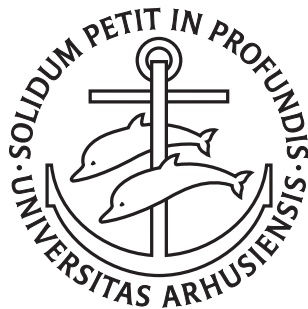

Enforcement of Timing-Sensitive Security Policies in Runtime Systems

Mathias Vorreiter Pedersen

PhD Dissertation



Department of Computer Science
Aarhus University
Denmark

Enforcement of Timing-Sensitive Security Policies in Runtime Systems

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Mathias Vorreiter Pedersen
July 31, 2019

Abstract

The work presented in this dissertation focuses on applying information-flow control (IFC) techniques to the construction of secure runtime systems that do not leak sensitive information through their timing behavior. The dissertation also presents applications of IFC to the development of secure programs that require dynamic authorization policies.

Chapter 2 presents a series of timing channel attacks on automatic memory management, and a calculus in which the attacks can be formally studied. We then study an enforcement technique for preventing the discovered timing leaks, and prove the enforcement sound using the Coq proof assistant. The enforcement combines a security type system with a secure semantics for garbage collection.

Chapter 3 studies the feasibility of implementing the secure garbage collection semantics, and other runtime related activities, like secure thread scheduling. We present a language for implementing such secure runtime components, along with a type system for which we prove that any well-typed program is secure. Thus, runtime system activities can be implemented as a program in the language itself, and the well-typing of the program guarantees that the runtime activity does not leak information through timing.

Finally, Chapter 4 studies authorization policies using the Flow-Limited Authorization Model (FLAM), and shows how developers can program with dynamic and decentralized authorization policies while guaranteeing security. Previous work applying FLAM to the development of secure programs has focused on static and fine-grained enforcement techniques, but we argue that FLAM is better suited for a dynamic enforcement of coarse-grained information-flow control.

Resumé

Arbejdet, der præsenteres i denne afhandling, omhandler brugen af IFC-teknikker (information-flow control-teknikker) til konstruktionen af sikre runtime-systemer, der ikke lækker følsom information gennem deres kørselstid. Afhandlingen præsenterer også anvendelser af IFC til udviklingen af sikre programmer, der kræver dynamiske autoriseringspolitikker.

Den første publikation præsenterer en række kørselstidsangreb, der benytter kørselstiden af automatisk hukommelsehåndtering til at lække følsom information. Vi præsenterer også en kalkule, hvor disse angreb kan blive studeret formelt, og bruger denne kalkule til at undersøge en håndhævelsesteknik, som bruges til at forhindre disse nye kørselstidsangreb, og beviser denne håndhævelsesteknik sund ved brug af Coq-bevisassistenten. Håndhævelsesteknikken kombinerer et sikkerhedstypesystem og en sikker semantik for hukommelsesindsamling.

Den anden publikation undersøger, hvorvidt det er muligt at implementere denne sikre semantik for hukommelsesindsamling, og andre relaterede aktiviteter som f.eks. sikker trådschedulering. Vi præsenterer et programmeringssprog til implementation af sådanne sikre runtime-komponenter sammen med et typesystem, og vi beviser at ethvert program, som overholder typesystemet, er sikkert. På denne måde kan runtime-systemaktiviteter blive implementeret i sproget selv, og fordi programmet overholder typesystemet, er det garanteret, at aktiviteten ikke lækker følsom information gennem programmets kørselstid.

Den tredje publikation studerer autoriseringspolitikker ved brug af the Flow-Limited Authorization Model (FLAM). Vi viser hvordan udviklere kan programmere med dynamiske og decentraliserede autoriseringspolitikker og samtidigt garantere, at det resulterende system er sikkert. Tidligere arbejde, som anvender FLAM til udviklingen af sikre programmer, har fokuseret på statiske og finkornede håndhævelsesteknikker, men vi argumenterer for, at FLAM er bedre egnet til en dynamisk håndhævelsesteknik med grovkornet IFC.

Acknowledgments

While the cover of this dissertation only has my name below the title, none of this could have existed was it not for a large group of people.

First, I would like to thank Aslan Askarov for his supervision and helpful advice regarding everything from information-flow control to “how to have a life” as a PhD student. Thank you for teaching me the value of practicing technical presentations again and again ... and then again. I would also like to thank Anders Møller for initially offering me a PhD position, and for his patience while I was struggling to find my research area.

I also want to thank the entire Turing-2 hallway at the Department of Computer Science at Aarhus University for the endless amount of runs to the coffee lounge, and all the other fun things we have done together.

This acknowledgment would be incomplete without a special thanks to Professor Stephen Chong, and the rest of the MD309 gang at Harvard University, for giving me an amazing place to work during my research stay abroad. I will never forget that brief, but very formative, period of my life.

Of course, I want to thank you, Mom, for always welcoming me with open arms when I return home to Vejle. Even though you have long given up understanding what I have been doing during my PhD studies, you have always supported me whenever I needed it.

And finally, I want to thank you, Ida. I would have never reached the end of this journey without you. You have helped me through the most difficult times, and you have made the best times even better. I cannot thank you enough for everything you have done for me, and I hope to be able to support you as much as you have supported me.

*Mathias Vorreiter Pedersen,
Aarhus, July 31, 2019.*

Contents

Abstract	i
Resumé	iii
Acknowledgments	v
I Overview	1
1 Introduction	3
1.1 Background	4
1.2 Information-Flow Control	5
1.3 Enforcement Techniques	8
1.4 Information-Flow Control in the 21 st Century	12
1.5 Covert Channels	18
1.6 Outline of the Dissertation	25
II Publications	29
2 From Trash to Treasure: Timing-Sensitive Garbage Collection	31
2.1 Introduction	31
2.2 Background	34
2.3 Attacking JVM and V8	36
2.4 Language	43
2.5 Semantics for Secure Garbage Collection	46
2.6 Type System	55
2.7 Security Guarantees	60
2.8 Connections to Real Time Garbage Collection	66
2.9 Related Work	67
2.10 Conclusion	68
3 Static Enforcement of Security in Runtime Systems	69
3.1 Introduction	69

3.2	Programming in Zee	71
3.3	Language	76
3.4	Security Guarantees	89
3.5	Case Studies	93
3.6	Implementation	101
3.7	Related Work	101
3.8	Conclusion	103
4	Programming with Flow-Limited Authorization: Coarser is Better	105
4.1	Introduction	106
4.2	Background on FLAM and LIO	108
4.3	Introduction to Flamio by Example	111
4.4	A Calculus for Flamio	115
4.5	Security Guarantees	127
4.6	Implementation and Case Studies	133
4.7	Related Work	137
4.8	Conclusion	139
4.9	Acknowledgements	139
5	Conclusion and Future Work	141
III	Appendix	143
A	Static Enforcement of Security in Runtime Systems	145
A.1	Syntax of Zee	145
A.2	Semantics of Zee	146
A.3	A Type System for Zee	147
A.4	Semantic Well-Formedness	152
A.5	Attacker Model	154
A.6	Required Properties of the Instantiation Languages	160
A.7	Noninterference for Zee	161
B	Programming with Flow-Limited Authorization: Coarser is Better	165
B.1	Syntax	165
B.2	Local Semantics	166
B.3	Semantics of Authorization	169
B.4	Global Semantics	170
B.5	Type System	171
B.6	Attacker Model	174
B.7	Noninterference for Flamio	175
	Bibliography	185

Part I

Overview

Chapter 1

Introduction

There is no arguing that security is important in today's society: we put more and more information on the Internet, and rely on modern cryptographic tools such as encryption, or system-level enforcement techniques such as access control, to ensure that sensitive information does not leak to unintended principals. However, these techniques are unable to enforce the complex security policies demanded by today's applications.

As a concrete example, consider a device that most people nowadays carry in their pocket: their smartphone. This device contains a variety of sensitive information: your contact list contains personal phone numbers of your friends and family, each text message in your text history includes private communication only intended to be shared with one other participant, and your photo album might contain private images not intended to be shared with anyone.

On the other hand, most smartphones today come with the option of downloading additional applications onto the device. While very useful, this feature can be detrimental to the overall security guarantees of the system: untrusted applications often need access to personal information to fulfill their required tasks. For instance, a text messaging application needs to access the contact list, and in order to send and receive images, the application also needs to access the photo album on the device.

In this situation neither encryption nor access control provide *end-to-end security guarantees* [111]: even though a text message sent from principal A to principal B is encrypted during transmission, nothing prevents B from releasing A's confidential message once it has been decrypted after transmission.

Similarly, access control fails to provide end-to-end security guarantees: once an access control mechanism has concluded that the text message ap-

plication can access the photo album of the device, the application has full control over the propagation of the information. Thus, neither solution protects the information throughout the whole “pipeline”.

Information-flow control (IFC) solves these problems, and provides end-to-end security guarantees by not only controlling access to information, but also the further propagation of the information. This is easier said than done, however: there are a multitude of ways an adversarial application can transmit information using *covert channels*, unintended communication channels. Common covert channels include timing channels, power channels, and even electromagnetic channels.

This dissertation studies covert channels arising in runtime systems of modern programming languages, and how such channels can be closed. Runtime systems sit between the programming language and the underlying operating system, and implement part of the execution environment in which programs are run. For instance, most high-level general-purpose programming languages today provide automatic memory management (i.e., a garbage collector), relieving the developer from the mental task of ensuring that allocated memory is freed exactly once.

Such a feature is great for mitigating a large class of bugs but can also serve as a covert channel for transmitting information by carefully changing the memory layout such that the automatic memory management procedure leaks information through its timing behavior. Likewise, other components of the runtime system such as thread scheduling, can be exploited to leak information through timing channels.

The dissertation also studies complex label models for specifying security policies, and how developers can take advantage of such models for specifying confidentiality and integrity requirements on data during program execution.

1.1 Background

This section provides some historical background on the topic of information-flow security, its central concepts, and modern enforcement techniques. An overview of the study of covert channels from a programming language point of view is then provided, and finally the section ends with an outline of the rest of the dissertation.

1.2 Information-Flow Control

Having seen how traditional security mechanisms fail to provide satisfactory end-to-end guarantees, a demonstration of how information-flow control achieves these goals is now presented.

1.2.1 Confining Program Executions

The problem of controlling the propagation of information by an executing program was first studied by Lampson [66]. Lampson argued for the importance of being able to *confine* a program P such that it would be impossible for P to communicate with the outside world during its execution. This would allow a host system to invoke untrusted code without the fear of unintended information leakage to the outside world, even in the case where the host would allow P access to the host's sensitive information.

1.2.2 Noninterference

The problems described by Lampson leads one to ask the question of which actions a confined program should be allowed to perform. In other words, what is the security policy that we expect to be enforced by the confinement mechanism? This problem was first formulated by Goguen and Meseguer [46] who defined much of the terminology of today, including the notion of a security policy, a security model, and what it means for one group of users to be *noninterfering* with respect to another group of users.

According to Goguen and Meseguer, a security policy is the security requirements for a given system. The system is then represented by a security model using some suitable high-level abstract machine description.

Of all contributions put forth by Goguen and Meseguer, the most important is the notion of noninterference that serves as the key security property for information-flow control. Intuitively, a group *Secret* is noninterfering with another group *Public* if whatever *Secret* does cannot influence what *Public* observes. This is represented by the diagram in Figure 1.1. All actions performed by the group *Public* can be observed by *Secret*, but only the group *Secret* can observe the actions performed by *Secret*.

Yet another, more modern, take on noninterference is provided by the theory of *hyperproperties* developed by Clarkson and Schneider [31] that showed that noninterference (along with many related security definitions like observational determinism [138]) falls into this category. Intuitively, while a property P is a set of traces for which P holds, a hyperproperty \mathbf{P} is a *set of sets of traces* for which \mathbf{P} holds. As noninterference is an example of a hyper-

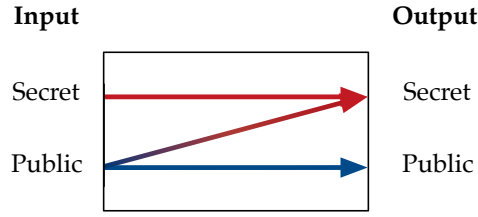


Figure 1.1: Graphical representation of noninterference.

property,¹ it is always stated as a hyperproperty involving two traces.² This point is revisited later in Section 1.3.

1.2.3 Lattices as Models for Information Flow

Denning and Denning [34] demonstrated that information flow could be described using a lattice structure \mathcal{SC} (for Security Classes), and derived the axioms that form a lattice from first principles. As an introduction to language-based information-flow control this section replays the arguments given by Denning and Denning.

The elements of \mathcal{SC} are classes describing disjoint classes of information, and a partial ordering $\sqsubseteq \subseteq \mathcal{SC} \times \mathcal{SC}$ such that given $A, B \in \mathcal{SC}$ it holds that $A \sqsubseteq B$ (pronounced *A flows to B*, or *A may be relabeled as B*) if information classified as A may flow into class B . We write $a \in A$ if variable a has security class A . For the remaining parts of this section we assume a simple programming language with variables x, y, z, \dots belonging to security classes X, Y, Z, \dots . Finally, a program is a sequence of assignment commands of the form $x := e$, where an expression e can be either variables, constants $n \in \mathbb{N}$ or binary operations $f(e_1, e_2)$ of expressions e_1, e_2 .

Given variables $a \in A$ and $b \in B$, an assignment command $b := a$ is secure if $A \sqsubseteq B$. We can now argue from first principles why \sqsubseteq should be a partial order on \mathcal{SC} . First, \sqsubseteq should be reflexive because the command $a := a$ should always be considered secure. Similarly, if commands $b := a$ and $c := b$ are both secure (i.e., $A \sqsubseteq B$ and $B \sqsubseteq C$), it should also be the case that command $c := a$ is secure (i.e., $A \sqsubseteq C$), giving rise to the natural condition of transitivity. Lastly, antisymmetry arises from a pragmatic point of view: if both $A \sqsubseteq B$ and $B \sqsubseteq A$ are true, elements of A and B may flow freely between the two classes, and there is little reason to distinguish them.

