *foo* can typecheck as all constraints can be discharged by the type class and type family instances.

Consider now the case where some of the labeled values are dynamically loaded from the network with *readRemote* from the previous section, and we furthermore address the label creep issue by packing the result in a labeled value:

$$\mathit{readRemote} :: \mathit{URI} \rightarrow \mathit{SLIO}\ \ell_i\ \ell_i\ (\mathit{LabeledX}\ \mathit{String})$$
$$\mathit{foo} = \mathbf{do}$$
$$\quad \mathit{LabeledX}\ (lv_1 :: \mathit{SLabeled}\ \ell_1\ \mathit{String}) \leftarrow \mathit{readRemote}\ \mathit{host}_1$$
$$\quad \mathit{LabeledX}\ (lv_2 :: \mathit{SLabeled}\ \ell_2\ \mathit{String}) \leftarrow \mathit{readRemote}\ \mathit{host}_2$$
$$\quad \mathit{toLabeled}\ (\mathit{lReport2}\ lv_1\ lv_2)$$

The program is ill-typed for two reasons. First, the existential label variables ℓ_1 and ℓ_2, arising from unpacking the existentials that we read with *readRemote*, escape in the return type, i.e., $\mathit{SLabeled}\ (\mathit{Join}\ (\mathit{Join}\ \ell_i\ \ell_1)\ \ell_2)\ \mathit{String}$. To address this problem we could pack the return type in an existential (using *LabeledX*) to prevent the existential label from escaping. The modification is shown in Figure 6. However, even if we prevent the escape of existential variables in the return type of *foo*, there is another problem: the existential variables also escape *in the constraint*, i.e., $\mathit{Flows}\ (\mathit{Join}\ (\mathit{Join}\ \ell_i\ \ell_1)\ \ell_2)\ L$, which makes *foo* ill-typed.

$$\mathit{foo} = \mathbf{do}$$
$$\ldots$$
$$r \leftarrow \mathit{toLabeled}\ (\mathit{lReport2}\ lv_1\ lv_2)$$
$$\quad \text{– pack result in existential}$$
$$\mathit{return}\ (\mathit{LabeledX}\ r)$$

Fig. 6. Hiding existential types

Since we do not statically know the remote labels, one may wonder if there is a way to rewrite the program to "assume the worst" (that they are both H) and that the current label after unlabelling them is always—conservatively—H. This option is a non-starter: first, *lReport2* would always be returning high-labeled values, but much more worryingly,

we would not be in a position to call *report* any more, even in the case where the actually read remote labels were both L.

A more appealing way to implement *foo* is to restructure the code to incorporate a *runtime test* that inspects the remote labels:

```
foo = do
  LabeledX lv1 ← readRemote host1
  LabeledX lv2 ← readRemote host2
  case (labelOf lv1, labelOf lv2) of
    (L, L) → do
      lv ← toLabeled (lReport2 lv1 lv2) :: SLIO L L String
      return (LabeledX lv)
    _ → error "Both strings should be public!"
```

The GADT branch on the labels tests for a specific combination of remote labels, which allows the type checker to refine the corresponding type-level labels and discharge all generated constraints. We have also introduced annotations in each branch to fix the SLIO pre- and post-conditions and guide the type inference engine. In the case of the 2-point lattice, the above restructuring is not terrible (only one combination of 4 is a non-error), but more complicated lattices can quickly introduce lots of GADT pattern matches in potentially multiple places inside the user code.

The example illustrates one awkward aspect of the static approach: every time we have to move dynamic data into a statically typed piece of code, programs have to be restructured to introduce runtime tests. While the runtime tests in this situation are *unavoidable*, in this paper we show how to do this *without restructuring* the implementation.

9.4 HLIO: Mixing Static and Dynamic Typing

In HLIO, users can instead take the "natural" way to write *foo* and make the program typeable by using our primitive *defer* (underlined below):[4]

```
foo host1 host2 = do
  LabeledX lv1 ← readRemote host1
  LabeledX lv2 ← readRemote host2
  lv ← defer (toLabeled (lReport2 lv1 lv2))
  return (LabeledX lv)
```

The role of *defer* is to defer static constraints to runtime; in this case, the one which arises from *toLabeled* (*lReport2* lv_1 lv_2). This constraint will be $\ell_i \sqcup \ell_1 \sqcup \ell_2 \sqsubseteq L$, where ℓ_i is the initial label and ℓ_1 and ℓ_2 are the labels of the returned labeled values from the two *readRemote* calls.

[4] We do not yet give type signatures since types slightly differ from the types of the corresponding primitives in SLIO.

To demonstrate how this works, assume that *readRemote* returns a high-labeled value from host "secure.org", but a low-labeled value from "public.org". The following sequence of calls (using *runHLIO*, the *HLIO* analogue of *runSLIO*) shows that indeed our primitive performs the check at runtime:

```
ghci> runHLIO L (foo "secure.org" "public.org")
*** Exception: IFC violation!
ghci> runHLIO L (foo "public.org" "public.org")
Success
ghci> runHLIO H (foo "public.org" "public.org")
*** Exception: IFC violation!
```

In the first case, the first labeled value will contain a high label that taints the current label and results eventually in an IFC exception. In the second case, we only *readRemote* from public domains and hence no exception is thrown. In the final case, although we read from two public sites, we start from an already high label.

The *defer* primitive can be used at every point in the assembly of a computation to selectively defer to runtime the constraints arising from a subcomputation, *at the programmer's will*. For example, the following variations are all well-typed:

$lvL :: SLabeled\ L\ String$ – a statically known public value
$bar\ x = \mathbf{do}$
  $LabeledX\ lv \leftarrow readRemote\ host$
  $s_1 \leftarrow defer\ (toLabeled\ (lReport2\ x\ lv))$
  $s_2 \leftarrow lReport2\ x\ lvL$
  $return\ s_2$
$baz\ x = \mathbf{do}$
  $LabeledX\ lv \leftarrow readRemote\ host$
  $s_1 \leftarrow defer\ (toLabeled\ (lReport2\ x\ lv))$
  $s_2 \leftarrow defer\ (lReport2\ x\ lvL)$
  $return\ s_2$

The difference between *bar* and *baz* lies in the set of constraints they dynamically check; in *bar*, we have to statically discharge the constraints that arise from the computation of s_2, but we will dynamically check the constraints arising from *lReport2 x lv* when computing s_1. In *baz*, we will convert the constraints from s_2 to be runtime checks. In both cases, we *must* defer the constraints that arise from the computation of s_1 as the label of *lv* would otherwise escape in the returned constraint.

The mechanism of *defer* has also the benefit of addressing the incompleteness of type inference engines or type-level lattice specifications—any time we are faced with a constraint that we cannot statically discharge, *defer* will convert it to a runtime check.

Having described the functionality we are aiming for, we now present the HLIO API without yet diving into the internals of its implementation.

Label Expressions Whenever a *getLabel* operation runs, we must produce a runtime representation of the current label, i.e., a singleton. Consider the case where the current label is of the form $Join\ \ell_1\ \ell_2$. When ℓ_1 and ℓ_2 are known statically, we can just apply the type family and compute the resulting label. However, if ℓ_1 and ℓ_2 are existentially quantified, we need a way of computing a singleton for the *Join* by combining the singletons for ℓ_1 and ℓ_2. Therefore, it will be convenient to introduce another promoted datatype that captures unevaluated label *expressions* as well as a type family to reduce them to *Label* types. As we will see in Section 9.6, this additional level of indirection allows us to compute singletons for *Join* and also to defer constraints involving existentials.

```
data LExpr a = LVal a | LJoin (LExpr a) (LExpr a)
type family E (ℓ :: LExpr Label) :: Label where
  E (LVal x) = x
  E (LJoin ℓ1 ℓ2) = Join (E ℓ1) (E ℓ2)
class Flows (E ℓ1) (E ℓ2) ⇒
  FlowsE (ℓ1 :: LExpr Label) (ℓ2 :: LExpr Label)
instance Flows (E ℓ1) (E ℓ2) ⇒ FlowsE ℓ1 ℓ2
data SLabeled (ℓ :: LExpr Label) a =
  SLabeled (SLabel (E ℓ)) a
```

Data type *LExpr Label* captures unevaluated label expressions at the type level, and $\mathcal{E}$ reduces them to *Label* values. The type class *FlowsE* is isomorphic to *Flows*, with the exception that it ranges over *LExpr Label* instead of *Label*. Note that we also redefine the *SLabeled* data type to include arbitrary labeled expressions. Type family *LJoin* encodes $\sqcup$ at the level of types.

HLIO Monad GHC introduces a kind *Constraint* to classify constraints and allows constraint polymorphism [Orchard and Schrijvers, 2010]. This means that ADTs can be parameterized over constraints. HLIO exploits this feature to provide a monad *HLIO* below:

```
data HLIO (c :: Constraint)
          (ℓi :: LExpr Label) (ℓo :: LExpr Label) a
```

The *HLIO* datatype is very similar to *SLIO* except that it also records a constraint c::*Constraint* (we motivate this design choice in Section 9.6). The rest of the HLIO API provides mechanisms to discharge these constraints statically or dynamically. A computation $HLIO\ c\ \ell_i\ \ell_o\ a$ should be read as a computation that, under constraint c and from initial label ℓ_i produces

a value a and raises the current label to ℓ_o. The types of $(\gg\!=)$ and *return* show how constraints are collected:

$$
\begin{aligned}
&(\gg\!=) :: \mathit{HLIO}\ c_1\ \ell_1\ \ell_2\ a \\
&\quad \to (a \to \mathit{HLIO}\ c_2\ \ell_2\ \ell_3\ b) \to \mathit{HLIO}\ (c_1, c_2)\ \ell_1\ \ell_3\ b \\
&\mathit{return} :: a \to \mathit{HLIO}\ ()\ \ell_i\ \ell_i\ a
\end{aligned}
$$

Note that the type of $(\gg\!=)$ creates a *tuple* of constraints (c_1, c_2) by collecting constraints c_1 and c_2 from the sub-computations. The type of *return* collects a trivial constraint $()$.