¹Alternative formulations of noninterference that transforms secure information-flow to trace properties also exist (see [18, 103]).

²Or in the case of nonmalleable information-flow control [28]: as a hyperproperty on four traces.

Denning and Denning also argue for an associative *class-combining* operator $\sqcup : \mathcal{SC} \times \mathcal{SC} \rightarrow \mathcal{SC}$ (pronounced *join*) that, given two classes A and B , describes the class in which any result of a binary operation depending on classes A and B belongs. That is, if $a \in A$ and $b \in B$ it holds that $f(a, b) \in A \sqcup B$ for any binary operation f . They argue that $A \sqcup B$ should be the *least upper bound* of A and B , as the model otherwise would have “semantic absurdities”. In particular, it must hold that

1. $A \sqsubseteq A \sqcup B$ and $B \sqsubseteq A \sqcup B$.
2. If $A \sqsubseteq C$ and $B \sqsubseteq C$ then $A \sqcup B \sqsubseteq C$.

Condition (1) states that the join upper bounds its arguments, and (2) states that the join must be minimal.

If (1) is not satisfied for some $A, B \in \mathcal{SC}$ there are some operands $a \in A$ and $b \in B$ such that it is not allowed for a to influence the result of the binary operation $f(a, b)$. Even worse is the situation when f implements addition on natural numbers \mathbb{N} . In this case, it could be that $c := a + b$ can be considered secure (i.e., if $A \sqcup B \sqsubseteq C$), but $c := a$ be considered insecure (i.e., if $A \not\sqsubseteq C$). But the relation $A \sqcup B \sqsubseteq C$ should mean that the command $c := a + b$ is secure for any values of a and b , so in particular the execution of the command should be secure when $b = 0$, in which case the command is equivalent to $c := a$, which was considered insecure. Requiring (1) ensures that $A \sqcup B \sqsubseteq C$ implies $A \sqsubseteq C$, eliminating this absurd scenario.

To see why (2) is a natural condition, consider three classes A , B and C , and pick $a \in A$, $b \in B$ and $c, c_1, c_2 \in C$. If $A \sqsubseteq C$ and $B \sqsubseteq C$ the following assignments are considered secure

1. $c_1 := a$
2. $c_2 := b$

And as $C \sqsubseteq C$ the assignment $c := c_1 + c_2$ is also secure, and so the combined program $c_1 := a; c_2 := b; c := c_1 + c_2$ is also secure. But this program is equivalent to $c := a + b$, which is only secure if $A \sqcup B \sqsubseteq C$. So we should require that $A \sqsubseteq C$, $B \sqsubseteq C$ implies $A \sqcup B \sqsubseteq C$, which is the exact property stated in (2).

It is for these reasons that we consider an arbitrary join-semilattice \mathcal{L} in Chapters 2 and 3, and enrich the structure considered in Chapter 4 so that it forms a lattice. For many examples we will use the simple two-point lattice consisting of the security classes $\mathcal{L}_2 = \{\mathbf{L}, \mathbf{H}\}$ with the ordering $\mathbf{L} \sqsubseteq \mathbf{H}$ for all

$\ell \in \mathcal{L}_2$ and $\mathbf{H} \sqsubseteq \mathbf{H}$, and the least upper bound \sqcup given as $\mathbf{L} \sqcup \ell = \ell \sqcup \mathbf{L} = \ell$, and $\mathbf{H} \sqcup \ell = \ell \sqcup \mathbf{H} = \mathbf{H}$ for all $\ell \in \mathcal{L}_2$.

Having considered how a lattice \mathcal{L} helps formalize information flows in a simple language with only assignment operations, we now turn to much more sinister kinds of information flows and demonstrate how the theory developed by Denning and Denning scales to a larger set of language features.

1.2.4 Explicit Versus Implicit Information Flows

For this section we assume a mapping Γ from variable names to security levels and define a function $\llbracket \cdot \rrbracket_\Gamma : e \rightarrow \mathcal{L}$ from expressions to security elements in \mathcal{L} . It is defined inductively as

$$\llbracket x \rrbracket_\Gamma = \Gamma(x) \quad \llbracket n \rrbracket_\Gamma = \perp \quad \llbracket f(e_1, e_2) \rrbracket_\Gamma = \llbracket e_1 \rrbracket_\Gamma \sqcup \llbracket e_2 \rrbracket_\Gamma$$

where $\perp \in \mathcal{L}$ represents the least element in \mathcal{L} . That is, for all $\ell \in \mathcal{L}$ it holds that $\perp \sqsubseteq \ell$, and $\perp \sqcup \ell = \ell \sqcup \perp = \ell$.

In the simple language setting that has been discussed so far, there is only one type of information flow which I denote as a *direct* flow [34]. A direct flow occurs as a result of executing an assignment command $x := e$, where information classified as $\llbracket e \rrbracket_\Gamma$ flows to $\Gamma(x)$. However, there is a much more subtle kind of information flow that happens when we add a conditional command to the language. Consider two extensions to the language: first we add a conditional command if e then c_1 else c_2 that executes c_1 if e evaluates to 0, and executes c_2 otherwise. Second, we add a no-op command skip that performs no observable side-effects.

Consider now the program if x then $y := e$ else skip. This program has a direct flow from $\llbracket e \rrbracket_\Gamma$ to $\Gamma(y)$ if x is non-zero, but it also exhibits an *implicit flow* from $\Gamma(x)$ to $\Gamma(y)$ which does not depend on the value of x . The implicit flow from $\Gamma(x)$ to $\Gamma(y)$ occurs since the choice of executing, or *not* executing, a statement that causes the explicit flow from $\llbracket e \rrbracket_\Gamma$ to $\Gamma(y)$ depends on a value from security class $\Gamma(x)$. We now turn to the question of how information flows, both explicit and implicit, can be restricted such that only allowed flows occur during execution.

1.3 Enforcement Techniques

Having developed a lattice-model of information flow, this section now turns to the question of how information-flow control can be *enforced*. More precisely, given a lattice \mathcal{L} , what mechanisms can be used to enforce the

property that only flows allowed by \mathcal{L} occur at runtime? Coarsely, the mechanisms can be split into two categories: static enforcement techniques and dynamic enforcement techniques.

1.3.1 Static Enforcement Mechanisms

Volpano et al. [130] introduced type-based analyses for information flow, paving the way for many type systems for information-flow control [8, 27, 28, 69, 81, 83, 85, 92, 93, 103, 104, 114, 124, 145]. The well-known saying by Milner [74] that “well-typed programs can’t go wrong” can be restated as “well-typed programs don’t interfere”, expressing that, if a program is well-typed in the type system of Volpano et al. it satisfies noninterference.

In this simple setting, a typing derivation $\Gamma \vdash e : \ell$ states that e depends on information classified by the security class ℓ or below (wrt. the partial order on security classes \sqsubseteq). Dually, the judgment $\Gamma \vdash c : \ell$ states that c only modifies data at security class ℓ or *higher*. That is, if $\Gamma \vdash c : \mathbf{H}$ it follows that c will never execute an assignment to a variable x such that $\Gamma(x) = \mathbf{L}$. Following the terminology of Lampson [66], this is known as the confinement property, or colloquially the *high-pc* lemma.

The big-step style semantics of Volpano et al. of the form $m \vdash c \Rightarrow m'$ can be reformulated as a small-step style semantics on configurations of the form $\langle c, m \rangle \rightarrow \langle c', m' \rangle$, expressing that command c in memory m performs a single step to command c' and modifies the memory to m' . Finally, we write the reflexive, transitive closure of \rightarrow as \rightarrow^* .

As discussed in the previous section, noninterference is a hypersafety property over two traces. This becomes especially clear in the type soundness theorem obtained by Volpano et al. To formulate their result precisely, a security class-indexed equivalence relation is defined on memories. Specifically, $\mathcal{A} \in \mathcal{L}$ denotes the level of the attacker, i.e., an upper bound on the security classes the attacker can observe. The relation $=_{\mathcal{A}} \subseteq \mathcal{M} \times \mathcal{M}$ defines which memories an attacker \mathcal{A} considers equivalent:

Definition 1.3.1 (Memory \mathcal{A} -equivalence). *Two memories m_1 and m_2 are \mathcal{A} -equivalent, written $m_1 =_{\mathcal{A}} m_2$, if*

$$\begin{aligned} \text{dom}(m_1) &= \text{dom}(m_2) = \text{dom}(\Gamma) \wedge \\ \forall x \in \text{dom}(\Gamma). \Gamma(x) &\sqsubseteq \mathcal{A} \implies m_1(x) = m_2(x) \end{aligned}$$

The type soundness theorem of Volpano et al. [130] can now be stated as follows.

Theorem 1.3.1 (Termination-insensitive noninterference). *If $\Gamma \vdash c : \ell$ and $m_1 =_{\mathcal{A}} m_2$ and*

$$\langle c, m_i \rangle \rightarrow^* m'_i \text{ for } i = 1, 2$$

then $m'_1 =_{\mathcal{A}} m'_2$.

Theorem 1.3.1 states that if a program is well-typed, and is executed on two \mathcal{A} -equivalent memories, the resulting memories will still be \mathcal{A} -equivalent. In this setting all flows have been verified statically by a type system and no information flow checks are performed during execution. This is the approach taken in Chapters 2 and 3 of this dissertation. An alternative approach is to check information flow violations during execution.

1.3.2 Dynamic Enforcement Mechanisms

Dynamic enforcement of information-flow control goes back to work by Fenton [41]. Intuitively, during execution each variable would hold not just its current value, but also its current security class. At each command generating an explicit flow from some security class A to security class B , the system would verify that $A \sqsubseteq B$.

However, Fenton noted that such a purely dynamic enforcement mechanism was not sufficient for securing information flow, as such a mechanism could not precisely reason about implicit flows. This is, once again, because information-flow control is not just a property of a single program trace, but rather a hyperproperty quantifying over all possible execution paths of a program.

To control implicit flows, Fenton [40] adds a special program counter security class pc that represents an upper bound on the sensitivity of information that could be learned by knowing that the program reached a particular point.

Concretely, whenever a program executes a command $a := e$, the system checks not just that $\llbracket e \rrbracket \sqsubseteq A$ but that $\llbracket e \rrbracket \sqcup pc \sqsubseteq A$. This ensures that the direct flow caused by the command $a := e$ does not cause illegal implicit flows. Whenever a conditional command `if e then c_1 else c_2` branches on the runtime value of e , the program counter label pc is raised to $pc \sqcup \llbracket e \rrbracket$ during the execution of c_1 or c_2 , ensuring that the program counter label correctly upper bounds the sensitivity of the program point.

Depending on the end-to-end security guarantees required by the system, the program counter label could then be lowered from $pc \sqcup \llbracket e \rrbracket$ back to pc once the execution of c_1 or c_2 finished. This distinction is related to the

enforcement of termination-sensitive versus termination-insensitive noninterference, which we return to in Section 1.5.1.

The idea of a label on the program counter to control implicit flows has carried over to static enforcement techniques, where the program counter label is upper bounded statically when verifying the typability of a program [107].

Dynamic enforcement techniques have the pleasant property of being able to precisely reason about information-flow control even in the presence of dynamically changing security classes [84]. Chapter 4 in this dissertation considers a dynamic enforcement technique that combines information-flow control and authorization to handle dynamically changing security policies.

1.3.3 A Note on Duality in Information Flow

Enforcing confidentiality policies means controlling what information flows *out of* the system: the more sensitive the information, the less places it can flow. Dually, enforcing integrity policies means controlling what information flows *into* the system: the more sensitive the information, the *more* places it can flow. This situation is illustrated in Figure 1.2. The duality between confidentiality and integrity was first noted by Biba [21], who observed that his model for integrity was dual to the confidentiality model of Bell and La Padula [19].

With the development of *robust declassification* by Zdancewic and Myers [137], the duality seemed less clear, as robust declassification requires that values can be declassified only in sufficiently high integrity contexts. But recently, the duality has been re-established with the development of nonmalleable information-flow control by Cecchetti et al. [28], who present a security condition *transparent endorsement* that is the dual of robust declassification: values can only be endorsed if they have sufficiently low confidentiality.

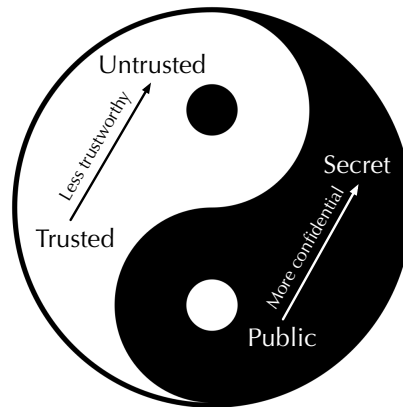


Figure 1.2: Duality of confidentiality and integrity.

The duality was more formally studied by Montagu et al. [75] who define an algebraic structure that they call a label algebra, and show that label

algebras are closed under dualization.

As the work in Chapters 2 and 3 is developed in a setting with an arbitrary security lattice, the duality between confidentiality and integrity implies that the enforcement techniques designed can be considered not just as mechanisms for restricting what an attacker can learn by observing the system but also as mechanisms for restricting how an attacker can affect the system.

The third publication contained in this dissertation (cf. Chapter 4) is developed in a setting with a concrete lattice structure containing both confidentiality and integrity information, where the duality is even more evident.

1.4 Information-Flow Control in the 21st Century

When the Internet became commonplace, language-based information-flow control started focusing on decentralized models of security where mutually distrusting principals could share sensitive information in order to perform their required tasks [82].

In order to operate on shared data, it is sometimes necessary to *downgrade* data from one security level to another. Downgrading of confidentiality is known as *declassification*, and downgrading of integrity is known as *endorsement*.

Myers and Liskov [82] developed the *Decentralized Label Model* (DLM), where each piece of data is associated with a set of *owners*, each of which specify a security policy that must be enforced when accessing the data. While earlier label models restrict downgrading to be performed only by trusted principals, DLM allows each owner of a piece of data to downgrade the data, removing the need for trusted principals.

The next section introduces the basic principles of DLM. The goal is to familiarize the reader with concepts necessary to understand the *Flow-Limited Authorization Model*, which is the topic of Chapter 4 in this dissertation.

1.4.1 The Decentralized Label Model

The label model of Myers and Liskov [82] starts from a principal hierarchy representing *trust* and from that defines a lattice for information flow.

Principals and Principal Hierarchies

A principal is the basic unit in DLM. Examples of principals include users such as Alice and Bob, companies such as Acme³ and even roles such as

³https://en.wikipedia.org/wiki/Acme_Corporation

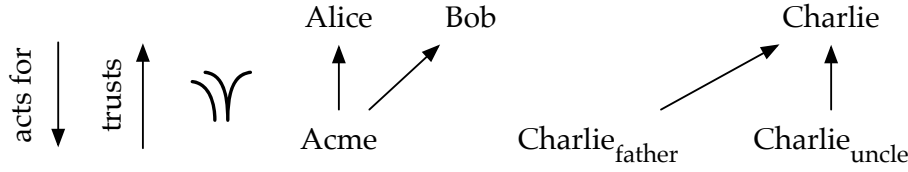


Figure 1.3: An example of a principal hierarchy. Alice and Bob both act for Acme (an example of a group), and Charlie acts for both $\text{Charlie}_{\text{father}}$ and $\text{Charlie}_{\text{uncle}}$ (an example of roles).

Alice_{employee}. Principals delegate trust between them using an *acts-for* relation, written $p \succcurlyeq q$ (pronounced *p acts for q*, or *q trusts p*). Using the trust relation DLM can model concepts such as groups, by delegating trust from the group principal to each member of the group, and roles, by delegating trust from each role to the principal fulfilling that role. Figure 1.3 shows an example of a principal hierarchy containing both groups and roles.

Information-Flow Labels in DLM

A label in DLM is a set of policies, and each policy is either a confidentiality policy⁴ $o \rightarrow p_1, \dots, p_n$, or an integrity policy $o \leftarrow p_1, \dots, p_n$, where o and p_1, \dots, p_n are principals. We call the principal o the *owner* of the policy, and p_1, \dots, p_n the *readers* (resp. *writers*) of the policy.

The intuition behind a label on some data is that each policy on the data must be enforced. That is, a principal p may read from (resp. write to) the data only if every policy associated with the label p specifies p as a reader (resp. writer). For example, if a piece of data is attached the label

$$\{p_1 \rightarrow q, r; p_2 \rightarrow p_1, q, r'; q_1 \leftarrow r; q_2 \leftarrow r, r'\}$$

only the principal q may read the data, as it is the only principal that is included in both confidentiality policies⁵, and only r may modify the data, as it is the only principal that is included in both integrity policies.

Myers and Liskov [82] provide definitions of the relabeling relation \sqsubseteq , the join \sqcup and the meet \sqcap operations, and proves that the relabeling relation is sound (i.e., when $\ell_1 \sqsubseteq \ell_2$ it holds that ℓ_2 allows at most as many flows as ℓ_1).

⁴Myers and Liskov [82] call these privacy policies.

⁵To be precise, the owner of a read policy is implicitly assumed to be a reader of the policy. So p_1 is also allowed to read this data as it is implicitly a member of p_1 's confidentiality policy, and explicitly specified as a reader in p_2 's confidentiality policy.

The Flow-Limited Authorization Model

In this section we introduce a recently developed label model, the Flow-Limited Authorization Model (FLAM), by Arden et al. [9] that unifies the notion of principals and labels. The work on Fabric by Liu et al. [67] left a side channel, mounted by dynamic authorization checks in distributed and decentralized systems, as future work [68]. This side channel is the motivation for the development of FLAM, and Arden et al. prove that authorization queries in FLAM leak no sensitive information to attackers.

Authorization and information flow interact in subtle ways, which is illustrated using the following example by Hirsch et al. [57]: imagine a social network where each user u_1 has a friend list with the security policy that only friends of u_1 (i.e., principals u_2 that have been authorized by u_1) may view u_1 's friend list. Furthermore, imagine that the social network also stores personal images, and that only friends of u_1 may view the images uploaded by u_1 . If a user u_2 attempts to view a picture uploaded by u_1 , u_2 will observe a "401 Unauthorized" HTTP response only if u_2 is not a friend of u_1 , which allows u_2 to learn whether he is a friend of u_1 . But if u_2 is not a friend of u_1 , information about u_1 's friend list may not flow to u_2 ! So the security policy is violated.

Arden et al. present three types of attacks on existing decentralized label models like DLM. The next three sections give an overview of these attacks and how FLAM prevents such attacks.

Delegation Loopholes

As mentioned in Section 1.4.1, DLM can represent group membership using delegations: a principal Alice is said to be a member of the group Emp if $\text{Alice} \succcurlyeq \text{Emp}$. That is, the group principal Emp trusts the group member Alice. In a decentralized system, different principals may disagree on trust, and care must be taken to prevent untrusted principals from affecting authorization decisions. For example, consider a piece of data x that should only be readable by employees of the company Globex.⁶ A sensible label in DLM for x would be $\{\text{Globex} \rightarrow \text{Emp}\}$, expressing that Globex only allows principals trusted by Emp to read x . In particular, Alice can read x as $\text{Alice} \succcurlyeq \text{Emp}$, but there is a subtle issue: as Alice can delegate trust to other principals, she can add members to the group Emp. For instance, if Alice places a delegation $\text{Bob} \succcurlyeq \text{Alice}$ it follows by transitivity of \succcurlyeq that $\text{Bob} \succcurlyeq \text{Emp}$, meaning that now Bob can read x even though Globex never hired Bob. Arden et al. call these attacks *delegation loophole* attacks.

⁶https://simpsons.fandom.com/wiki/Globex_Corporation

This attack has also previously been investigated by Swamy et al. [119], where the authors used a label model with explicit set membership that is not related to delegation of trust.

Poaching

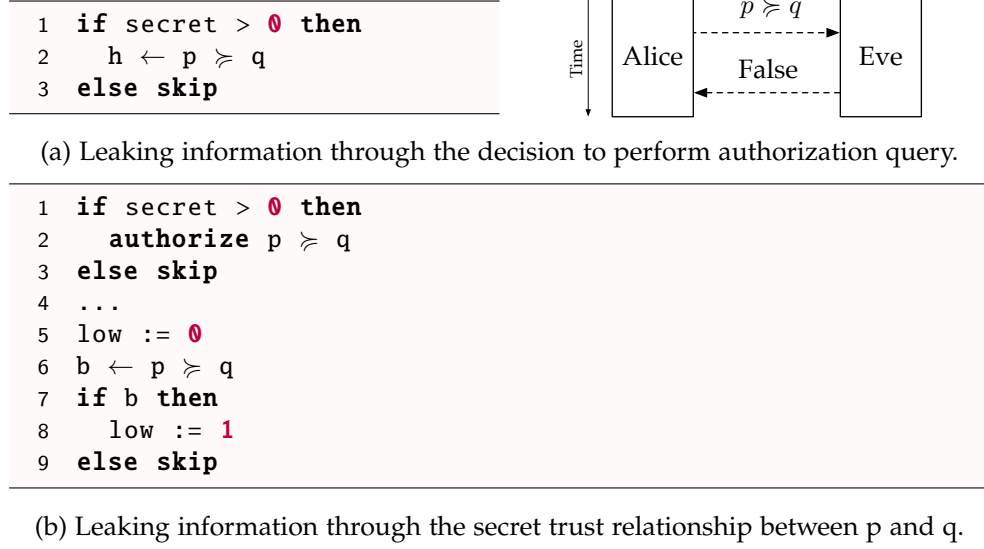
While delegation loophole attacks abuse delegations to *allow* more flows, poaching attacks abuse delegations to *prevent* future revocations of flows. Suppose that Alice is an employee of Globex, which we represent by delegating trust from Emp to Alice. Assume the existence of a piece of data x , labeled with the DLM label $\{\text{Globex} \rightarrow \text{Emp}\}$, which specifies that Globex allows principals trusted by Emp to read the value of x . In particular, since $\text{Alice} \succcurlyeq \text{Globex}$, Alice can read the value of x .

If Alice is ever fired from Globex, the trust delegation $\text{Alice} \succcurlyeq \text{Emp}$ is revoked by Globex, and Alice can no longer read the value of x . However, Alice can read x and store it in a variable y labeled as $\{\text{Emp} \rightarrow \text{Alice}\}$ since $\{\text{Globex} \rightarrow \text{Emp}\} \sqsubseteq \{\text{Emp} \rightarrow \text{Alice}\}$, and so the assignment $y := x$ is allowed. Now, even though Globex revoked the delegation $\text{Alice} \succcurlyeq \text{Globex}$, Alice can still read the value of y to obtain information that was previously labeled as $\{\text{Globex} \rightarrow \text{Emp}\}$.

Information Leaks Through Authorization

In a decentralized setting, no single node has a complete view of the trust relationship between all principals, and nodes must thus query each other to resolve authorization decisions. Decentralized authorization can leak information in two distinct ways: first, the decision to perform an authorization query can leak information if the decision is based on sensitive data. This is an example of a *read channel* [67, 68, 139], where a request to access data leaks information about the node performing the request. As an example, consider the example in Figure 1.4a. If `secret` is greater than zero line 2 checks whether $p \succcurlyeq q$. In order to prove (or refute) this trust relationship, Eve is contacted, which allows Eve to deduce information about the value of `secret`.

The second way authorization can leak information is if the result of an authorization (i.e., whether the authorization procedure outputs “yes” or “no”) is affected by sensitive information. This can happen when a trust delegation between two principals is added based on secrets. For example, consider the program in Figure 1.4b. If `secret` is greater than zero, line 2 delegates trust from q to p . Then, at some point later, the trust relationship between p and q is inspected on line 6, and a public side-effect is executed on line 8 if the relationship holds. Thus an attacker observing the assignment



`low := 1` can deduce that `q` trusts `p`, which implies that `secret` is greater than zero.

Information-Flow Policies on Authorization queries

FLAM solves the problem of information leaks through authorization by parameterizing the judgment for answering authorization queries by information-flow policies (i.e., labels). An authorization judgment in FLAM is written as $\mathcal{H}; c; pc; \ell \vdash p \succcurlyeq q$. Here, \mathcal{H} is a function that maps a node n to which delegations n knows about locally, and c is the node performing the authorization query. The labels pc and ℓ capture the information-flow requirements of the judgment. The program counter label pc is similar to the traditional pc label defined by Fenton [40]: it captures the sensitivity of the context at the point where the authorization query was initiated. It can be thought of as a lower bound on the sensitivity at which nodes can be contacted during authorization.⁷ While pc represents a lower bound on the sensitivity of the *context*, the delegation label ℓ represents an upper bound on the sensitivity of the *result*. The roles of both the program counter label pc and the delegation label ℓ are seen in the FLAM rule for forwarding authorization queries to remote nodes:

$$\frac{\mathcal{H}; c; pc; \ell \models n \succcurlyeq pc^{\rightarrow} \wedge \ell \quad \mathcal{H}; n; pc \sqcup \ell \sqcup c^{\leftarrow}; \ell \sqcap c^{\rightarrow} \vdash p \succcurlyeq q}{\mathcal{H}; c; pc; \ell \models p \succcurlyeq q}$$

⁷However, the program counter label is also used to define *robust* [137] authorization judgments that prevent delegation loopholes and poaching attacks. I refer the reader to Arden et al. [9] for details on robust authorization.

The first premise formalizes the intuition that the node n being contacted can be trusted to learn (1) the information that caused the query to be initiated, and (2) the delegations used so far in the proof search. This uses the robust judgment \models , which ensures that attackers cannot influence the result of this judgment. The second premise formalizes the forwarding of the trust query to the node n . Here, the program counter label is raised to $pc \sqcup \ell \sqcup c^{\leftarrow}$ to reflect that the “control flow” of the proof search depends on (1) information up to ℓ , and (2) the fact that node c decided to perform a forwarding of the query. Finally, the delegation label is attenuated to $\ell \sqcap c^{\rightarrow}$ to prevent node n from using delegations that c is unauthorized to know about.

Epistemic Formulations of Noninterference

Epistemic formulations of noninterference have been used in the literature [12, 16, 36, 52], and throughout this dissertation we follow the presentation by Askarov and Sabelfeld [12] who formalized an attacker’s knowledge as the set of memories consistent with the trace they observed. That is, the smaller the attacker’s knowledge set is, the more certain the attacker is about the initial secrets. And dually, a larger knowledge set corresponds to more uncertainty.