IFC Functionality HLIO provides the same API as *SLIO*:

$$
\begin{aligned}
&\mathit{labelOf} :: \mathit{SLabeled}\ \ell\ a \to \mathit{SLabel}\ (\mathcal{E}\ \ell) \\
&\mathit{getLabel} :: \mathit{HLIO}\ ()\ \ell_i\ \ell_i\ (\mathit{SLabel}\ (\mathcal{E}\ \ell_i)) \\
&\mathit{unlabel} :: \mathit{SLabeled}\ \ell\ a \to \mathit{HLIO}\ ()\ \ell_i\ (\mathit{LJoin}\ \ell_i\ \ell)\ a \\
&\mathit{label} :: \mathit{SLabel}\ \ell \to a \\
&\qquad \to \mathit{HLIO}\ (\mathit{FlowsE}\ \ell_i\ (\mathit{LVal}\ \ell)) \\
&\qquad\qquad\quad \ell_i\ \ell_i\ (\mathit{SLabeled}\ (\mathit{LVal}\ \ell)\ a) \\
&\mathit{toLabeled} :: \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a \\
&\qquad\quad \to \mathit{HLIO}\ c\ \ell_i\ \ell_i\ (\mathit{SLabeled}\ \ell_o\ a)
\end{aligned}
$$

Unlike in SLIO, *label* just records constraint $\mathit{FlowsE}\ \ell_i\ (\mathit{LVal}\ \ell)$ in its result type—instead of actually constraining the whole type of the function. This is the only *HLIO* primitive that generates a constraint.

Deferring and Simplifying Constraints In addition to the core IFC functionality, *HLIO* adds the ability to *defer* collected constraints, or explicitly *simplify* them in one go:

$$
\begin{aligned}
&\mathit{defer} :: \mathit{Deferrable}\ c \Rightarrow \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a \to \mathit{HLIO}\ ()\ \ell_i\ \ell_o\ a \\
&\mathit{simplify} :: c \Rightarrow \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a \to \mathit{HLIO}\ ()\ \ell_i\ \ell_o\ a
\end{aligned}
$$

The function *defer* accepts an *HLIO* computation that would be typeable under constraint c, and returns a computation that is typeable under *no constraint!* Indeed, the purpose of this combinator is to discharge the constraint by a runtime test. The puzzled reader may wonder how it is even possible to have a sound implementation of *defer*. The magic is in the *Deferrable* type class, which we describe in Section 9.5.

Dually to deferring constraints to runtime, we may require them to be *statically discharged*—function *simplify* allows us to do that. Like *defer*, *simplify* accepts an *HLIO* computation that is typeable under constraint c, and returns a computation that is typeable under the empty constraint provided that we can discharge c statically (hence the quantification $c \Rightarrow$ ...).

Running HLIO Computations Finally, the function that runs *HLIO* computations is analogous to *runSLIO*, except that we require the collected constraints to be provable.[5]

$$runHLIO :: c \Rightarrow SLabel\ \ell \rightarrow HLIO\ c\ (LVal\ \ell)\ \ell_o\ a \rightarrow IO\ a$$

The Rest of the Paper In the rest of the paper, we describe the *Deferrable* class which enables us to implement the *defer* combinator (Section 9.5), and we present the design decisions and the implementation of the HLIO API (Section 9.6). We formalize the core features of HLIO as a calculus and prove non-interference by elaboration to (ordinary) *LIO* (Section 9.7). We discuss other applications of *Deferrable* beyond IFC (Section 9.8).

9.5 Deferrable Constraints

To understand the implementation of HLIO, we first dive into the internals of *Deferrable*. For a given constraint c, an instance of *Deferrable* c defines a single function *deferC*:

```
class Deferrable (c :: Constraint) where
  deferC :: forall a.Proxy c → (c ⇒ a) → a
```

The *Proxy* c argument is a commonly used technique to get around the lack of explicit type applications in the Haskell source language—instead, we provide a never-evaluated *Proxy* c argument that we can provide an annotation for, e.g., $deferC\ (\bot :: Proxy\ (C\ Int))\ m$.

The second argument, $c \Rightarrow a$, represents a computation that can only be executed if we can statically satisfy the constraint c. The return type of defer is plainly the result of that computation.

It should (rightly so) seem impossible to implement an instance of class *Deferrable* for *every* possible constraint c. However, we can provide instances for specific constraints, provided we have enough runtime information around. In what follows, we show how to provide an instance for *FlowsE*. We start by creating a type-class capturing a singleton label:

```
class ToSLabel (ℓ :: LExpr Label) where
  slabel :: LProxy ℓ → SLabel (ℰ ℓ)
instance ToSLabel (LVal H) where slabel _ = H
instance ToSLabel (LVal L) where slabel _ = L
instance (ToSLabel ℓ1, ToSLabel ℓ2)
```

[5] Alternatively, we could equally require that c be simply *Deferrable*, or that c be () and make use of the appropriate *defer* or *simplify* combinators when constructing an HLIO computation.

```
  ⇒ ToSLabel (LJoin ℓ1 ℓ2) where
  slabel _ = case (slabel p1, slabel p2) of
    (H, H) → H
    (H, L) → H
    (L, L) → L
    (L, H) → H
    where p1 = ⊥ :: LProxy ℓ1; p2 = ⊥ :: LProxy ℓ2
```

Note that we have given instances for the full range of label expressions *LExpr Label*.

If we have instances for *ToSLabel* ℓ_1 and *ToSLabel* ℓ_2 around, then we effectively have runtime witnesses for the corresponding singleton labels, and, in that case, it is very simple to provide an instance for *Deferrable* (*FlowsE* ℓ_1 ℓ_2) [6]:

```
instance (ToSLabel ℓ1, ToSLabel ℓ2) ⇒
  Deferrable (FlowsE ℓ1 ℓ2) where
  deferC p m = case (slabel p1, slabel p2) of
    (L, L)  → m
    (L, H)  → m
    (H, H) → m
    (H, L)  → error "IFC violation!"
      where p1 = ⊥ :: LProxy ℓ1; p2 = ⊥ :: LProxy ℓ2
```

The implementation of *deferC* pattern matches against the runtime representations of the labels ℓ_1 and ℓ_2. In each corresponding case, the GADT pattern match (e.g. (L, L) in the first case) allows the type system to refine ℓ_1 and ℓ_2 (e.g. $\ell_1 := LVal\ L$ and $\ell_2 := LVal\ L$ in the first case). Thus, *every constraint* *FlowsE* ℓ_1 ℓ_2 required by m can be refined (e.g. to *FlowsE* (*LVal L*) (*LVal L*) in the first case) and can be readily discharged by top-level instances for *FlowsE*. It is still possible to forget to include some of the cases, but this will only make the test more conservative.

Note that in the fourth case above (for which no instance exists!), we have no way of calling m, i.e., *deferC* would be ill-typed if we tried. This case corresponds to a genuine runtime error, and we return an *error* indicating a violation of the IFC policy.

Constraints will be collected together in tuples through uses of $(\gg\!=)$ and hence we also provide an instance for pairs of constraints, i.e., *Deferrable* (c_1, c_2), which can be found in Appendix C.

Finally, we also revisit our definition of *LabeledX* to include a dictionary for *ToSLabel* to produce a singleton for the existentially-quantified label.

```
data LabeledX a where
  LabeledX :: ToSLabel ℓ ⇒ SLabeled (ℓ :: Label) a
                         → LabeledX a
```

[6] Readers can ignore the proxy arguments p, p_1 and p_2.

This is necessary for applying *defer* to computations involving labeled expressions that have been unpacked from a *LabeledX*.

The *Deferrable* class is an extremely powerful abstraction for transforming static errors to dynamic checks, and we later show that even type checker equalities generated by the compiler inference mechanism can be defered (Section 9.8). We proceed to show how *Deferrable* can be used to implement the *defer* primitive.