Traditionally, noninterference is stated as a two-run formulations (as in Theorem 1.3.1), but Broberg et al. [24] argues that an epistemic formulation is a more natural way of stating the property. For this reason most security properties⁸ stated in this dissertation are formulated as epistemic properties. To show how an epistemic formulation of noninterference can be expressed, one can instrument the language of Volpano et al. [130] with events that capture modifications of the memory:

$$\begin{aligned} ev &::= \varepsilon \mid (x, v) \\ t &::= \varepsilon \mid ev \cdot t \end{aligned}$$

That is, an event ev is either an empty event ε or an assignment event (x, v) capturing the assignment of v to the variable x . The *observable* part of the trace t is the part of the trace $\lfloor t \rfloor_{\mathcal{A}}$ that the attacker \mathcal{A} can observe.

$$\lfloor \varepsilon \rfloor_{\mathcal{A}} = \varepsilon \qquad \lfloor ev \cdot t \rfloor_{\mathcal{A}} = \begin{cases} ev \cdot \lfloor t \rfloor_{\mathcal{A}} & ev = (x, v) \text{ and } \Gamma(x) \sqsubseteq \mathcal{A} \\ \lfloor t \rfloor_{\mathcal{A}} & \text{otherwise} \end{cases}$$

We write $t_1 =_{\mathcal{A}} t_2$ if $\lfloor t_1 \rfloor_{\mathcal{A}} = \lfloor t_2 \rfloor_{\mathcal{A}}$. We can now define the attacker’s knowledge $k(c, t)$ to be the set of initial memories that cause c to produce a particular trace t .

⁸The noninterference statement in Pedersen and Askarov [93] is formulated as a two-run property instead of an epistemic property due to space reasons.

Definition 1.4.1 (Attacker knowledge). *Given a command c and a trace t , let $k(c, t)$ be the set of memories that are consistent with observing t when executing c .*

$$k(c, t) \triangleq \left\{ m \mid \langle c, m \rangle \xrightarrow{t'}^* \wedge t =_{\mathcal{A}} t' \right\}$$

Having defined the knowledge of the attacker \mathcal{A} , we can now design security policies by specifying how small \mathcal{A} 's knowledge set is allowed to become. The noninterference result presented in Theorem 1.3.1 corresponds to the following policy, which states that \mathcal{A} must only be able to exclude memories based on the program's termination-behavior.

Definition 1.4.2 (Terminating memories). *Let $k_{\mathcal{A}}^{\downarrow}(c, m)$ be the set of memories that are \mathcal{A} -equivalent to memory m such that command c terminates.*

$$k_{\mathcal{A}}^{\downarrow}(c, m_1) \triangleq \{ m_2 \mid m_1 =_{\mathcal{A}} m_2 \wedge \langle c, m_2 \rangle \rightarrow^* \}$$

Theorem 1.3.1 can now be restated using attacker knowledge as follows:

Theorem 1.4.1 (Termination-insensitive noninterference). *If $\Gamma \vdash c : \ell$ and $\langle c, m \rangle \xrightarrow{t}^*$ then $k(c, t) \supseteq k_{\mathcal{A}}^{\downarrow}(c, m)$.*

That is, any memory that leads to termination will be consistent with the trace emitted by the program.

1.5 Covert Channels

Section 1.2 introduced language-based information-flow control in a simple setting where an attacker observes the final memory after the execution of a program finishes, or observes a sequence of events describing the assignments that occurred during the execution of the program. However, these attacker models fail to capture a number of possible information flows. In particular, the semantics presented does not capture *covert channels*, which are communication channels not originally intended for communication but which can be abused by attackers for the transmission of information.

This observation is not new; Lampson [66] noted that one process can communicate with another process by opening and closing file handles. He also noted that a process can transmit information by carefully scheduling CPU intensive computations and apply techniques from information theory to obtain a reliable communication channel.

Today, the list of covert channels is very long. This section reviews covert channels based on the termination behavior of programs and their timing behavior.

1.5.1 Termination

When designing information-flow control policies and mechanisms, the goals of security and practicality are often at odds. At one extreme, rejecting all programs as “possibly insecure” certainly guarantees that the mechanism enforces security,⁹ but enforcing such a policy will not help programmers write secure code. A less extreme, but just as important, decision is whether to allow information leaks through the termination behavior of a program.

Termination-*insensitive* noninterference [110] is a variant of noninterference that allows transmission of information through a program’s termination behavior. Theorems 1.3.1 and 1.4.1 are examples of termination-insensitive noninterference results (i.e., they only guarantee preservation of \mathcal{A} -equivalence when both programs terminate), and many researchers have achieved similar termination-insensitive noninterference guarantees [8, 12, 17, 28, 55, 61, 69, 81, 83, 92, 93, 96, 103, 104, 115, 128, 143, 145].

Askarov et al. [13] showed that if an attacker observes intermediate output events, the common [83] argument that “the termination channel leaks only one bit of information” breaks down. In fact, if the attacker observes intermediate events, the termination channel can be used to leak arbitrarily large secrets. Askarov et al. define progress-insensitive security as a version of termination-insensitive noninterference that is suitable to a setting in which an attacker observes intermediate outputs, and proves that the “best possible attack” (in a sequential setting) is a brute-force search through the exponential search space of secrets:

```

1  i = 0
2  while i ≤ N do
3    output i;
4    if i = h then (while 1 do skip) else skip
5    i = i + 1

```

Here, we assume that $h \in [0, N]$. For instance, an attacker could pick N to be the largest possible integer representable on a machine. An attacker observes outputs $0, 1, \dots, h-1$, and when no more output is being produced the attacker can deduce that the value of h .

In the sequential setting this is the best an attacker can do, and colloquially we say that “the termination channel cannot be amplified”. However, adding a primitive for concurrency (i.e., a **fork** operator) makes it possible to exploit the termination channel to leak secrets in linear time [54]: an at-

⁹At least those policies expressible as safety (trace or hyper) properties [31].

tacker can spawn $\lceil \log N \rceil$ copies of the above attack that check if each bit is 0:

```

1  i = 0
2  while i <  $\lceil \log N \rceil$  do
3      fork (if  $h_i = 0$  then (while 1 do skip) else skip;
4          output i)
5      i = i + 1

```

We write h_i for the i 'th bit of h (i.e., $h_i = (h \gg i) \& 1$). Each thread either diverges if the i 'th bit of h is 0 and thus each output of i corresponds to a non-zero bit of the secret. Intuitively, the extra potency of the termination channel arises because the divergence of one thread has no effect on the divergence of other threads.

As the publications contained in this dissertation only consider the sequential setting, we prove termination-insensitive noninterference and enjoy the benefit of more permissive enforcement mechanisms and the knowledge that the termination channel cannot be amplified.

1.5.2 Timing

A more general covert channel is that of a *timing* channel, in which the execution of a program depends on sensitive information. Zdancewic and Myers [138] distinguish between *external* and *internal* timing observations. An external timing observation is made by an outside observer using a “stopwatch” (see, e.g., the pioneering work by Kocher [62]), and an internal timing observation is an observation made by another program running on the same system.

External timing attacks (i.e., attacks based on external observations) are typically harder to reason about without appealing to lower-level mechanisms [61, 138]. Chapters 2 and 3 of this dissertation are concerned with eliminating external timing attacks.

For the rest of this section we review well-known sources of timing channels mounted by hardware, the operating system, and even by the high-level features in modern programming languages.

Cache-Based Timing Attacks

The main memory of a traditional computer typically includes multiple layers of data and instruction caching, which is usually denoted as the L_1 , L_2

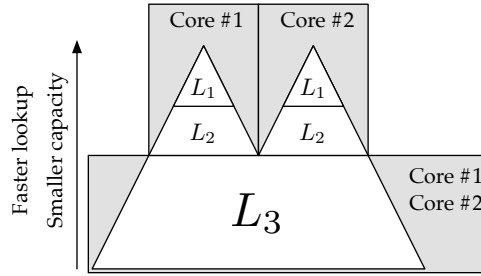


Figure 1.5: Organization of caches in a typical multi-core CPU.

and L_3 caches. Figure 1.5 shows a typical organization of caches in a multi-core CPU with three levels of caching. Cache L_i is usually smaller than L_{i+1} , but allows for faster access to its content. In this particular architecture, each core has its own first two levels of caching, but the L_3 cache is shared between the cores. Whenever the CPU requests data from main memory, each level of cache is searched, and a lookup in the main memory is performed only if the data is not found in any cache. As processes on the same core share L_1 and L_2 caches, and even share L_3 caches across different cores, this shared resource is very suitable for inter-process external timing attacks. In fact, Osvik et al. [90] and Bernstein [20] have independently demonstrated that such attacks are feasible on typical computer architectures and shown how the full key for the Advanced Encryption Standard (AES, [38]) can be obtained from observing the effects of S -box¹⁰ lookups on any level of caching.

Similarly, Percival [98] studied the OpenSSL¹¹ implementation of the Rivest–Shamir–Adleman (RSA, [105]) algorithm for public-key encryption, and showed that the use of the “sliding window” technique for efficient modular exponentiation, which precomputes certain powers, makes the implementation vulnerable to external cache-based timing attacks.

Language-Based Solutions to External Timing Attacks

Zhang et al. [143] propose a language-based approach to the prevention of external timing attacks. They expose an abstract *machine environment* at the software level and annotate program commands with *read* and *write labels* to control how the machine environment is modified during execution. A read

¹⁰S-box refers to a substitution-box. Intuitively, a substitution-box is a lookup table, and is used for efficient implementations of many cryptographic algorithms.

¹¹From <https://openssl.org>: “OpenSSL is a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is also a general-purpose cryptography library.”

```

1  if h1 then[L,L]
2    h2 := low1[H,H]
3  else
4    h2 := low2[H,H];
5  low3 := low1[L,L]

```

Figure 1.6: Preventing external cache timing attacks using read and write labels [143].

label specifies an upper bound on which parts of the machine environment may affect the execution time of a command, and a write label specifies a lower bound on which part of the machine environment a command may modify.

Figure 1.6 presents an example of a simple external timing attack, mounted by the data cache, that is prevented using read and write labels. On typical hardware, a read of variable `low1` on line 2 would update the data cache such that the subsequent read of the same variable on line 5 would execute faster. Thus, an attacker would be able to distinguish whether line 2 or line 4 is executed based on the execution time of the above program and thereby learn information about the secret variable `h1`.

The write labels (i.e., the second label component) on line 2 and 4 ensure that only the parts of the machine environment corresponding to the security level `H` is updated. Dually, the read label (i.e., the first label component) on line 5 ensures that the execution time only depends on the parts of the machine environment corresponding to the security level `L`. Zhang et al. then design a type system that ensures correct selection of read and write labels.

Enforcement mechanisms at lower levels of abstraction have also been proposed. Zhang et al. [144] present an extension of the hardware description language Verilog [47] with information-flow labels for controlling the subtle flows arising from hardware features. Zhang et al.’s [144] goal is to enforce the complex security policies required by Zhang et al. [143] using a type system with dependent information-flow types (i.e., where the security label on a type can depend on runtime values). Using (the noninterference result of) this type system, they prove the timing-sensitive security of a complete MIPS processor.

Ferraiuolo et al. [43] present Hyperflow, an extension of the RISC-V architecture that enforces timing-sensitive nonmalleable information-flow control [28]. The architecture encodes FLAM labels using a bit vector representation which they call the hypercube label model that allows for efficient computa-

tions of meets, joins and relabeling checks. The extension is implemented in a security-typed hardware description language embedded in Scala called Chiselflow, which compiles to an intermediate representation for which they prove that well-typed programs satisfy observational determinism [138]; a common security definition for nondeterministic systems, which is also described in the next section.

Continuing the line of work on Hyperflow, Zagieboylo et al. [136] present a modified RISC-V instruction set architecture that enforces timing-sensitive nonmalleable information-flow control dynamically.

Scheduler-Based Timing Attacks

Most systems today use multiple threads of execution. In general, concurrency introduces nondeterminism in the semantics of programming languages, and the traditional notion of (termination-insensitive) noninterference is no longer an adequate notion of security for such systems (see, e.g., Smith and Volpano [114] or Sabelfeld and Sands [108] for possibilistic or probabilistic formulations of noninterference, or Zdancewic and Myers [138] for a definition based on observational determinism).

For timing channels specifically, concurrency is especially dangerous as it allows the internalization of otherwise external timing channels [129, 138]. As an example, consider the program in Figure 1.7 by Zdancewic and Myers [138].

<pre> 1 x := true; // Thread 1 2 if h then delay(100) else skip; 3 x := false </pre>		<pre> // Thread 2 delay(50); low := x </pre>
--	--	--

Figure 1.7: Leaking one bit of h through an internal timing channel.

Here, both threads are secure if run in isolation, but when running concurrently the program leaks the one bit of h due to internal timing leakage: if h is non-zero, thread 1 waits for 100 steps before executing the assignment $x := \text{false}$, meaning thread 2 executes the assignment $\text{low} := x$ while x is still true, but if h is zero, the assignment in thread 2 happens after x has been assigned to false. So an attacker observing the final value of low can deduce one bit of h .

Lazy Evaluation-Based Timing Attacks

Non-strict (i.e., lazy) evaluation has been considered in the context of security going back to Sabelfeld and Sands [110] who noted that the termination-insensitive enforcement mechanism by Heintze and Riecke [56] could be strengthened to a termination-sensitive enforcement merely by changing the call-by-value semantics to a non-strict semantics (i.e., call-by-name or call-by-need). Based on this observation Sabelfeld and Sands argue that “lazy programs are intrinsically more secure than strict ones”. However, Sabelfeld and Sands only considered a sequential setting with no timing-sensitive security properties.

In contrast, when both call-by-need evaluation and concurrency are present in a language, Buiras and Russo [25] have showed that an internal timing channel can be constructed. Intuitively, the attack is similar to the internal timing channel attack in Figure 1.7 where two threads race for a write to a variable that is observable to an attacker. Although Buiras and Russo use Haskell to present the attack, Figure 1.8 restates the attack using a simple imperative language with fork-based concurrency and non-strict evaluation (i.e., commands `lazy` and `force` for constructing and forcing lazy computations, respectively).

```

1  thunk = lazy(computation);
2  fork(if h then force(thunk) else skip); // Thread 1
3  delay(2*N);
4  fork(force(thunk); low := 0);           // Thread 2
5  fork(delay(N); low := 1)                // Thread 3

```

Figure 1.8: Leaking one bit of h through an internal timing channel using lazy evaluation.

On line 1, a lazy computation is constructed and stored in variable `thunk`. Line 2 forks a thread that forces the evaluation of `thunk` only if h is non-zero. Line 3 delays the main thread long enough for thread 1 to finish execution. Finally, lines 4 and 5 create two threads that race on a write to the \mathcal{A} -observable variable `low`. Since we assume the code is untrusted, the attacker can provide a suitable value for N that activates the attack. To see how the attack works, consider the case where h is non-zero. Then `thunk` will be forced in thread 1, causing the subsequent assignment `low := 0` to happen *before* in the assignment `low := 1` in thread 3. Thus, given a sufficiently large N the assignment `low := 1` in thread 3 overwrites the previous write to `low`, and thus the final value of `low` is 1.

However, if h is zero, line 2 does not force the value of `thunk`, and thus the evaluation of `force(thunk)` will consume more time, causing the assignment `low := 0` in thread 2 to happen *after* the assignment `low := 1` in thread 3 (assuming a suitable value for N), and so the final value of `low` is 0.

Note that, as the termination channel can be amplified to leak secrets in linear time using concurrency (see Section 1.5.1), we are interested in progress-sensitive security [13] in this example and so we need to fork a thread (i.e., thread 1) in order to branch on the secret value h and still perform side effects observable to the attacker on line 4 and 5.

The solution proposed by Buiras and Russo, which was later formalized and proven secure by Vassena et al. [127], is to restrict how thunks are shared between threads using a construct called `lazyDup` originally proposed by Breitner [23].¹² Using their approach, the value of the unevaluated computation, `thunk`, will be duplicated when forking thread 1 on line 2. This ensures that `thunk` will always need to be evaluated on line 4 independently of the value of h .

1.6 Outline of the Dissertation

In this section we briefly describe the publications contained in the second part of this dissertation. The following publications are included in the dissertation:

- [92] *From Trash to Treasure: Timing-Sensitive Garbage Collection*
Mathias V. Pedersen and Aslan Askarov.
In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.
Included in Chapter 2.
- [93] *Static Enforcement of Security in Runtime Systems*
Mathias V. Pedersen and Aslan Askarov.
In *Proceedings of the IEEE Symposium on Computer Security Foundations (CSF)*, 2019.
Included in Chapter 3.
- [96] *Programming with Flow-Limited Authorization: Coarser is Better*
Mathias V. Pedersen and Stephen Chong.
In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.
Included in Chapter 4.

¹²Buiras and Russo [25] and Breitner [23] call this construct `deepDup`, but we follow the convention given by Vassena et al. [127] and refer to it as `lazyDup`.

The author has contributed significantly to all of the above-mentioned publications. Minor layout adjustments and typo corrections have been made in the papers since their publication.

1.6.1 Leaking Information Through Automatic Memory Management

The first publication in this dissertation, *From Trash to Treasure: Timing-Sensitive Garbage Collection* [92], appeared at the IEEE Symposium on Security and Privacy 2017, and presents the first timing attack on real garbage collectors in modern programming language runtime systems, along with sufficient conditions for secure garbage collection in the presence of external timing measurements.

The presented attacks target two industrial scale virtual machines: Java version 1.8 running on Java HotSpot(TM) 64-Bit Server VM (build 25.77-b03), and NodeJS version 6.2, both representing the latest versions of each system at the time of publication. The attacks demonstrate that it is possible to mount an amplifiable timing channel capable to leaking arbitrarily large secrets in linear time, and that the channel is observable over a network connection with sufficiently low latency.

The fact that the timing channel can be amplified of leak arbitrarily large secrets separates it from “benign” channels like the termination channel and demands an enforcement mechanism for controlling the information leakage. The paper presents a formal calculus for an imperative language with automatic memory management, and proves that the combination of:

1. a security level-partitioned heap,
2. a security-type system,
3. and a security-level aware garbage collector

enforces a version of possibilistic, termination-insensitive, and timing-sensitive noninterference. The proof of noninterference has been mechanized in the Coq proof assistant [120]. This work demonstrated the dangers of IFC language runtimes relying on existing, non-information-flow-aware components.

1.6.2 Provable Absence of Information Leaks in Runtime Systems

The second publication in this dissertation, *Static Enforcement of Security in Runtime Systems* [93], appeared at the IEEE Computer Security Foundations

Symposium 2019. The publication presents the language Zee: a security-typed language with static enforcement of termination-insensitive noninterference. The language is “low-level enough” to implement common runtime related tasks such as garbage collection and thread scheduling *in the language itself*, yet “high-level enough” to reason about the security guarantees of well-typed programs. As it happens, the idea of Zee is nicely summarized by the following quote from Sabelfeld and Sands [108, p. 205]:

Abstractly we will take a scheduler to be a mechanism for selecting threads which itself satisfies some noninterference property, i.e., its behaviour is independent of high data.

As Zee can reason about the implementation of the runtime environment, previous work on designing specialized, security-aware runtime components [61, 92, 108, 116] can be implemented as well-typed programs in the language itself, and the security theorem for Zee then guarantees a version of termination-insensitive and timing-sensitive noninterference [92] for the particular runtime environment component that has been implemented.

We have implemented a type checker and interpreter for the language in Haskell [100], which we use to develop two case studies: a secure cooperative thread scheduler and a mark-and-sweep garbage collector. Both programs respect the typing discipline of the language, and as such the noninterference theorem guarantees that the programs satisfy noninterference. Implementing a compiler for Zee is left as future work, and many challenges need to be solved in order to achieve this. For instance, the operational semantics of the language requires a second stack, similar to the shadow stack introduced by Abadi et al. [3], for managing runtime type information. This stack must be inaccessible at the language-level to ensure the integrity of the type information during execution, as this information is trusted and used to enforce information-flow policies at runtime.

1.6.3 Coarse-Grained Enforcement of Flow-Limited Authorization

The third paper contained in this dissertation, *Programming with Flow-Limited Authorization: Coarser is Better* [96], appeared at the IEEE European Symposium on Security and Privacy 2019. The paper proposes the thesis that the Flow-Limited Authorization Model (FLAM, see Section 1.4.1) is better utilized in a setting with dynamic enforcement of coarse-grained IFC. Specifically, we argue that the security guarantees of the logic are enforced at a coarse-grained level, where a single label (i.e., the delegation label described in Section 1.4.1) encapsulates the sensitivity of the all dependencies of the query result. This shares some similarities with coarse-grained enforcement

of IFC as seen in languages, such as LIO [115], and in operating systems [37, 64, 140].

To support the claim that the logic is better suited for coarse-grained enforcement of IFC, we implement the judgment for securely deciding trust relationships between principals directly in the dynamically enforced, coarse-grained IFC language LIO.

The resulting language, which we call Flamio, supports the enforcement of expressive and dynamic security policies, and contributes to the research on the interaction between authorization and information-flow control [8, 9, 57].

We have proven that program executions in the Flamio language satisfy termination-insensitive noninterference, and we have implemented the language as a Haskell library that provides a monadic interface, similar to how LIO is implemented in Haskell.

Part II

Publications

Chapter 2

From Trash to Treasure: Timing-Sensitive Garbage Collection

MATHIAS V. PEDERSEN, AARHUS UNIVERSITY, DENMARK
ASLAN ASKAROV, AARHUS UNIVERSITY, DENMARK

In Proceedings of the IEEE Symposium on Security and Privacy (S&P), 2017.

Abstract

This paper studies information flows via timing channels in the presence of automatic memory management. We construct a series of example attacks that illustrate that garbage collectors form a shared resource that can be used to reliably leak sensitive information at a rate of up to 1 byte/sec on a contemporary general-purpose computer. The created channel is also observable across a network connection in a datacenter-like setting. We subsequently present a design of automatic memory management that is provably resilient against such attacks.

2.1 Introduction

When a computer system allows third-party code to access sensitive information, it is necessary to ensure confidentiality of the sensitive information handled by the code. Language-based information flow control is a popular approach to solve this problem [107]. This approach uses programming language techniques to analyze information flows in the untrusted programs

before and/or during the execution in a way that prevents potentially insecure code. The advantage of this approach is that it allows fine-grained control, compared to coarse-grained systems approaches. The disadvantage is that source-code analysis is limited to flows that only have control graph representation, and that allows malicious code to still leak sensitive data using runtime side-channels such as CPU caches [90], schedulers [138], or programming-languages features such as lazy evaluation [25].

This paper studies another important aspect of program runtime – automatic memory management. We show that memory management represents a vulnerable shared resource through which an attacker can launder sensitive information. We present a series of simple attacks on modern runtimes, in particular Java sequential and parallel garbage collections and V8 default garbage collector, that illustrate the potential of the attack.

Attack model We consider a threat model where an attacker-provided program operates on confidential information. The attacker observes the public input and output of the program, but does not observe either the secret input or the output. Furthermore, the attacker code is subjected to a number of syntactic and runtime checks that prevent it from directly leaking the secret input.

We assume that the attacker program consists of secret-dependent (high) and secret-independent (low) computations. High computations can access sensitive data, but cannot directly communicate with the attacker. Low computations may communicate with the attacker or affect the public output, but their execution cannot immediately depend on secrets.

Figure 2.1 illustrates the high-level idea behind the attacks. The attacks are designed so that the high computation influences the amount of allocated and reclaimable memory, which in its turn, influences the timing of allocations in the low computations, via the garbage collector.

If there is no free space at the time of an allocation in the low computation, invocation of the garbage collector introduces observable delays. This delay is observed by two timing observations, before and after the allocation in the low computation. The attacks do not rely on measuring the timing of the high computations – these in fact are considered to be secret-independent.

The low computation can do its timing observations using a number of ways. Our initial set of attacks uses the system clock. However, access to the local clock is not essential. Local time measurements can be substituted by external ones if the low computation has network access and can send dummy messages.

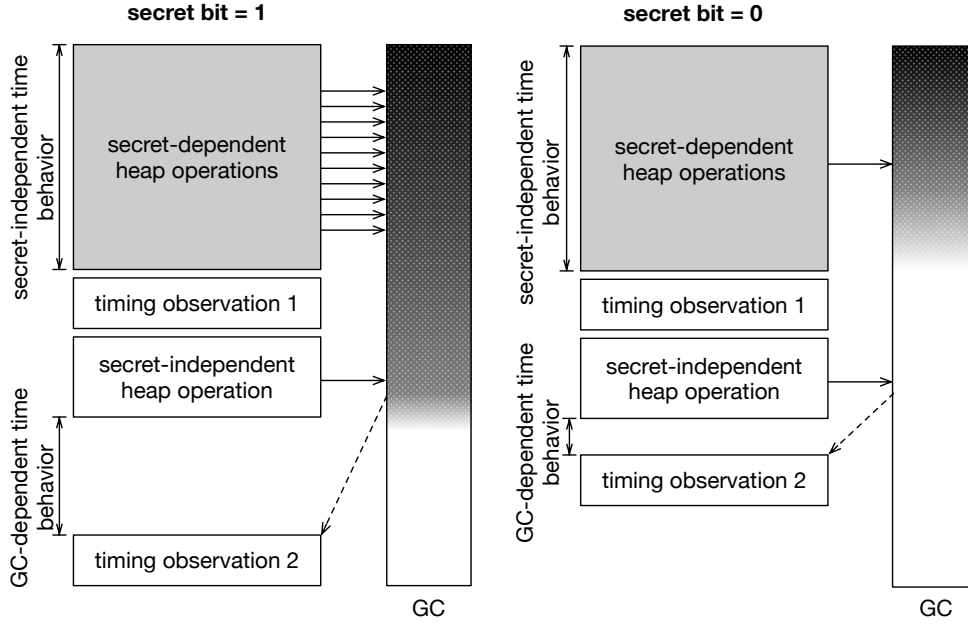


Figure 2.1: Attack scenario – high computations are marked in gray; low computations are marked in white; both low and high computations share the memory management component (GC). Heap manipulation by the high process affects the state of the GC in a way that is later observable by the low computation.

We further demonstrate that the observed channel can be amplified to leak arbitrary amount of information. Our experiments achieve the rate of nearly 1 byte/sec on a modern general-purpose laptop.

The attacks may not be surprising in hindsight, but their consequences are important. Automatic memory management is crucial in the implementation of modern object-oriented or functional programming languages. For strong-information security, a secure runtime is a must. Existing prototypes rely on source-to-source compilation [86, 112] or language-based monitoring [55, 115], yet they reuse commodity runtimes that are vulnerable to the types of attacks we describe here.

We note that while there have been remarks in the literature about the danger of memory management in information flow settings [91], we are not aware of previously published attacks or proposals that focus on the timing channels through memory management.

To address the problem of leaks via garbage collection, we study a model of a programming language that uses abstract secure runtime and security types to enforce security. Our programming language includes a command

for obtaining the current time, which allows us to cover a wide range of attacker models, because internal timing differences in a program can be converted into publicly observable output. Note the formal semantics of the language is designed so that it isolates leaks via garbage collection from other timing channels by using a non-standard primitive $\text{at } \ell \text{ with bound } e \text{ do } c$. This primitive pads the execution time of command c by the value of expression e .

We observe that garbage collection creates a bi-directional information flow channel. Securing garbage collection requires that high allocations cannot be collected in low computations and vice versa. We prove that the combination of these runtime restrictions with standard information flow type system is sufficient to close leaks via memory management.

In summary, the contributions of this paper are the following:

1. We develop the first amplifiable covert channel via automatic memory management. We show that the channel can be observed locally and over a network.
2. We observe that garbage collection creates a bi-directional information channel, which severely restricts the design space of securing automatic memory management. We present formal requirements that secure garbage collectors must satisfy.
3. We demonstrate that a secure garbage collector can be incorporated with traditional information flow analysis to provably establish a non-interference property.