9.6 HLIO Design and Implementation

In Haskell, we embed HLIO as a GADT where the constructors correspond to the primitives described in Section 9.4. More specifically, data type *HLIO* has constructors *Return*, *Bind*, *Unlabel*, *Label*, *ToLabeled*, *GetLabel*, *Defer*, and *Simplify*, which represent uninterpreted commands *return*, *bind*, *unlabel*, *label*, *toLabeled*, *getLabel*, *defer*, and *simplify*, respectively. The types for these constructors match the types given for the commands they represent. In order to give semantics to *HLIO* terms, we provide an interpretation function *go* with the type

$$\begin{aligned}&go :: forall\ c\ \ell_i\ \ell_o\ a.HLIO\ c\ \ell_i\ \ell_o\ a\\ &\quad \to (c \Rightarrow SLabel\ (\mathcal{E}\ \ell_i) \to IO\ (a, SLabel\ (\mathcal{E}\ \ell_o)))\end{aligned}$$

The interpretation of *HLIO* is in an *IO* monad combined with a state to represent the current label (in the style of LIO). Although it might be tempting to get rid of the runtime representation of the current label, this is not possible since code is allowed to inspect it at any time (as a runtime value) using *getLabel*.

$$\begin{aligned}&go\ (Return\ x)\ \ell_i = return\ (x, \ell_i)\\ &go\ (Bind\ m\ f)\ \ell_i = \mathbf{do}\ (a, \ell_i') \leftarrow go\ m\ \ell_i; go\ (f\ a)\ \ell_i'\\ &go\ (GetLabel\ \ell_i) = return\ (\ell_i, \ell_i)\\ &go\ (Unlabel\ (SLabeled\ \ell\ v))\ \ell_i = return\ (v, \ell_i\ `ljoin`\ \ell)\\ &go\ (Label\ \ell\ a)\ \ell_i = return\ (SLabeled\ \ell\ a, \ell_i)\\ &go\ (ToLabeled\ (m :: HLIO\ c\ \ell_i\ \ell_o'\ a'))\ \ell_i = \mathbf{do}\\ &\quad (x, \ell_o) \leftarrow go\ m\ \ell_i; return\ (SLabeled\ \ell_o\ x, \ell_i)\\ &go\ (Defer\ slio)\ \ell_i = deferC\ (setProxy\ slio)\ (go\ slio\ \ell_i)\\ &\quad \mathbf{where}\ setProxy :: HLIO\ c\ \ell_i\ \ell_o\ a \to Proxy\ c\\ &\qquad setProxy = error\ \texttt{"Proxy!"}\\ &go\ (Simplify\ m)\ \ell_i = go\ m\ \ell_i\end{aligned}$$

The interesting cases are the definitions for *Unlabel*, *Defer*, and *Simplify*. For *Unlabel*, *go* performs an ordinary term-level *ljoin*:

$$ljoin :: SLabel\ \ell_1 \to SLabel\ \ell_2 \to SLabel\ (Join\ \ell_1\ \ell_2)$$

but we never get to *inspect* the return label unless we explicitly perform a *getLabel* and subsequently strictly use the label, or unless we perform some form of runtime check. For *Defer*, *go* applies the technique from Section 9.5 with the appropriate proxy. *Simplify* executes m, but exposing its constraints to GHC in order to statically discharge them.

We briefly motivate some of the design choices made in HLIO.

(Singleton Classes) We have seen in the previous section that the motivation for a type-class *ToSLabel* ℓ containing a singleton *SLabel* $(\mathcal{E}\ \ell)$ comes from the need for deferring *FlowsE* constraints.

(LExpr Label and Deferrable) When describing SLIO, we used the *Label* datatype and the *Flows* type class. However, HLIO shifted to the datatype *LExpr Label* and the *FlowE* type class to be able to defer constraints. To illustrate the reason behind that, consider an alternative *Deferrable* instance, without all the *LExpr* complications, and where *ToSLabel* was indexed by *Label*:

instance $(\mathit{ToSLabel}\ (\ell_1 :: \mathit{Label}), \mathit{ToSLabel}\ (\ell_2 :: \mathit{Label})) \Rightarrow$
$\mathit{Deferrable}\ (\mathit{Flows}\ \ell_1\ \ell_2)$ **where**

With this definition, we may find ourselves in need of deferring constraints of the form $\mathit{Flows}\ (\mathit{Join}\ \ell_1\ L)\ L$, where *Join* is the $\sqcup$-operation type family implementation directly on *Label*s. But type class axioms do not match on type families! (They only match on rigid type constructors.) Consequently, it is *impossible* to discharge that constraint either statically or dynamically. In contrast, by exposing a rigid constructor *LJoin*, we were able to give instances for the join of two labels; with our approach, *it is* true that the constraint $\mathit{ToSLabel}\ (\mathit{LJoin}\ \ell_1\ L)$ is automatically discharged from $\mathit{ToSLabel}\ \ell_1$.

(Use of a GADT) We chose to represent HLIO computations using a GADT and then interpreting them using the *go* function. This is a somewhat arbitrary choice. A shallow embedding where $\mathit{HLIO}\ c\ \ell_i\ \ell_o\ a$ is isomorphic to $c \Rightarrow \mathit{IO}\ a$ seems to be possible, though we found the deep embedding more straightforward, allowing potential optimisations to be performed by the *go* function. We remark that the programmer is neither expected to construct these computations nor to invoke the *go* function explicitly; these can be hidden behind the interface.

(Embedding Constraints in HLIO) The introduction of constraint c as part of the *HLIO* definition achieves a purely syntactic manipulation of constraints, and excludes *any possible simplification* by GHC—except when the programmer explicitly requires so with *simplify*. This aspect is beneficial for two reasons: Firstly, this allows us to prevent eager simplification of certain constraints into a form that cannot be deferred or even

discharged. For instance, imagine that a constraint $FlowsE\ \ell_1\ \ell_2$ floats outside of the *HLIO* type. In this case, GHC tries to discharge it by proving $Flows\ (\mathcal{E}\ \ell_1)\ (\mathcal{E}\ \ell_2)$. However, as we discussed before, type class axioms do not match on type families. Moreover, even if that were possible, deferring such a constraint would require instances of $ToSLabel\ (LVal\ (\mathcal{E}\ \ell_1))$ and $ToSLabel\ (LVal\ (\mathcal{E}\ \ell_2))$, which cannot be constructed from instances of $ToSLabel\ \ell_1$ or $ToSLabel\ \ell_2$. Secondly, when evaluating a *defer* expression, the constraint c in *HLIO* makes it possible for the *go* function to automatically supply a proxy to instantiate c (by ***unification***) for a particular constraint in the type of $deferC$, thus allowing the type checker to select the right instance of *Deferrable* without any help from the programmer. If we were not collecting the constraint c in *HLIO*, the programmer would have to supply these proxies explicitly, making HLIO much more cumbersome to use.

In summary, we have chosen to keep the constraints in their unsimplified form as much as possible, and give the programmer the freedom to decide whether they are to be checked statically or dynamically via explicit annotations (*simplify* and *defer*).

9.7 Formal Semantics and Non-interference

In this section, we formalize HLIO and provide security guarantees for our approach by interpreting HLIO in LIO and showing an equivalence in the security checks performed by both systems.

Figure 7 presents the security-relevant rules for a type system for HLIO. The remaining rules can be found in Figure 11 in Appendix C. The terms of HLIO are the same as in LIO, with the addition of the *defer* construct. A lattice expression ℓ is either a primitive label *Label*, a ***join*** operation ($\sqcup$), or a ***meet*** operation ($\sqcap$). A constraint c is either the empty constraint $()$, a pair of two constraints $((c, c))$, or a flow constraint among label expressions ($\ell \sqsubseteq \ell$). The type *HLIO* is a Hoare state monad in the style of statically-typed LIO, as presented in Section 9.3, except that it also includes a constraint c. A computation with type $HLIO\ c\ \ell_i\ \ell_o\ \tau$ is subject to constraints c, and takes the current label from ℓ_i to ℓ_o, and produces a value of type τ. The type $SLabeled\ \ell\ \tau$ represents expressions with label ℓ and type τ, and the type $Label\ \ell$ is a ***singleton*** type for label ℓ, i.e., a type with a single total inhabitant, which can be identified with ℓ.

The typing rule for *return* simply states that the current label is not changed and no constraints need to be checked. Rule (BIND) looks like the usual typing rule for ($\gg\!=$), but it additionally combines the constraints generated by m and f (c and c') into one and also expresses that the final label of computation m should match the initial label of the computation produced by f. Rule (LABEL) generates a security check as a constraint ($\ell_i \sqsubseteq \ell$), and also expresses that the current label does not change. Note

that in this rule we also check the connection between term-level s and type-level ℓ, by using the singleton type. Rule (UNLABEL) reflects the fact that unlabeling an expression labeled ℓ raises the current label ℓ_i to the join $\ell_i \sqcup \ell$. Rule (TOLABELED) checks that the subcomputation m has a valid HLIO type, and expresses that the *toLabeled* computation will not change the current label (or rather, that it will be restored after m finishes), and also that the resulting value of type a is protected by label ℓ_o, i.e., the maximum (and final) label attained by m. Rule (DEFER) checks that the subcomputation m has a valid type and hides the constraints produced by m, so that the expression *defer* m is subject to no static checks.

9.7.1 Semantics for LIO

Figure 8 shows the semantics of LIO, which we will use to interpret HLIO. The semantics closely follows previous work on LIO [Stefan et al., 2011b], given as a small-step operational semantics based on a transition relation $\longrightarrow$ between configurations of the form $\langle \ell_{\text{cur}} \mid t \rangle$, where ℓ_{cur} is the current label and t is the term being evaluated. As before, we only show the rules for computations with security-relevant effects. The full presentation also includes a relation for pure computation ($\rightsquigarrow$), which is used in the rule for *labelOf*, but we elide the details since they are not relevant for our purposes. The semantics uses Felleisen-style evaluation contexts to specify evaluation order, where *Ep* stands for contexts for pure computations and E stands for contexts for effectful ones. As usual, we define $\longrightarrow^*$ to be the reflexive and transitive closure of $\longrightarrow$. Additionally, our transitions are labeled by the information-flow constraints that are being checked at runtime, as can be seen in rule (LABEL). We write $A \xrightarrow{c}{}^* B$ if $A \longrightarrow^* B$ while performing the set of security checks c. For technical reasons, we also include a nonstandard primitive *eval* which is used to force pure computations. Despite not being a part of LIO, we remark that it acts on pure values and its evaluation involves no security-relevant effects, so it is easy to prove that the calculus is still sound after adding it. Essentially, LIO already includes a way to force evaluation for booleans (if statements), so *eval* is merely a generalization of this construct.