The rest of the paper is structured as follows. Section 2.2 provides a background on timing channels and garbage collection. Section 2.3 explains our attacks and their results. Sections 2.4–2.5 develop a formal language model for a small imperative language with allocatable arrays and garbage collection. Section 2.6 designs a type system necessary for secure coordination with the runtime. Section 2.7 studies the resulting security guarantees. Section 2.8 discusses the applicability of real-time garbage collectors in the light of the discovered attacks. Sections 2.9 and 2.10 discuss the related work and conclude.

2.2 Background

Programming languages that focus on information flow security statically reject programs that contain information flow violations, such as explicit and implicit flows.

A common approach is to add security labels to types, corresponding to the confidentiality of the information stored in a variable of the given type. A natural consequence of security labels on types is the notion of a program counter label pc that determines the security context of the execution. However, many such approaches ignore other potential sources of information leakage.

A particularly dangerous source of information leaks is time. Timing dependencies in programs may be direct or indirect [143].

```

1  if (h > 0) then /* long computation */
2  else skip

```

An example of a direct timing dependency is the program on the left, where the decision to take one branch or the other depends on a confidential guard h . While direct timing attacks are difficult to close, they have control-flow representation, and thus are amenable to language-based mitigation [5, 143].

Indirect timing channel attacks are caused by interaction with the runtime system of the programming language, or the hardware upon which the program is running on. As such, they are much harder to close, because that requires careful interaction between the programming languages technology and the underlying runtime, often including the OS and the CPUs.

The focus of this paper is on indirect timing channel created by one aspect of language runtime system, specifically the automatic memory management.

We start with a brief overview of basic garbage collection concepts.

2.2.1 Garbage Collection Techniques

A garbage collector discovers which parts of the heap contain objects that will definitely not be accessed in the future, and then reclaims the memory occupied by these objects, allowing that memory to be reused.

Objects stored in the heap can point to other objects, and the heap objects thus form a directed graph. The root nodes of this object graph are the variables in the program, which provides “entry points” into the graph.

Determining whether an object will be accessed in the future is an undecidable problem [78], and thus garbage collection schemes conservatively approximate this property by assuming that every object reachable from the root nodes of the object graph will be accessed in the future.

The two garbage collection strategies attacked in this paper are mark-and-sweep collectors, and copy collectors.

Mark-and-sweep A mark-and-sweep garbage collector operates in two phases: A marking phase, where all reachable objects are marked as “live”, and a sweep phase, where unmarked objects in the heap are reclaimed. Note that the cost of a mark-and-sweep collection is the sum of the cost of marking, and the cost of sweeping. The cost of marking is linear in the size of the reachable objects, and the cost of sweeping is linear in the size of the entire heap.

One way to avoid having the cost be linear in the size of the entire heap is to use a copy collecting algorithm.

Copy collection A copy collecting garbage collector partitions the heap in two partitions of equal size. These partitions are called *from-space* and *to-space*. An invariant of this algorithm is that at any point during the execution of the program, only the from-space is modified.

When the from-space partition is filled, the collector builds a copy of the object graph in the to-space partition. This is known as the evacuation phase. Afterwards all of the memory in the from-space is reclaimed, and the to-space becomes the new from-space, and vice versa.

Generational collection Efficient garbage collectors avoid traversing the entire object graph by assuming the *weak generational hypothesis* which states that “most objects die young” [59, 125], meaning that newly allocated objects on the heap become unreachable fast.

The heap can then be partitioned in several partitions, known as generations. All allocations are initially stored in a small “young” generation, and a garbage collection invocation need only traverse this small partition. When an object survives a collection, it is moved to an “older” generation.

A minor collection is a collection that only traverses the object graph in the young generation, and a major collection is a collection that traverses both generations.

2.3 Attacking JVM and V8

This section presents two general amplifiable timing attack strategies that exploit the garbage collector in order to leak one bit of information. Both attacks work for two garbage collection strategies used by Java, as well as for the generational mark-sweep/mark-compact strategy used in V8.

The section also presents a technique for amplifying the one-bit leak to a general N-bit leak that works for all of the garbage collection strategies mentioned above.

2.3.1 High Dependency in Low Context

This attack exploits the fact that, during evacuation from from-space to to-space, the amount of bytes copied depends on the reachable memory at the current point in the program. Thus, by creating a sufficiently large difference in reachable and unreachable nodes, the time required to perform a minor/major garbage collection becomes observable.

Figure 2.2 demonstrates the attack in Java. The example leaks whether $h > 0$ by observing the time difference caused by the allocation on line 15. If the value of *diff* is large then $h > 0$, and otherwise $h \leq 0$.

The attack works as follows. Suppose that constants $size_1$ and $size_2$ are chosen so that the following constraints hold, where *collectionThreshold* is an experimentally obtained constant that triggers garbage collection, and S is the number of bytes required to represent an integer in Java.

- $2 \cdot S \cdot size_1 \leq collectionThreshold$
- $S \cdot (2 \cdot size_1 + size_2) \geq collectionThreshold$

Line 1 allocates a new array, and keeps a reference to that array in the variable *a*. If $h > 0$ we allocate a new array on line 6 and store a reference to this array in the variable *b*. On the other hand, if $h \leq 0$ we allocate a new array and store the reference in the variable *c*, which will become unreachable as soon as we reach line 13. Note that on line 11 we keep a reference to the array allocated on line 1, meaning that *a* and *b* point to the same array. Now assume the allocation on line 15 invokes the garbage collector. If $h > 0$ there will be two distinct arrays that need to be copied from from-space to to-space, meaning that $2 \cdot size_1$ integers will be copied. However, if $h \leq 0$ then the only array which is reachable, and thus should be copied, is the array allocated on line 1. Thus we only copy $size_1$ integers. This difference in the number of bytes that should be copied creates an observable difference in timing.

2.3.2 Low Modification in High Context

The previous program demonstrated how the evacuation of sensitive information in public contexts can lead to a covert channel. The next attack shows how garbage collection of public information in a sensitive context also leads

```

1  int[] a = new int[size1];
2  int[] b = null;
3  int[] c = null;
4  int[] d = null;
5  if (h > 0) {
6      b = new int[size1];
7      d = a;
8  }
9  else {
10     c = new int[size1];
11     b = a;
12 }
13 c = null;
14 long before = System.nanoTime();
15 int[] x = new int[size2];
16 long after = System.nanoTime();
17 long diff = after - before;
18

```

Figure 2.2: Java program leaking one bit of information based on evacuation time.

to a covert channel. Consider the program in Figure 2.3 and suppose that constants size_1 and size_2 are chosen so that the following constraints hold.

- $S \cdot \text{size}_2 \leq \text{collectionThreshold}$
- $S \cdot (\text{size}_1 + \text{size}_2) \geq \text{collectionThreshold}$

If $h > 0$ the allocation on line 2 partially fills up the heap, so that the allocation on line 7 triggers a garbage collection, and thus the value of diff is large.

However, if $h > 0$ is false then no garbage collection occurs on line 7, as the size of the memory does not exceed the implementation’s threshold for garbage collection. This results in a small value for diff .

2.3.3 Amplifying the Attacks

We now amplify the leakage of the attacks described in Section 2.3.1 and 2.3.2. This leads to an attack that leaks the value of a 32 bit integer. To illustrate the technique we describe how the attack from Section 2.3.1 can be extended.

First, note that the attacks cannot be extended naively by repeating the algorithm for each bit. To see why, assume this approach is taken and that we

```
1  if(h > 0) {  
2      int[] b = new int[size1];  
3      b = null;  
4  }  
5  
6  long before = System.nanoTime();  
7  int[] c = new int[size2];  
8  long after = System.nanoTime();  
9  long diff = after - before;  
10
```

Figure 2.3: Java program leaking one bit of information based on the presence of a garbage collection.

enter the first iteration with 0% of the available memory having been allocated. We would like to keep this 0% allocated memory as a loop invariant, so that we can repeat the attack for each bit.

Next, we might allocate memory equal to 75% of the available memory, which we turn into garbage by removing any referencing to the allocated memory. We then perform a “leaky allocation” of some amount of memory which will cause a garbage collection to occur (i.e., we request strictly more than 25% of the available memory, such that we in total have requested strictly more than 100% of the available memory, forcing a GC to occur). This collects the 75% of available memory, but it also gives us a non-zero block of memory in return, meaning that we enter the next iteration of the loop with a non-zero percent of the available memory having been allocated. Thus the invariant has been broken.

Thus we must modify the attack in several ways. The resulting attack is shown in Figure 2.4.

First, we repeat the attack N (where $N \in [10, 20]$ is sufficient) times and measure each trial run. If the allocation duration is larger than some threshold, we store the time required to perform the allocation in the array times. By filtering out allocations with short allocation times we filter out iterations that does not lead to an invocation of the garbage collector.

Second, instead of allocating one array when filling up the memory, we allocate K arrays, where K is usually less than 10. This increases the evacuation time since more memory will need to be copied. Larger values of K will lead to a greater timing difference between a zero bit and a one bit, at the cost of a greater allocation time. Thus there is a time/precision trade off.

Third, we compute the average allocation time for each of the trial runs

which invoked the garbage collector. If the average garbage collection time is above some chosen Δ we conclude that the current bit is one, and zero otherwise.

Finally, in case no garbage collection occurs we retry the current iteration. Note that the probability of invoking a collection in subsequent tries is increased because more memory has been allocated from the previous tries of the same iteration, meaning that several tries of the same iteration is rare.

2.3.4 Results

The blue plot of Figure 2.5a shows the output of running the program described in Figure 2.4 on the secret input 5342121 with the serial garbage collection strategy used by Java when invoked with the parameter `-XX:+Use-SerialGC`. Similarly the red part of the figure shows the output of the same program, with modified constants, on the same secret input with the parallel collection strategy used when invoking Java with the parameter `-XX:+Use-ParallelGC`.¹

Figure 2.5b shows the output obtained by running a similar attack on the V8 JavaScript engine using Node.js. This attack follows the same pattern as the attack in Figure 2.4, and has therefore been omitted.

The experiment results are gathered from a machine with the following specs: Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz, Memory: 8 GB. NodeJS version 6.2.0. Java version “1.8.0_77”, Java(TM) SE Runtime Environment (build 1.8.0_77-b03), Java HotSpot(TM) 64-Bit Server VM (build 25.77-b03, mixed mode).

All figures show a clear distinction in garbage collection time consumption in the aftermath of processing a one bit, and processing a zero bit.

Observations over network The timing observations in these attacks are of sufficient magnitude to be observed over an internal network, e.g., in a datacenter-like setting, with ping latency of 0.5ms. Figure 2.5c shows the timing observed by a client communicating with a server over 25 trials, where each trial consists of the following operations:

1. first, the server sends a ping to the client
2. then, the server performs an allocation similar to the allocation on line 15 in Figure 2.2

¹The appendix contains a link to supplementary material with a VM containing all of our examples.

```

1  long[] times = new long[N];
2  int guess = 0;
3
4  for(int bit = 31; bit >= 0; --bit) {
5      for(int i = 0; i < N; ++i) {
6          int[][] a = new int[K][size];
7          int[][] b = null;
8          int[][] c = null;
9          int[][] d = null;
10         if (((secret >> bit) & 1) > 0) {
11             b = new int[K][size];
12             d = a;
13         }
14         else {
15             c = new int[K][size];
16             b = a;
17         }
18         c = null;
19         long before = System.nanoTime();
20         int[] c = new int[size2];
21         long after = System.nanoTime();
22         if(after - before > threshold) {
23             times[i] = after - before;
24         }
25         else {
26             times[i] = 0;
27         }
28     }
29
30     long sum = 0;
31     int numOfGCs = 0;
32     for(int i = 0; i < times.length; ++i) {
33         long t = times[i];
34         if(t != 0) {
35             sum += t;
36             ++numOfGCs;
37         }
38     }
39     if(numOfGCs == 0) {
40         ++bit;
41         continue;
42     }
43     if(sum / numOfGCs > Δ) {
44         guess += Math.pow(2, bit);
45     }
46 }

```

Figure 2.4: Program leaking 32 bits of information based on evacuation time.

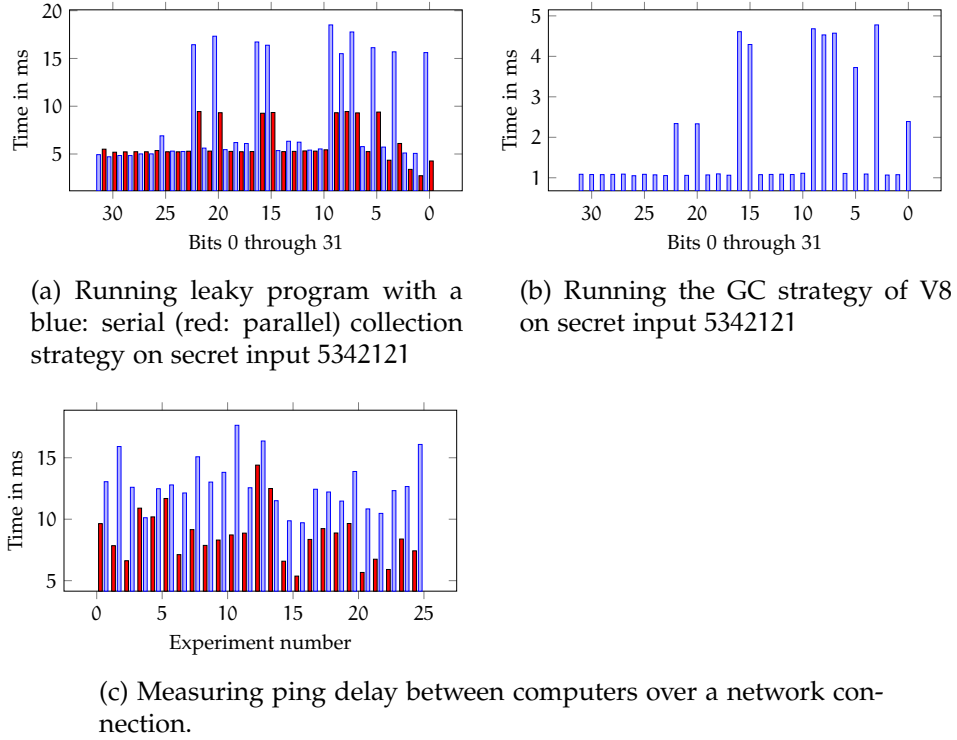


Figure 2.5: Experimental results for network ping delay with a leaky program running on the sample input $5342121 = (00000000\ 01010001\ 10000011\ 10101001)_2$

3. finally, the server sends another ping to the client.

Figure 2.5c thus shows the difference in the delay between the two pings sent by the server during step (1) and (3). The red bars show the delay when roughly half as much memory should be garbage collected.

Rate By measuring the execution time of the program in Figure 2.4 over 25 iterations we calculate the rate of the channel obtained. This yields a channel rate of 0.98 bytes per second.

In the sections to follow, we construct a formalism for studying these attacks. We first introduce a standard imperative language using a small-step semantics, with a few technical deviations to facilitate an isolated study of garbage collection. We extend the semantics to incorporate garbage collection transitions, and prove that our garbage collection semantics satisfies functional correctness. Finally we add a standard type system for information flow which, when combined with the semantics of Section 2.4.2, implies resilience against the presented attacks.

2.4 Language

This section presents a design of a small imperative programming language with automatic memory management. The key element of the design is that careful combination of the guarantees obtained via typing and the runtime constraints on the memory management eliminate timing leaks via garbage collection.

Syntax Figure 2.6 describes the syntax of our language. It is a standard imperative language [107, 133, 143] extended with heap allocated arrays and the corresponding getters and setters, a command for obtaining the current time, and the two security-related constructs, as explained below. For expressions, n ranges over the set of integers \mathbb{Z} and x, y, z range over variables. Finally, \oplus ranges over binary integer operations. A special expression `null` corresponds to the only memory location representable at the source level.

Non-standard features The runtime of the language has an explicit notion of time that may be observed programmatically using command $x := \text{time}()$. This particular design choice has the advantage that it provides a powerful attacker model without complicating the formal setup, e.g., introducing intermediate outputs. This includes a network attacker who observes timing of the network communication, as well an attacker providing untrusted code with access to a clock, which may in general be needed for functionality.

Our formal semantics (cf. Section 2.4.1) models the time using simple instruction counting – operational steps in the computation. This simplification is sufficient for our purposes of expressing the fundamental constraints necessary to eliminate timing leaks via memory management. Naturally, a realistic implementation would need to soundly relate the operational steps with the wall-clock, but that is outside of the scope of the current work.

In order to secure the behavior of the garbage collector at runtime, the language contains a notion of the *runtime program counter level*. The program counter is manipulated via two security-relevant constructs. Command `at ℓ with bound e do c` raises the program counter level, while command `restore ℓ` lowers it. Note that `restore` command never appears in the program source, but is explicitly inserted in the operational semantics to restore the level of the program counter level after an `at ℓ with bound e do c` command [81].

Another aspect of `at` and `restore` commands is that they implement light-weight predictive mitigation [14, 142, 143] of direct timing channels, i.e.,

$$\begin{aligned}
 e &::= n \mid \text{null} \mid x \mid e \oplus e \\
 c &::= \text{skip} \mid x := e \mid c; c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \\
 &\quad \mid x := \text{new}_\ell(e, e) \mid x := y[e] \mid x[e] := e \mid x := \text{time}() \\
 &\quad \mid \text{at } \ell \text{ with bound } e \text{ do } c \mid \boxed{\text{restore } \ell \text{ when } n}
 \end{aligned}$$

Figure 2.6: Syntax of the language. The boxed command is not part of the surface syntax.

channels that have control-flow representation, such as secret conditionals. An execution of statement at ℓ with bound n do c is padded to take exactly n steps. The execution is blocked if c takes more than n steps. This particular design aspect provides clear containment of direct timing channels, which in its turn allows isolated study of the leaks via memory management.

2.4.1 Semantic Environments

Our formal semantics partitions the program memory into memory environment m , which models variables that are typically allocated on stack, and the heap environment h .

Values, variables, and locations A value in the language is either an integer or a location in the heap, including null. For our purposes it is sufficient to leave heap locations abstract, and our model simply assumes a set Loc of abstract heap locations that may be distinguished from each other. We further assume that the set of locations is disjoint from the set of integer values. Let v range over values Val , and denote the set of all variables as Var .

Memory and heap environments Memory environment is a partial function $m : \text{Var} \rightarrow \text{Val}$. For convenience, we also use the set notation, and write $(x, v) \in m$ when $m(x) = v$.

For heap environment, an important characteristic of our model is that allocations on the heap are tagged with security levels. This is necessary for constraining collecting behavior, as explained later in this section.

We define a heap as a partial function $h : \text{Loc} \rightarrow (\mathbb{N} \rightarrow \text{Val}) \times \mathcal{L}$ from heap locations to pairs consisting of *lookup functions* and security levels. A lookup function is a partial function from the integer offsets to values stored in the heap. We write $(l, \mu, \ell) \in h$ when $h(l) = (\mu, \ell)$, where l is the abstract location, μ is the lookup function, and ℓ is the security level. We refer to ℓ as the *heap level* of l . When heap h can be split into two disjoint heaps h_1 and h_2 , we write $h = h_1 \uplus h_2$.

Maximum heap size Our model considers both unbounded and bounded heaps. The semantics is parametrized with the function that returns the maximum size of the heap $\text{mx} : \mathcal{L} \rightarrow \mathbb{N} \cup \{\infty\}$ that returns the maximum amount of heap memory available for allocation at that security level. Unless explicitly specified, the formal results apply to both bounded and unbounded heaps.

When the current size of the heap h at a particular level ℓ needs to be compared against the maximum available space, we compute the current size using function $\text{size}_\ell(h) \triangleq \sum_{(l, \mu, \ell) \in h} |\text{dom}(\mu)|$.

2.4.2 Semantics without Garbage Collection

We start by introducing the semantics of the language without garbage collection. Section 2.5 presents the semantics of garbage collection.

Semantic configurations have the form $\langle c, pc, m, h, t \rangle$, where c is the current program, pc is the runtime program counter level, m and h are the memory and the heap respectively, and t is the time counter. Terminal configurations are marked by the dedicated stop command. The semantics is a combination of a standard big-step evaluation relation for expressions, and a small-step transition relation for commands.

Expressions

Figure 2.7 presents the evaluation relation $\langle x, m \rangle \Downarrow v$ for expression semantics. Note how this relation only includes standard memory; all heap-related operations are modeled as commands.

Commands

Figures 2.8 and 2.9 present the semantics of commands in the absence of garbage collection. None of the standard commands touch the heap or the program counter level; moreover they all take one computation step. This semantics is given by rule (S-LIFT-STANDARD) in Figure 2.9 together with the standard transition relation $\langle c, m \rangle \rightarrow \langle c', m' \rangle$, defined in Figure 2.8.

The remaining of the rules in Figure 2.9 present the transition relation for the and non-standard commands.

Rule (S-TIME) updates variable x with the current value of the time counter.

Rule (S-NEW) models the allocation in partition ℓ . The amount of allocated memory is computed by evaluating expression e . The command extends the heap with the new lookup function μ , and updates the variable in the memory with the value of the new location. The lookup function μ uses

$$\begin{array}{c}
\frac{}{\langle n, m \rangle \Downarrow n} \qquad \frac{}{\langle \text{null}, m \rangle \Downarrow \text{null}} \qquad \frac{m(x) = v}{\langle x, m \rangle \Downarrow v} \\
\\
\frac{\langle e_i, m \rangle \Downarrow v_i \quad i = 1, 2 \quad v_1 \oplus v_2 = v}{\langle e_1 \oplus e_2, m \rangle \Downarrow v}
\end{array}$$

Figure 2.7: Semantics of expressions

the default value computed by evaluating expression e_{init} . The rule has two notable constrains. First, the location must be fresh, expressed by the premise $l \notin \text{dom}(h)$. Second, there must be enough available space in the heap at partition ℓ for the allocation, which is expressed by the premise $\text{size}_\ell(h') \leq \text{mx}(\ell)$, where h' refers to the heap updated after the allocation.

Rule (S-SET) updates the array at a specified index. This is expressed as in-place update of the lookup function. Rule (S-GET) retrieves the value stored at a particular index in an array; the result is stored in the memory.

Rule (S-AT) updates the program counter label. Additionally, this rule computes the time n the at command is expected to consume. This rule inserts a restore command that restores the program counter, and the expected time $t + n$ by which the restore should be reached.

Rule (S-RESTORE-PROGRESS) restores the program counter label and continues with the execution of the body of the restore command only if the current time matches the expected time specified in the restore command. Rule (S-RESTORE-WAIT) skips until the current time matches the argument of the restore command. Note that if the body of at happens to take more time than expected, the semantics blocks the execution.

2.5 Semantics for Secure Garbage Collection

The previous section defined the program semantics without garbage collection transitions, i.e., the size of the heap would monotonically increase throughout the execution. This section defines the collection semantics that specifies how garbage collection is allowed to affect the heap.

The main insight is that the runtime program counter label constrains which parts of the heap can be collected and when. When the runtime program counter is low, only low parts of the heap can be collected; when the runtime program counter is high, only high parts of the heap can be collected. While, on first sight, this isolation-like constrain may appear unnecessarily strong

$$\begin{array}{c}
\frac{}{\langle \text{skip}, m \rangle \rightarrow \langle \text{stop}, m \rangle} \qquad \frac{\langle e, m \rangle \Downarrow v}{\langle x := e, m \rangle \rightarrow \langle \text{stop}, m[x \mapsto v] \rangle} \\
\\
\frac{\langle c_1, m \rangle \rightarrow \langle c'_1, m' \rangle \quad c'_1 \neq \text{stop}}{\langle c_1; c_2, m \rangle \rightarrow \langle c'_1; c_2, m' \rangle} \qquad \frac{\langle c_1, m \rangle \rightarrow \langle \text{stop}, m' \rangle}{\langle c_1; c_2, m \rangle \rightarrow \langle c_2, m' \rangle} \\
\\
\frac{\langle e, m \rangle \Downarrow n \quad n \neq 0 \implies i = 1 \quad n = 0 \implies i = 2}{\langle \text{if } e \text{ then } c_1 \text{ else } c_2, m \rangle \rightarrow \langle c_i, m \rangle} \\
\\
\frac{\langle e, m \rangle \Downarrow n \quad n \neq 0}{\langle \text{while } e \text{ do } c, m \rangle \rightarrow \langle c; \text{while } e \text{ do } c, m \rangle} \qquad \frac{\langle e, m \rangle \Downarrow n \quad n = 0}{\langle \text{while } e \text{ do } c, m \rangle \rightarrow \langle \text{stop}, m \rangle}
\end{array}$$

Figure 2.8: Semantics of standard commands

in an information-flow setting, it is necessary, because the garbage collection represents a bi-directional information-flow channel. We explain this using two simple examples inspired by our experiments from Section 2.3.

2.5.1 Motivating Security Restrictions on Garbage Collection

This section presents two examples that motivate our restrictions on garbage collection semantics. Each of the examples demonstrates the danger of collecting parts of the memory that do not match the current program counter level. Note that while the examples are written in the style that follows our typing discipline of Section 2.6, the typing is not required here.

Implicit Flows when Collecting L in H

Consider program below, where we assume that N and M are constants, and v is picked sufficiently pessimistically to bound the execution of the `at` command.

```

1 // new array of size N with default element value 0
2 y := newL(N, 0);
3 y := null;
4 // the y-array can now be reclaimed
5 at H with bound v do
6   if h > 0 then
7     // new array of size M that requires GC
8     x := newH(M, 0)
9   else skip;
```

$$\begin{array}{c}
\text{S-LIFT-STANDARD} \\
\frac{\langle c, m \rangle \rightarrow \langle c', m' \rangle}{\langle c, pc, m, h, t \rangle \xrightarrow{\text{mx}} \langle c', pc, m', h, t + 1 \rangle} \\
\\
\text{S-TIME} \\
\frac{}{\langle x := \text{time}(), pc, m, h, t \rangle \xrightarrow{\text{mx}} \langle \text{stop}, pc, m[x \mapsto t], h, t + 1 \rangle} \\
\\
\text{S-NEW} \\
\frac{\langle e, m \rangle \Downarrow n \quad l \notin \text{dom}(h) \quad \text{size}_\ell(h') \leq \text{mx}(\ell) \quad \langle e_{\text{init}}, m \rangle \Downarrow v \quad h' = h[l \mapsto (\mu, \ell)] \quad \mu(x) = \begin{cases} v & 0 \leq x < n \\ \text{undef} & \text{otherwise} \end{cases}}{\langle x := \text{new}_\ell(e, e_{\text{init}}), pc, m, h, t \rangle \xrightarrow{\text{mx}} \langle \text{stop}, pc, m[x \mapsto l], h', t + 1 \rangle} \\
\\
\text{S-SET} \\
\frac{\langle e_1, m \rangle \Downarrow n \quad \langle e_2, m \rangle \Downarrow v \quad 0 \leq n < |\text{dom}(\mu)| \quad l = m(x) \quad (\mu, \ell) = h(l) \quad \mu'(x) = \begin{cases} v & x = n \\ \mu(x) & \text{otherwise} \end{cases}}{\langle x[e_1] := e_2, pc, m, h, t \rangle \xrightarrow{\text{mx}} \langle \text{stop}, pc, m, h[l \mapsto (\mu', \ell)], t + 1 \rangle} \\
\\
\text{S-GET} \\
\frac{\langle e, m \rangle \Downarrow n \quad l = m(y) \quad \mu = h(l) \quad v = \mu(n)}{\langle x := y[e], pc, m, h, t \rangle \xrightarrow{\text{mx}} \langle \text{stop}, pc, m[x \mapsto v], h, t + 1 \rangle} \\
\\
\text{S-AT} \\
\frac{\langle e, m \rangle \Downarrow n \quad t' = t + n}{\langle \text{at } \ell \text{ with bound } e \text{ do } c, pc, m, h, t \rangle \xrightarrow{\text{mx}} \langle c; \text{restore } pc \text{ when } t', \ell, m, h, t + 1 \rangle} \\
\\
\text{S-RESTORE-PROGRESS} \\
\frac{t = t'}{\langle \text{restore } \ell \text{ when } t', pc, m, h, t \rangle \xrightarrow{\text{mx}} \langle \text{stop}, \ell, m, h, t + 1 \rangle} \\
\\
\text{S-RESTORE-WAIT} \\
\frac{t < t'}{\langle \text{restore } \ell \text{ when } t', pc, m, h, t \rangle \xrightarrow{\text{mx}} \langle \text{restore } \ell \text{ when } t', pc, m, h, t + 1 \rangle}
\end{array}$$

Figure 2.9: Semantics of heap and non-standard commands

```

10 t1 := time();
11 // GC time depends on whether array y
12 // has been collected earlier
13 y := newL(N, 0);
14 t2 := time();
15 low := t2 - t1

```

Note how the high conditional is guarded by the `at` command that ensures that the execution of the conditional takes v steps. This means that the value of t_1 does not depend on which branch of the conditional is taken.