9.7.2 Semantics for HLIO

Figure 9 introduces the functions *interp* and *toLIO*, which we use to interpret HLIO and relate this interpretation with the corresponding standard LIO semantics. These two functions are defined as term-to-term transformations. The returned term, however, only utilizes LIO primitives. (Function *interp* closely follows the definition of function *go* described in Section 9.6.)

The function *interp* provides an interpretation of HLIO in dynamic LIO [Stefan et al., 2012b]. Given a well-typed HLIO computation m,

$$
\begin{array}{lll}
\textit{Values} & v & ::= \textit{True} \mid \textit{False} \mid () \mid \lambda x.t \mid \textit{Label} \\
 & & \quad \mid \textit{LIO}^{\text{TCB}}\ t \mid \textit{SLabeled}^{\text{TCB}}\ \ell\ t \\
\textit{Terms} & t & ::= v \mid x \mid t\ t \mid \textit{fix}\ t \mid \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t \\
 & & \quad \mid t\ \otimes\ t \mid \textit{return}\ t \mid t \gg\!= t \mid \textit{getLabel} \\
 & & \quad \mid \textit{label}\ t\ t \mid \textit{unlabel}\ t \mid \textit{labelOf}\ t \\
 & & \quad \mid \textit{toLabeled}\ t\ t \mid \textit{defer}\ t \\
\textit{LOps} & \otimes & ::= \sqcup \mid \sqcap \mid \sqsubseteq \\
\textit{Lattice} & \ell & ::= \textit{Label} \mid \ell\ \sqcup\ \ell \mid \ell\ \sqcap\ \ell \\
\textit{Constraints} & c & ::= () \mid (c, c) \mid \ell\ \sqsubseteq\ \ell \\
\textit{Types} & \tau & ::= \textit{Bool} \mid () \mid \tau \to \tau \mid \textit{HLIO}\ c\ \ell_i\ \ell_o\ \tau \\
 & & \quad \mid \textit{SLabeled}\ \ell\ \tau \mid \textit{Label}\ \ell
\end{array}
$$

RETURN
$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash \textit{return}\ x : \textit{HLIO}\ ()\ \ell_i\ \ell_i\ \tau}$$

BIND
$$\frac{\Gamma \vdash m : \textit{HLIO}\ c\ \ell_i\ \ell\ a \qquad \Gamma \vdash f : a \to \textit{HLIO}\ c'\ \ell\ \ell_o\ b}{\Gamma \vdash m \gg\!= f : \textit{HLIO}\ (c, c')\ \ell_i\ \ell_o\ b}$$

LABEL
$$\frac{\Gamma \vdash t : a \qquad \Gamma \vdash s : \textit{Label}\ \ell}{\Gamma \vdash \textit{label}\ s\ t : \textit{HLIO}\ (\ell_i\ \sqsubseteq\ \ell)\ \ell_i\ \ell_i\ (\textit{SLabeled}\ \ell\ a)}$$

UNLABEL
$$\frac{\Gamma \vdash v : \textit{SLabeled}\ \ell\ a}{\Gamma \vdash \textit{unlabel}\ v : \textit{HLIO}\ ()\ \ell_i\ (\ell_i\ \sqcup\ \ell)\ a}$$

TOLABELED
$$\frac{\Gamma \vdash m : \textit{HLIO}\ c\ \ell_i\ \ell_o\ a}{\Gamma \vdash \textit{toLabeled}\ m : \textit{HLIO}\ c\ \ell_i\ \ell_i\ (\textit{SLabeled}\ \ell_o\ a)}$$

DEFER
$$\frac{\Gamma \vdash m : \textit{HLIO}\ c\ \ell_i\ \ell_o\ a}{\Gamma \vdash \textit{defer}\ m : \textit{HLIO}\ ()\ \ell_i\ \ell_o\ a}$$

Fig. 7. Type system for HLIO (main rules).

$$
\begin{array}{rcl}
Ep & ::= & Ep\ t \mid \mathit{fix}\ Ep \mid \textbf{if}\ Ep\ \textbf{then}\ t\ \textbf{else}\ t \mid Ep \otimes t \mid v \otimes Ep \\
 & & \mid \mathit{label}\ Ep\ t \mid \mathit{unlabel}\ Ep \mid \mathit{labelOf}\ Ep \mid \mathit{toLabeled}\ Ep\ t \\
E & ::= & [\,] \mid Ep \mid E \gg\!= t
\end{array}
$$

$$\textsc{getLabel}\quad \frac{}{\langle \ell_{\text{cur}} \mid E\ [\mathit{getLabel}] \rangle \longrightarrow \langle \ell_{\text{cur}} \mid E\ [\mathit{return}\ \ell_{\text{cur}}] \rangle}$$

$$\textsc{toLabeled}\quad \frac{\ell_{\text{cur}} \sqsubseteq \ell \qquad \langle \ell_{\text{cur}} \mid t \rangle \xrightarrow{c}{}^{*} \langle \ell'_{\text{cur}} \mid \mathit{LIO}^{\text{TCB}}\ t' \rangle \qquad \ell'_{\text{cur}} \sqsubseteq \ell}{\langle \ell_{\text{cur}} \mid E\ [\mathit{toLabeled}\ \ell\ t] \rangle \xrightarrow{c} \langle \ell_{\text{cur}} \mid E\ [\mathit{label}\ \ell\ t'] \rangle}$$

$$\textsc{label}\quad \frac{\ell_{\text{cur}} \sqsubseteq \ell}{\langle \ell_{\text{cur}} \mid E\ [\mathit{label}\ \ell\ t] \rangle \xrightarrow{\ell_{\text{cur}} \sqsubseteq \ell} \langle \ell_{\text{cur}} \mid E\ [\mathit{return}\ (\mathit{SLabeled}^{\text{TCB}}\ \ell\ t)] \rangle}$$

$$\textsc{unlabel}\quad \frac{\ell'_{\text{cur}} = \ell_{\text{cur}} \sqcup \ell}{\langle \ell_{\text{cur}} \mid E\ [\mathit{unlabel}\ (\mathit{SLabeled}^{\text{TCB}}\ \ell\ t)] \rangle \longrightarrow \langle \ell'_{\text{cur}} \mid E\ [\mathit{return}\ t] \rangle}$$

$$\textsc{Eval}\quad \frac{t \rightsquigarrow^{*} v}{Ep\ [\mathit{eval}\ t] \rightsquigarrow Ep\ [v]} \qquad \textsc{labelOf}\quad \frac{}{Ep\ [\mathit{labelOf}\ (\mathit{SLabeled}^{\text{TCB}}\ \ell\ t)] \rightsquigarrow Ep\ [\ell]}$$

Fig. 8. Evaluation contexts and reduction rules.

interp m runs m without performing any security checks, except for those in *defer*. This fact can be seen in the definition for *label*—the case where security side-effects are triggered. This case simply synthesizes a labeled term ($\mathit{SLabeled}^{\text{TCB}}\ \ell\ t$), thus skipping any security check. The *unlabel* operation performs no security checks, so its interpretation is exactly the same as in LIO. The interpretation of labeled terms are simply cast into dynamic labeled terms in *LIO*, where the dynamic label is determined by static information (i.e., $\mathit{SLabeled}^{\text{TCB}}\ t{:}\mathit{SLabeled}\ \ell\ a$). In the interpretation of *defer*, we use the *guards* command, which takes a set of constraints and checks all of them at runtime, aborting the program if any of them fails. These constraints are checked in one go, before running the subcomputation itself. The static version of *toLabeled* is translated into its dynamic counterpart, where the final current label (after executing m) is predicted to be ℓ_o. The interpretation of ($\gg\!=$) simply applies *interp* to its arguments.

Different from *interp*, the function *toLIO* directly translates an HLIO computation into a dynamic LIO computation where *all* the security checks occur dynamically. The translation for labeled terms, *toLabeled*, and ($\gg\!=$) are defined similarly as in *interp*. Label and unlabel, however, simply refor-

$$
\begin{aligned}
&interp\ (label\ t\ t') = SLabeled^{\text{TCB}}\ (eval\ t)\ (interp\ t')\\
&interp\ (unlabel\ t) = unlabel\ (interp\ t)\\
&interp\ (SLabeled^{\text{TCB}}\ t : SLabeled\ \ell\ \tau) = SLabeled^{\text{TCB}}\ \ell\ (interp\ t)\\
&interp\ (defer\ (m : HLIO\ c\ \ell_i\ \ell_o\ \tau)) = guards\ c \gg interp\ m\\
&interp\ (toLabeled\ (m : HLIO\ c\ \ell_i\ \ell_o\ \tau)) =\\
&\quad toLabeled\ \ell_o\ (interp\ m)\\
&interp\ (m \gg\!= f) = interp\ m \gg\!= interp.f\\
&\ldots\\
&toLIO\ (label\ t\ t') = label\ t\ (toLIO\ t')\\
&toLIO\ (unlabel\ t) = unlabel\ (toLIO\ t)\\
&toLIO\ (SLabeled^{\text{TCB}}\ t : SLabeled\ \ell\ a) = SLabeled^{\text{TCB}}\ \ell\ (toLIO\ t)\\
&toLIO\ (defer\ m) = toLIO\ m\\
&toLIO\ (toLabeled\ (m : HLIO\ c\ \ell_i\ \ell_o\ a)) =\\
&\quad toLabeled\ \ell_o\ (toLIO\ m)\\
&toLIO\ (m \gg\!= f) = toLIO\ m \gg\!= toLIO.f\\
&\ldots
\end{aligned}
$$

Fig. 9. The functions *interp* and *toLIO*. The missing equations just behave homomorphically.

mulate the command in *LIO*, where the corresponding security side-effects might be triggered.

9.7.3 Non-interference

We define the simulation relation $\sim$, which expresses that two terminating programs perform the same information flow checks and compute the same values.

Definition 1 *(Simulation between LIO terms) Let A and B be LIO configurations, then* $A \sim B$ *iff* $A \xrightarrow{c}{}^{*} X$ *and* $B \xrightarrow{c}{}^{*} X$*, where X is A's weak head normal form. Note that we only consider terminating programs due to the fact that LIO only provides security guarantees for terminating runs.*

We define a big-step evaluation relation $\Downarrow$ for HLIO terms.

Definition 2 *(Big-step semantics for HLIO) Given an HLIO term,* $(t : HLIO\ c\ \ell_i\ \ell_o\ \tau) \Downarrow v$ *if and only if* $\langle \ell_i \mid interp\ t\rangle \xrightarrow{c'}{}^{*} \langle \ell_o \mid toLIO\ v\rangle$.

The definition leverages the LIO semantics. It applies *interp* to the term being reduced as well as *toLIO* to the result. Observe that *toLIO* is needed for cases where v still contains HLIO terms, e.g., when v is composed of nested labeled terms.

The next lemma (see details in Appendix B) introduces a relationship between the security checks done by HLIO and LIO.

Lemma 1 (Simulation between HLIO and LIO terms)
Given that $(t : HLIO\ c\ \ell_i\ \ell_o\ \tau) \Downarrow v$, *then* $\langle \ell_i \mid guards\ c \gg interp\ t \rangle \sim \langle \ell_i \mid toLIO\ t \rangle$.

The lemma states that if we take the statically-determined constraints c for a well-typed term t into account, we can prove that the programs $guards\ c \gg interp\ t$ and $toLIO\ t$ are in simulation with respect to their security checks and final values. The former performs all statically-determined security checks in the beginning, and then runs the program with the deferred checks. The latter is obtained by viewing the original program as an LIO program, where all *defer* operations are removed.

The semantic correspondence from Lemma 1 guarantees that if an HLIO program is well-typed and terminates successfully, then the equivalent LIO program would also terminate successfully. Conversely, if the LIO program fails with a security error, the HLIO program will either not have a type or fail during a *defer* computation. Since the HLIO and LIO enforcement mechanisms are equivalent in this sense, and LIO enforces noninterference [Stefan et al., 2011b], we can show that HLIO enforces the same property.

For our security guarantees, we consider an attacker at sensitivity level l, who can only observe values at a security level at most l. LIO defines two terms t_1 and t_2 to be l-equivalent (written $t_1 \approx_l t_2$) if the attacker is unable to distinguish between them, e.g. $SLabeled^{TCB}\ L\ 3 \approx_l SLabeled^{TCB}\ L\ 3$ and $SLabeled^{TCB}\ H\ 1 \approx_l SLabeled^{TCB}\ H\ 5$, but $SLabeled^{TCB}\ L\ 2 \not\approx_l SLabeled^{TCB}\ L\ 1$—LIO also extends this notion to configurations. We leverage LIO definitions to express our non-interference theorem—after all, HLIO gets interpreted in LIO!

Noninterference expresses the notion that a program cannot leak secrets. Intuitively, a program is noninterfering if, considering two independent runs with l-equivalent inputs, their final values are also l-equivalent. In other words, attackers cannot distinguish the values of secret inputs by observing the outputs.

Theorem 1 (Termination-insensitive noninterference)
Given HLIO terms t_1 *and* t_2 *with no constructors* $\cdot^{TCB}$ *such that constraints* c_1 *and* c_2 *hold,* $(t_1 : HLIO\ c_1\ \ell_1\ \ell_2\ \tau) \Downarrow v_1$, $(t_2 : HLIO\ c_2\ \ell_i\ \ell_o\ \tau') \Downarrow v_2$, *and* $\langle \ell_i \mid toLIO\ t_1 \rangle \approx_l \langle \ell_i \mid toLIO\ t_2 \rangle$, *then it holds that* $\langle \ell_o \mid toLIO\ v_1 \rangle \approx_l \langle \ell_o \mid toLIO\ v_2 \rangle$.

Proof sketch 1 *The proof uses Lemma 1 to relate the reductions of* $interp\ t_1$ *and* $interp\ t_2$ *with* $toLIO\ t_1$ *and* $toLIO\ t_2$, *respectively. Once that is done, the result follows by applying the LIO non-interference theorem in [Stefan et al., 2012b]. This theorem requires that* t_1 *and* t_2 *do not include constructors of the form* $\cdot^{TCB}$. *Consequently, observe that it is not possible to directly consider* l*-equivalence between interpreted terms, i.e.,* $interp\ t_1$ *and* $interp\ t_2$*—they introduce constructors* $SLabeled^{TCB}$ *to avoid security checks. The proof is given in Appendix B.*

The theorem indicates that l-equivalent (fully) dynamic interpretations of HLIO terms (i.e., $\langle \ell_i \mid \mathit{toLIO}\ t_1 \rangle \approx_l \langle \ell_i \mid \mathit{toLIO}\ t_2 \rangle$), where the static checks hold, produce l-equivalent results (in LIO) (i.e., $\langle \ell_o \mid \mathit{toLIO}\ v_1 \rangle \approx_l \langle \ell_o \mid \mathit{toLIO}\ v_2 \rangle$). Observe that if any HLIO terms leaked secrets, l-equivalence involving v_1 and v_2 would not hold.

9.8 Discussion

This section explains some design choices, while exploring others.

The toLabeled Function The *HLIO* type for *toLabeled* deserves some attention. From Section 9.2, we know that $\mathit{toLabeled}\ \ell\ m$ in *LIO* performs two security checks: $\ell_{\text{cur}} \sqsubseteq \ell$ at the begining of *toLabeled*, and $\ell'_{\text{cur}} \sqsubseteq \ell$ where ℓ'_{cur} is the current label obtained by evaluating m. A directly corresponding static version of *toLabeled* (and its dynamic checks) might be:

$$\begin{aligned}&\mathit{toLabeled} :: \mathit{Label}\ \ell \to \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a \\ &\quad \to \mathit{HLIO}\ (c, \mathit{FlowsE}\ \ell_i\ \ell, \mathit{FlowsE}\ \ell_o\ \ell)\ \ell_i\ \ell_i\ (\mathit{SLabeled}\ \ell\ a)\end{aligned}$$

Recall that, for security reasons, the role of the first argument (of type $\mathit{Label}\ \ell$) is to statically predict an upper bound of the current label obtained by running m. The constraints in the return type of *toLabeled* express this fact. In HLIO, however, that prediction is already given! Observe that type $m :: \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a$ says "after running m, the final current label is ℓ_o." We can use ℓ_o as the upper bound, i.e., $\ell \equiv \ell_o$, and remove the static check $\mathit{FlowsE}\ \ell_o\ \ell$. Moreover, we know that $\ell_i \sqsubseteq \ell_o$ by construction, which allows the removal of $\mathit{FlowE}\ \ell_i\ \ell$. By taking all these facts together, we can dismiss all the extra constraints.

$$\mathit{toLabeled} :: \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a \to \mathit{HLIO}\ c\ \ell_i\ \ell_i\ (\mathit{SLabeled}\ \ell_o)\ a$$

In Section 9.7, we have formally proved that this primitive is secure by establishing a simple relationship with its counterpart in LIO.

Conditionals The monad *HLIO* is embedded in Haskell as a GADT, so it is possible to use Haskell's **if** statements to express conditional branching. However, the Haskell type system requires that the types of both branches be the same. In particular, if the branches are *HLIO* computations, their types must also completely agree, *including* constraints and initial and final labels. Unfortunately, this means that it is not possible to have **if** statements where one branch produces a constraint and the other one does not or, more generally, where the branches produce different sets of constraints. For example, the following expression, where $x :: \mathit{Int}$, is ill-typed:

$$\textbf{if}\ x > 0\ \textbf{then}\ (\mathit{label}\ H\ x \gg \mathit{return}\ x)\ \textbf{else}\ \mathit{return}\ (x + 1)$$

The reason for the type error is that one branch has type $HLIO\ (Flows\ \ell_i\ H)\ \ell_i\ \ell_i\ Int$, while the other one has type $HLIO\ ()\ \ell_i\ \ell_i\ Int$. When it comes to disparities in the constraints, it is possible to work around this restriction by means of *defer* operations. The programmer can use *defer* to check one or both of the branches dynamically, which causes the constraints in the *HLIO* type to be (), thus keeping the Haskell type checker happy. However, if the current label is not updated in exactly the same way in both branches, the **if** statement will also be ill-typed. Note that this cannot be solved with *defer*.

An alternative solution that addresses the problem with both constraints and the current label involves adding another primitive for **if** statements, i.e., a constructor *If* for the *HLIO* GADT. The type of this constructor would accurately express the connection between constraints and current labels in both branches, as follows:

$$\begin{aligned} &If :: Bool \to HLIO\ c_1\ \ell_i\ \ell_o\ a \to HLIO\ c_2\ \ell_i\ \ell'_o\ a \\ &\quad \to HLIO\ (c_1, c_2)\ \ell_i\ (LJoin\ \ell_o\ \ell'_o)\ a \end{aligned}$$

Essentially, the primitive would over-approximate the constraints and the final label, as can be expected from a static analysis. This solution would not only introduce notational overhead but also complicate the formal treatment of HLIO significantly, as we would no longer have a one-to-one correspondence between static and dynamic checks. Instead, we could prove that the dynamic checks are a *subset* of the statically-determined constraints. In order to simplify our exposition, we chose to avoid this solution, but we believe it would be a reasonably straightforward extension.

Deferring Constraints beyond Non-interference The *Deferrable* type class enables programmers to give instances for deferring the check for a constraint to runtime. In this section, we show how to push this idea to the extreme, by deferring the check for type equalities that are generated by GHC's type inference. We iterate that the code in this section (and everywhere in this paper) requires no modifications to GHC.

We wish to defer a type equality between two types ta and tb, which in GHC type system would be expressed as $ta{\sim}tb$ of kind *Constraint*. Of course, in order to perform such a test at runtime, we need to have *runtime type information* around about the shape of types ta and tb. GHC provides the *Typeable* type class that captures runtime type representations. This enables the following instance definition:

```
instance (Typeable a, Typeable b) ⇒
  Deferrable (a∼b) where
  defer _p m = case eqT :: Maybe (a :∼: b) of
    Nothing → error "type error!"
    Just Refl → m
```

Function eqT is a standard library function, providing a runtime witness of the equality of two types that are instances of *Typeable*:

$$eqT :: (Typeable\ b, Typeable\ a) \Rightarrow Maybe\ (a :\sim: b)$$

and $a :\sim: b$ is a GADT expressing with its only constructor *Refl* the fact that a and b are in fact equal:

$$\textbf{data}\ (a :\sim: b)\ \textbf{where}\ Refl :: (a \sim b) \Rightarrow (a :\sim: b)$$

If programmers write a program that contains a type error:

$$foo :: forall\ a.a \to a \to a$$
$$foo\ x\ y = \textbf{if}\ x\ \textbf{then}\ False\ \textbf{else}\ y$$

GHC will report: `Couldn't match expected type `Bool' with actual type `a'`. As we may, in fact, apply *foo* to two boolean values at runtime, programmers may want to make this program typeable by deferring the constraint:

$$foo :: forall\ a.Typeable\ a \Rightarrow a \to a \to a$$
$$foo\ x\ y = defer\ p\ (\textbf{if}\ x\ \textbf{then}\ False\ \textbf{else}\ y)$$
$$\quad \textbf{where}\ p :: Proxy\ (a \sim Bool) = \bot$$

In this case, *foo True False* returns *False*, while *foo* 3 4 produces `*** Exception: type error`. Note that this behavior is different from related work [Vytiniotis et al., 2012], which defers unsatisfiable constraints as errors to runtime. Instead, we do genuinely defer the check at the (unavoidable) cost of having the type representation around.

9.9 Related work

Hybrid IFC There is considerable literature on static analyses aiding IFC execution monitors for different purposes. To boost permissiveness, Le Guernic et al. provide monitors which statically analyze non-taken branches of secret conditionals [Le Guernic et al., 2007, Le Guernic, 2007]. Similarly, Shroff et al. design a monitor which leverages variable dependencies (provided by a type system) when programs branch on secrets [Shroff et al., 2007]. Besides permissiveness, hybrid analyses are used to avoid leaks in dynamic flow-sensitive IFC monitors, where variables change their security levels at runtime based on what data they store [Russo and Sabelfeld, 2010]. Moore and Chong utilizes static analysis to avoid tracking variables which do not impose security violations, thus improving performance on dynamic monitors [Moore and Chong, 2011]. Jif, an IFC-aware compiler for Java programs, supports dynamic labels to classify data based on runtime observations [Zheng and Myers, 2007]. Similar to our work, operations on

labels are modeled at the level of types. In the dynamic part, however, they only allow for runtime checks based on the $\sqsubseteq$ relationship. As in this work, there is some literature which connects dynamic and static analysis at the programming-language level. Disney and Flanagan describe an IFC type-system for a pure λ-calculus which defers cast checks to runtime when they cannot be determined statically [Disney and Flanagan, 2011]. Fennel and Thiemann extend that work to consider references [Fennell and Thiemann, 2013].

Security Libraries Li and Zdancewic's seminal work [Li and Zdancewic, 2006] shows how arrows [Hughes, 2000] can provide IFC without runtime checks as a library in Haskell. Tsai et al. [Tsai et al., 2007] extend Li and Zdancewic's work to support concurrency and data with multiple security labels. Rather than using arrows, Russo et al. [Russo et al., 2008] shows that monads are capable of providing a library which statically enforces IFC. Devriese and Piessens provide a monad transformer to extend imperative-like APIs with support for IFC. Their technique is applied to dynamic, static, and hybrid IFC techniques. Devriese and Piessens' work requires a deep embedding of the target language in order to perform static analysis. In contrast, our approach leverages the type-system features found in Haskell. Jaskelioff and Russo implements a library which dynamically enforces IFC using secure multi-execution (SME) [Jaskelioff and Russo, 2011]—a technique that runs programs multiple times (once per security level) and varies the semantics of inputs and outputs to protect confidentiality. The series of work on LIO can be referred to as the state-of-the-art in dynamic IFC in Haskell [Stefan et al., 2011b, 2012b,a, Buiras et al., 2013, Buiras and Russo, 2013, Buiras et al., 2014].

Programming Languages Combining dynamic and static analysis is not exclusive to IFC research. It has been extensively studied by the programming languages community. We briefly mention some highlights and their relation to this work. Flanagan [Flanagan, 2006] develops the concept of *hybrid type checking* for type systems capable of delaying subtyping checks until runtime. Siek and Taha [Siek and Taha, 2006] coined the term *gradual typing*, which applies when programmers can control the combination of static and dynamic approaches at the programming language level—simultaneously, Hochstadt and Felleisen [Hochstadt and Felleisen, 2006] introduce similar ideas. Due to the *defer* primitive, HLIO can be considered as a simple gradual typing system. Wadler and Findler [Wadler and Findler, 2009] leverage the notion of *blame* from Findler and Felleisen's contracts [Findler and Felleisen, 2002] to explain failure of dynamic type casts (specially for languages with higher-order functions). HLIO is a system which only produces positive blame. Recently, the idea of gradual typing has gained popularity among several programming languages. Typed Scheme [Hochstadt and Felleisen, 2006] and Racket [Takikawa et al., 2012] allow Scheme pro-

grammers to decorate their code with type annotations. Reticulated Python [Vitousek et al., 2014] implements gradual typing, where a type checker is provided in combination with a code-to-code transformation into Python 3. JavaScript has been also a recent target of this kind of systems [Swamy et al., 2014, Rastogi et al., 2015]. Different from these approaches, HLIO does not provide a fully-fledged gradual typing system. On the other hand, it avoids any compiler modification by leveraging Haskell's powerful type system.

9.10 Conclusions and Future Work

We have presented HLIO, a new hybrid IFC enforcement in Haskell that allows programmers to defer static constraints to runtime. This feature is particularly useful, for instance, in production systems—where it is often the case that security labels are not available (or even known) at compile time. Different from other programming languages, GHC's powerful type-system and features allowed us to build HLIO as a simple library, where no runtime or compiler modifications were needed. On formal aspects, we showed that the library satisfies termination-insensitive non-interference for an arbitrary security lattice.

As part of developing HLIO, we have identified an independently useful technique for deferring other forms of static constraints, including ordinary type equalities. In future work, we aim to explore the use of these techniques in languages with similarly expressive type systems, such as dependently typed languages. In addition, we plan to further explore the design and application space of these techniques, and explore their usability in embedded domain-specific languages and code generators.

Acknowledgments

This work was funded by DARPA CRASH under contract #N66001-10-2-4088, and the Swedish research agencies VR and the Barbro Osher Pro Suecia foundation. We thank the anonymous reviewers and Bart van Delft for useful comments and suggestions.

BIBLIOGRAPHY

Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2009.

Robert Atkey. Parameterised notions of computation. In *Proceedings of the 2006 International Conference on Mathematically Structured Functional Programming*, MSFP'06, pages 5–5, Swinton, UK, UK, 2006. British Computer Society. URL `http://dl.acm.org/citation.cfm?id=2228095.2228100`.

T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *APLAS*, volume 8301 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2013.

P. Buiras, D. Stefan, and A. Russo. On flow-sensitive floating-label systems. In *Proc. of 27th IEEE Computer Security Foundations Symp.*, July 2014.

Pablo Buiras and Alejandro Russo. Lazy programs leak secrets. In *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*. Springer Verlag, 2013.

Pablo Buiras, Amit Levy, Deian Stefan, Alejandro Russo, and David Mazières. A library for removing cache-based attacks in concurrent information flow systems. In *Trustworthy Global Computing - 8th International Symposium, TGC 2013*, 2013.

Tim Disney and Cormac Flanagan. Gradual information flow typing. In *Workshop on Script-to-Program Evolution (STOP)*, 2011.

Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364522. URL `http://doi.acm.org/10.1145/2364506.2364522`.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014.

Luminous Fennell and Peter Thiemann. Gradual security typing with references. In *Proceedings of the IEEE 26th Computer Security Foundations Symposium*, CSF '13. IEEE Computer Society, 2013.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. URL `http://doi.acm.org/10.1145/581478.581484`.

Cormac Flanagan. Hybrid type checking. In *Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06. ACM, 2006.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60. USENIX, 2012.

Michael Hicks, Gavin Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. Polymonadic programming. In *In Proceedings of the Mathematically Structured Functional Programming (MSFP) 2014*, 2014.

Sam T. Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006.

J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in Haskell. In *Perspectives of Systems Informatics - 8th International Andrei Ershov Memorial Conference, PSI*, 2011.

Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*. IEEE Computer Society, 2007.

Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Proc. of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06. Springer-Verlag, 2007.

P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW'06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

Erik Meijer and Peter Drayton. Static Typing Where Possible, Dynamic Typing When Needed. *Revival of Dynamic Languages*, 2005. URL `http://research.microsoft.com/~emeijer/Papers/RDL04Meijer.pdf`.

Scott Moore and Stephen Chong. Static analysis for efficient hybrid information-flow control. In *Proc. of the 24th IEEE Computer Security Foundations Symposium*. IEEE Press, June 2011.

Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.

Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and Separation in Hoare Type Theory. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 62–73, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. doi: 10.1145/1159803.1159812. URL `http://doi.acm.org/10.1145/1159803.1159812`.

Dominic Orchard and Tom Schrijvers. Haskell type constraints unleashed. In *Lecture Notes in Computer Science,*, pages 56–71. Springer, 2010. doi: 10.1007/978-3-642-12251-4_{}6. URL `https://lirias.kuleuven.be/handle/123456789/259608`.

Aseem Rastogi, Nikhil Swamy, Cedric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe and efficient gradual typing for typescript. In *In Proc. of the ACM Conference on Principles of Programming Languages (POPL) 2015*, January 2015.

Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. of the 2010 23rd IEEE Computer Security Foundations Symp.*, CSF '10, pages 186–199. IEEE Computer Society, 2010.

Alejandro Russo, Koen Claessen, and John Hughes. A library for lightweight information-flow security in Haskell. In *Haskell '08: Proc. of the first ACM SIGPLAN symposium on Haskell*, pages 13–24, 2008.

A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, Lecture Notes in Computer Science (LNCS). Springer Verlag, June 2009.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic Dependency Monitoring to Secure Information Flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07. IEEE Computer Society, 2007.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proc. of Scheme and functional programming workshop*. Technical Report. University of Chicago, 2006.

V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *NordSec 2011*, LNCS. Springer, October 2011a.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, pages 95–106. ACM SIGPLAN, September 2011b.

Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of 17th ACM SIGPLAN International Conference on Functional Programming*, Sep. 2012a.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012b.

Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual Typing Embedded Securely in JavaScript. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014.

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12. ACM, 2012.

David Terei, Simon Marlow, Simon Peyton Jones, , and David Mazières. Safe Haskell. In *Proceedings of the 5th Symposium on Haskell*, September 2012.

Ta-chung Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Computer Security Foundations Symp., 2007. CSF '07. 20th IEEE*, pages 187–202, July 2007.

Steve VanDeBogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. on Computer Systems*, 25(4):11:1–43, December 2007. A version appeared in *Proc. of the 20th ACM Symp. on Operating System Principles*, 2005.

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. of the 10th ACM Symposium on Dynamic Languages*, DLS '14. ACM, 2014.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.

Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *Proceedings*

of the 17th ACM SIGPLAN International Conference on Functional Programming, ICFP '12, pages 341–352, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364554. URL http://doi.acm.org/10.1145/2364527.2364554.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proc. of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09. Springer-Verlag, 2009.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. doi: 10.1145/2103786.2103795. URL http://doi.acm.org/10.1145/2103786.2103795.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow. *International Journal of Information Security*, 6(2–3), 2007.

A Secure *wgetLIO*

In the web application security model, scripts from one page cannot access data from a different web page unless the two pages have the same *origin*—a policy known as the *same origin policy*[7] (SOP). To illustrate how this policy can be enforced we introduce a more complex security lattice [Myers and Liskov, 1997, Stefan et al., 2011a, Broberg et al., 2013], that also includes *URIAuth* values, representing origins. (we omit ⊓ for brevity below)

```
data Label = L | H | Orig URIAuth
instance Lattice Label where
  ⊔ _          H          = H
  ⊔ H          _          = H
  ⊔ L          (Orig x)   = Orig x
  ⊔ (Orig x)   L          = Orig x
  ⊔ (Orig _)   (Orig _)   = H
  ⊔ L          L          = L

  ⊑ H          L          = False
  ⊑ H          (Orig _)   = False
  ⊑ (Orig o)   (Orig o')  = eqAuth o o'
  ⊑ (Orig _)   L          = False
  ⊑ _          _          = True
eqAuth o o' = ...   – equality on URIAuth
```

Using this lattice it is possible to implement a secure version of *wget* that accepts a string representing an HTTP URL, and returns the content labeled with the *URIAuth* origin of that URL:

```
wgetLIO :: String → LIO (Labeled String)
wgetLIO http = case parseURIAuth http of
    Just auth → toLabeled (Orig auth)
                          (liftIO (wget http [] []))
    Nothing → lerr "Invalid origin"
  where
    lerr msg = label L (error msg)
```

Function *parseURIAuth* parses an URL and returns possibly an URI authority[8]. From the security point of view, function *wget* produces both a read and a write effect. The read effect is on the *LIO* computation, which can observed the content of the requested URL—consequently, the result is wrapped into a labeled value to protect its confidentiality, which is returned by *toLabeled*. The write effect, on the other hand, is done on the reached host, which knows when a request is performed by *LIO* computations. We

[7] `https://www.w3.org/Security/wiki/Same_Origin_Policy`
[8] This function is provided by the Hackage package `network-uri`

should then avoid information from an origin to influence URL requests performed to another one—this could effectively encode data leaks. This is also captured in *toLabeled*, which checks $\ell_{\mathsf{cur}} \sqsubseteq \mathit{auth}$, before creating the labeled term and executing *wget*. In that manner, for instance, requests can fetch URLs from every origin provided that they are not *unlabel*, i.e., the current label being to L. Once information is read from a single origin, the current label gets tainted and *wgetLIO* subsequently allows to only fetch URLs from that same origin—effectively enforcing SOP.

B Formal Results

Lemma 1 (Simulation between HLIO and LIO terms)
Given that $(t : \mathit{HLIO}\ c\ \ell_i\ \ell_o\ \tau) \Downarrow v$, *then* $\langle \ell_i \mid \mathit{guards}\ c \gg \mathit{interp}\ t\rangle \sim \langle \ell_i \mid \mathit{toLIO}\ t\rangle$.

Proof sketch 2 *Induction on the derivation of* $\Gamma \vdash t : \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a$. *We briefly sketch how the security-relevant cases are handled.*

Case $t = \mathit{unlabel}\ t'$. *Then we know* $\Gamma \vdash t' : \mathit{SLabeled}\ \ell\ a$. *This command has no constraints* ($c = ()$) *and the dynamic version makes no checks, so* $\langle \ell_i \mid \mathit{interp}\ t\rangle \sim \langle \ell_i \mid \mathit{toLIO}\ t\rangle$ *trivially.*

Case $t = \mathit{label}\ t_1\ t_2$. *Then it must emit a constraint* c, *which trivially matches the constraint in the type. The computation* $\mathit{toLIO}\ (\mathit{label}\ t_1\ t_2)$ *reduces to* $\mathit{label}\ t_1\ (\mathit{toLIO}\ t_2)$, *which further reduces to* $\mathit{return}\ (\mathit{SLabeled}^{\mathit{TCB}}\ \ell\ (\mathit{toLIO}\ t_2))$, *where* $t_1 \rightsquigarrow^* \ell$. *This matches* $\mathit{interp}\ (\mathit{label}\ t_1\ t_2)$.

Case $t = m \gg\!= f$. *Then we know* $\Gamma \vdash m : \mathit{HLIO}\ c\ \ell_i\ \ell\ a$ *and* $\Gamma \vdash f : a \to \mathit{HLIO}\ c'\ \ell\ \ell_o\ b$. *By induction hypothesis, we have that* $\langle \ell_i \mid \mathit{guards}\ c \gg \mathit{interp}\ m\rangle \sim \langle \ell_i \mid \mathit{toLIO}\ m\rangle$ *and if* $\langle \ell_i \mid m\rangle \longrightarrow *\langle \ell \mid \mathit{LIO}^{\mathit{TCB}}\ a\rangle$, *then* $\langle \ell_i \mid \mathit{guards}\ c' \gg \mathit{interp}\ (f\ a)\rangle \sim \langle \ell_i \mid \mathit{toLIO}\ (f\ a)\rangle$. *We have to prove that* $\langle \ell_i \mid \mathit{guards}\ (c, c') \gg \mathit{interp}\ m \gg\!= \mathit{interp}.f\rangle \sim \langle \ell_i \mid \mathit{toLIO}\ (m \gg\!= f)\rangle$.

We can rewrite $\mathit{guards}\ (c, c') \gg \mathit{interp}\ m \gg\!= \mathit{interp}.f$ *as* $\mathit{guards}\ c \gg \mathit{interp}\ m \gg\!= \lambda a \to \mathit{guards}\ c' \gg \mathit{interp}\ (f\ a)$ *without changing the trace of checks performed or the normal form. Therefore, we can combine this with the previous facts to show that* $\langle \ell_i \mid \mathit{guards}\ (c, c') \gg \mathit{interp}\ m \gg\!= \mathit{interp}.f\rangle \sim \langle \ell_i \mid \mathit{toLIO}\ m \gg\!= \lambda a \to \mathit{toLIO}\ (f\ a)\rangle = \langle \ell_i \mid \mathit{toLIO}\ (m \gg\!= f)\rangle$.

Case $t = \mathit{defer}\ m$, *with* $c = ()$. *Then we know* $\Gamma \vdash m : \mathit{HLIO}\ c'\ \ell_i\ \ell_o\ a$ *for some* c', *and by induction hypothesis,* $\langle \ell_i \mid \mathit{guards}\ c' \gg \mathit{interp}\ m\rangle \sim \langle \ell_i \mid \mathit{toLIO}\ m\rangle$. *We have to prove that* $\langle \ell_i \mid \mathit{guards}\ () \gg \mathit{interp}\ (\mathit{defer}\ m)\rangle \sim \langle \ell_i \mid \mathit{toLIO}\ (\mathit{defer}\ m)\rangle$, *but applying the definitions of* *interp* *and* *toLIO* *we have* $\langle \ell_i \mid \mathit{guards}\ () \gg \mathit{guards}\ c' \gg \mathit{interp}\ m\rangle \sim \langle \ell_i \mid \mathit{toLIO}\ m\rangle$, *which follows from the induction hypothesis.*

Case $t = \mathit{toLabeled}\ m$. *Then we know*

$\Gamma \vdash m : \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a$

and we have to prove that

$\langle \ell_i \mid guard\ c \gg toLabeled\ \ell_o\ (interp\ m)\rangle \sim$
$\langle \ell_i \mid toLabeled\ \ell_o\ (toLIO\ m)\rangle$.

This boils down to proving that

$$\langle \ell_i \mid guard\ c \gg interp\ m\rangle \sim \langle \ell_i \mid toLIO\ m\rangle,$$

which follows by induction hypothesis.

Theorem 1 (Termination-insensitive noninterference)
Given HLIO terms t_1 and t_2 with no constructors $\cdot^{TCB}$ such that

- *Constrains c_1 and c_2 hold*
- $(t_1 : HLIO\ c_1\ \ell_1\ \ell_2\ \tau) \Downarrow v_1$,
- $(t_2 : HLIO\ c_2\ \ell_i\ \ell_o\ \tau') \Downarrow v_2$, *and*
- $\langle \ell_i \mid toLIO\ t_1\rangle \approx_l \langle \ell_i \mid toLIO\ t_2\rangle$

it holds that $\langle \ell_o \mid toLIO\ v_1\rangle \approx_l \langle \ell_o \mid toLIO\ v_2\rangle$.

By applying Definition 2, we have that

1. $\langle \ell_i \mid interp\ t_1\rangle \xrightarrow{c_{t_1}}{}^{*} \langle \ell_o \mid toLIO\ v_1\rangle$
2. $\langle \ell_i \mid interp\ t_2\rangle \xrightarrow{c_{t_2}}{}^{*} \langle \ell_o \mid toLIO\ v_2\rangle$

By applying Lemma 1 to the hypothesis on t_1 and t_2, we obtain that

1. $guard\ c_1 \gg interp\ t_1 \sim toLIO\ t_1$
2. $guard\ c_2 \gg interp\ t_2 \sim toLIO\ t_2$

This implies that there exists a c, c', X, and Y such that

5. $\langle \ell_i \mid guard\ c_1 \gg interp\ t_1\rangle \xrightarrow{c}{}^{*} X$
6. $\langle \ell_i \mid toLIO\ t_1\rangle \xrightarrow{c}{}^{*} X$
7. $\langle \ell_i \mid guard\ c_2 \gg interp\ t_2\rangle \xrightarrow{c'}{}^{*} Y$
8. $\langle \ell_i \mid toLIO\ t_2\rangle \xrightarrow{c'}{}^{*} Y$

By simple reduction rules, and the hypothesis that c_1 and c_2 hold, we know that

9. $\langle \ell_i \mid guard\ c_1 \gg interp\ t_1\rangle \xrightarrow{c_1}{}^{*} \langle \ell_i \mid interp\ t_1\rangle$
10. $\langle \ell_i \mid guard\ c_2 \gg interp\ t_2\rangle \xrightarrow{c_2}{}^{*} \langle \ell_i \mid interp\ t_2\rangle$

Confining these (9) and (10) with (1) and (2), and since we are in a deterministic system, we obtain that there

11. $X \equiv \langle \ell_o \mid toLIO\ v_1\rangle$
12. $Y \equiv \langle \ell_o \mid toLIO\ v_2\rangle$
13. $c = c_1 \cup c_{t_1}$

14. $c' = c_2 \cup c_{t_2}$

So, by using (11)–(14), we rewrite (5)–(8) as follows.

15. $\langle \ell_i \mid \mathit{guard}\ c_1 \gg \mathit{interp}\ t_1 \rangle \overset{c_1 \cup c_{t_1}}{\longrightarrow}^{*} \langle \ell_o \mid \mathit{toLIO}\ v_1 \rangle$
16. $\langle \ell_i \mid \mathit{toLIO}\ t_1 \rangle \overset{c_1 \cup c_{t_1}}{\longrightarrow}^{*} \langle \ell_o \mid \mathit{toLIO}\ v_1 \rangle$
17. $\langle \ell_i \mid \mathit{guard}\ c_2 \gg \mathit{interp}\ t_2 \rangle \overset{c_2 \cup c_{t_2}}{\longrightarrow}^{*} \langle \ell_o \mid \mathit{toLIO}\ v_2 \rangle$
18. $\langle \ell_i \mid \mathit{toLIO}\ t_2 \rangle \overset{c_2 \cup c_{t_2}}{\longrightarrow}^{*} \langle \ell_o \mid \mathit{toLIO}\ v_2 \rangle$

19. We have that $\langle \ell_i \mid \mathit{toLIO}\ t_1 \rangle \approx_l \langle \ell_i \mid \mathit{toLIO}\ t_2 \rangle$ by hypothesis, and by definition of *toLIO* and hypothesis, we know $\mathit{toLIO}\ t_1$ and $\mathit{toLIO}\ t_2$ do not include constructors $\cdot^{\mathsf{TCB}}$.

At this point, we can apply the non-interference theorem in [Stefan et al., 2012b] to $\langle \ell_i \mid \mathit{toLIO}\ t_1 \rangle \approx_l \langle \ell_i \mid \mathit{toLIO}\ t_2 \rangle$, (16), and (18), thus obtaining that $\langle \ell_o \mid \mathit{toLIO}\ v_1 \rangle \approx_l \langle \ell_o \mid \mathit{toLIO}\ v_2 \rangle$.

C Additional figures

```
instance (Deferrable c1, Deferrable c2) ⇒ Deferrable (c1, c2) where
  defer p f = deferPair p f
deferPair :: forall a c1 c2.(Deferrable c1, Deferrable c2) ⇒
  Proxy (c1, c2) → ((c1, c2) ⇒ a) → a
deferPair (_ :: Proxy (c1, c2)) f =
  let f1 :: c1 ⇒ (c2 ⇒ a)
      f1 = f   – Cast from (c1, c2) => a to c1 => (c2 => a)
               – Not possible to do it inside the type-class
      f2 :: c2 ⇒ a
      f2 = defer (⊥ :: Proxy c1) f1
  in defer (⊥ :: Proxy c2) f2
```

Fig. 10. Instance of *Deferrable* for pairs of constraints.

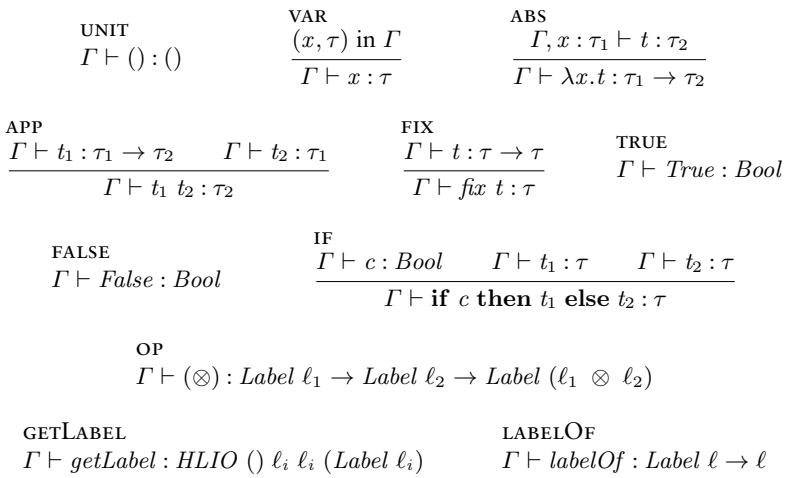

Fig. 11. Type system for HLIO (missing rules).