However, if semantics of garbage collection allows low parts to be collected inside `at`, say before executing the allocation on Line 8, then $t_2 - t_1$ is likely to be small. This motivates that garbage collection should not collect low allocations when the program counter level is high.

Implicit Flows when Collecting **H** in **L**

This example shows that collecting high allocations when the program counter is low is also dangerous. Suppose we are given constants M , N , and v as described earlier, and consider program below.

```

1 // new array of size M with
2 // default element value 0
3 x := newH(M, 0);
4 at H with bound v do
5   if h > 0 then
6     x := null
7     // array x can now be reclaimed
8   else skip;
9 t1 := time();
10 // GC time depends on whether array x is reclaimable
11 y := newL(N, 0);
12 t2 := time();
13 low := t2 - t1

```

As before, the timing of the high conditional is protected with an `at` command. Consider allocation on Line 11 that may trigger garbage collection. If semantics of the GC allows collecting high allocation in the low program counter, the amount of time that the collector spends here will depend on whether the array x can be reclaimed, affecting the value of t_2 . This motivates that garbage collection should not collect high allocations when the program counter is low.

2.5.2 Formal Semantics for Garbage Collection

Using the above examples as guideline, we now formulate the formal semantics for secure garbage collection. The rule for garbage collection is given by transition relation $\langle c, pc, m, h, t \rangle \dashrightarrow \langle c, pc, m, h', t' \rangle$ that relates two configurations before and after collection. Collection does not update the current command or memory or the program counter level, but updates the heap and consumes some time.

Figure 2.10 presents the collection rule formally. To explain the rule, we introduce the auxiliary concepts that it uses.

Abstract Collection Relation

The amount of time consumed by the collection is in general implementation-specific. We require the implementation to provide an interface for collecting specific *parts* of the heap. We model this by an abstract relation $h \rightsquigarrow_{\delta}^m h'$, where h is a subheap, and where h' is the result of collection in h that takes time δ , given memory m . An important constraint that we place on the \rightsquigarrow relation is that if two subheaps and starting memories are *isomorphic*, then it must take the same amount of time to collect in them. To formally express this, we introduce the notion of substitutions. This notion of substitution is closely related to the one by Banerjee and Naumann [17]. As we will see, the substitution is also used later in the paper when defining our GC requirement.

Definition 2.5.1. (*Substitution*) A substitution $\phi : \text{Val} \rightarrow \text{Val}$ is a partial mapping such that

1. ϕ is identity on integers: $\forall n . \phi(n) = n$.
2. ϕ is injective on locations: $\forall l, l' . \phi(l) = \phi(l') \implies l = l'$.
3. ϕ maps locations to locations: $\forall l \in \text{Loc} . \phi(l) \in \text{Loc}$.

Given a substitution ϕ , we write $\phi(m) = \{(\chi, \phi(v)) \mid (\chi, v) \in m\}$ and $\phi(h) = \{(\phi(l), \phi \circ \mu, \ell) \mid (l, \mu, \ell) \in h\}$. For the remaining parts of the paper we say substitution to mean a bijective substitution. That is a substitution ϕ such that

$$\forall v, v' . \phi(v) = v' \iff \phi^{-1}(v') = v.$$

We write $h \cong_{\phi} w$ (resp. $(m, h) \cong_{\phi} (s, w)$) when $\phi(h) = w$ (resp. $(\phi(m), \phi(h)) = (s, w)$) and $h \cong w$ (resp. $(m, h) \cong (s, w)$) when there exists a bijection ϕ such that $h \cong_{\phi} w$ (resp. $(\phi(m), \phi(h)) = (s, w)$). Two heaps are then isomorphic when $h \cong w$.

$$\begin{array}{c}
\text{GC-COLLECT} \\
\frac{\text{reach}(m, h_1 \uplus h_2) \cap \text{dom}(h_2) = \emptyset \quad h_2^{\neq pc} = \emptyset \quad h_1 = h_1^{\neq pc} \uplus h_1^{\neq pc} \quad h_1^{\neq pc} \uplus h_2 \rightsquigarrow_{\delta}^m h_1^{\neq pc}}{\langle c, pc, m, h_1 \uplus h_2, t \rangle \dashrightarrow \langle c, pc, m, h_1, t + \delta \rangle}
\end{array}$$

Figure 2.10: Reduction rule for garbage collection

Using the notion of substitution, we formulate our assumption on the abstract collection relation.

Assumption (Abstract collection). *Consider two memories m and s and two heaps h and w and a substitution ϕ such that $(s, w) = (\phi(m), \phi(h))$. Then $h \rightsquigarrow_{\delta}^m h'$ implies $w \rightsquigarrow_{\delta}^s w'$ for some w' .*

If two heaps are isomorphic they are equal up to renaming of locations, and the specific names of locations should not affect the behaviour of garbage collection.

Level-based Heap Partitioning

Given a heap h and a level ℓ , write $h^{\neq \ell}$ for the heap that includes all allocations tagged with security level ℓ :

$$h^{\neq \ell} \triangleq \{(x, \mu, \ell') \in h \mid \ell' = \ell\}.$$

and similarly, define the complement partition as

$$h^{\neq \ell} \triangleq \{(x, \mu, \ell') \in h \mid \ell' \neq \ell\}.$$

Given a level ℓ , any heap h can be decomposed into a disjoint union of its partition and its complement: $h = h^{\neq \ell} \uplus h^{\neq \ell}$.

Reachable Locations

Since deciding whether or not a location will be accessed in the future is undecidable [78], we follow real world implementations of garbage collection schemes, and treat a variable as live if it is reachable from the current set of variables in the program. The set of reachable values is then the values that are pointed to by variables in the memory, or by following a chain of reachable locations on the heap.

Definition 2.5.2 (Reachable locations). *Given a memory m and a heap h , the set of reachable locations $\text{reach}(m, h) \subseteq \text{Loc}$ is the smallest set such that*

1. all locations in memory m are reachable:

$$\text{cod}(m) \cap \text{Loc} \subseteq \text{reach}(m, h)$$

2. if $l \in \text{reach}(m, h)$ and $h(l) = (\mu, \ell)$ then locations that the allocation μ points to are reachable:

$$\text{cod}(\mu) \cap \text{Loc} \subseteq \text{reach}(m, h)$$

Note that reach is monotonic, as expressed by the following lemma.

Lemma 2.5.1 (Monotonicity of heap reachability). *Given two heaps h and w , if $h \subseteq w$ then $\text{reach}(m, h) \subseteq \text{reach}(m, w)$.*

With these definitions at hand, let us examine the garbage collection rule in Figure 2.10. The rule is defined when the program heap is split in two disjoint heaps h_1 and h_2 , where h_2 is collected after the transition. That it is functionally safe to collect h_2 is ensured by the first premise of the rule that stipulates that no location in h_2 is reachable from the current configuration. The remaining premises induce security restrictions on the collection. We restrict h_2 to only contain allocations that are exactly at the level of the program counter level pc – this is expressed by the requirement $h_2^{\neq pc} = \emptyset$, which could alternatively be stated as $h_2^{\neg pc} = h_2$. The idea is to constrain collections at a specific level only when the program counter matches that level. The rule further splits the non-collectable heap h_1 into two parts, based on the security level: the pc -part $h_1^{\neg pc}$ and its complement h_1^{pc} . Only the pc -part of the heap is used when invoking the abstract collector.

2.5.3 Functional Correctness

The remaining part of this section shows that our garbage collection strategy is functionally correct. That is, the collector never claims memory that is accessed in the future. We start with a formal definition of dangling pointer-freedom.

Definition 2.5.3 (Dangling pointer-freedom). *Given a memory m and heap h , say that (m, h) is free of dangling pointers when*

1. all locations in the memory point to a valid location in the heap:

$$\text{cod}(m) \cap \text{Loc} \subseteq \text{dom}(h)$$

2. pointers within heap are valid:

$$\forall (\mu, \ell) \in \text{cod}(h) . \text{cod}(\mu) \cap \text{Loc} \subseteq \text{dom}(h)$$

Informally, if m and h do not contain dangling pointers, then extending the heap does not increase reachability.

Lemma 2.5.2. *Given memory m and h such that (m, h) is free of dangling pointers then for all heaps $w \supseteq h$, it holds that $\text{reach}(m, h) = \text{reach}(m, w)$.*

For functional correctness, we show that adding the garbage collection rule in Figure 2.10 to the set of possible transitions does not modify the memory, nor the reachable part of the heap.

There is a technical challenge to overcome. The specific location allocated in the heap, and stored in the memory, depends on the current size of the heap (cf. the semantics in Figure 2.9). Thus by adding a garbage collector, which reduces the size of the heap, the locations allocated and stored in the memory will be different from the locations allocated without first reducing the size of the heap.

We therefore prove that the reachable parts of the heaps will be equal up to renaming of locations. We use the notion of substitution introduced earlier in Section 2.5.2 to relate pairs of memories and heaps that have isomorphic reachable parts but may have different amounts of garbage.

Definition 2.5.4 (Matching up to garbage). *Consider two pairs of memory and heaps (m, h) and (s, w) . Say that (s, w) matches (m, h) up to garbage via substitution ϕ , written $(m, h) \simeq_\phi (s, w)$ if $\text{reach}(\phi(m), \phi(h)) = \text{reach}(s, w)$.*

We write $(m, h) \simeq (s, w)$ if there exists ϕ s.t. $(m, h) \simeq_\phi (s, w)$.

The following lemma states that the semantics does not depend on garbage.

Let $\omega(\ell) = \infty$ denote the constant ∞ function. The relation $\xrightarrow{\omega}$ defines the semantics in an abstract setting with unbounded available memory.

Lemma 2.5.3 (Garbage independence in unbound heaps). *For all heaps h and w such that $w \supseteq h$, and (m, h) and (m, w) are free of dangling pointers it holds that if*

$$\langle c, pc, m, h, t \rangle \xrightarrow{\omega} \langle c', pc', m', h', t' \rangle$$

then $\langle c, pc, m, w, t \rangle \xrightarrow{\omega} \langle c', pc', s', w', t' \rangle$ and $(m', h') \simeq (s', w')$.

The next lemma states that garbage collection only collects garbage from the heap. That is, if garbage collection collects some portion of the heap, then succeeding transitions do not depend on the portion of the heap that has been collected.

Lemma 2.5.4 (Garbage only). *If (m, h) is free of dangling pointers and*

$$\langle c, pc, m, h, t \rangle \xrightarrow{mx} \langle c', pc', m', h', t' \rangle$$

then for all s, w such that (s, w) is free of dangling pointers, and $(s, w) \simeq (m, h)$ we have

$$\langle c, pc, s, w, t \rangle \xrightarrow{mx} \langle c', pc', s', w', t' \rangle.$$

and $(s', w') \simeq (m', h')$.

We can now state functional correctness of the garbage collection scheme. Intuitively, the reachable heap does not change when interleaving the reductions with garbage collections. This is expressed as a pair of theorems, in the style of the work by Morrisett et al. [78]. The first theorem states that running the GC followed by a regular transition is comparable to running a regular transition. The second one states that running a regular transition is comparable to running the GC followed by a regular transition.

Note that in the statements of the theorems below the time component of the configurations is reset. This ensures that the result of running command $x := \text{time}()$ is the same. Furthermore, the first theorem additionally qualifies the semantics to be unbounded in maximum available size. This is needed because otherwise the execution may run out of available heap.

Theorem 2.5.5 (Functional correctness for unbound heaps with time reset). *Consider memory m and heaps h and w such that (m, h) and (m, w) are free of dangling pointers. If $\langle c, pc, m, h, t \rangle \dashrightarrow \langle c, pc, m, w, t'' \rangle$ and $\langle c, pc, m, w, t \rangle \xrightarrow{\omega} \langle c', pc', m', w', t' \rangle$ then $\langle c, pc, m, h, t \rangle \xrightarrow{\omega} \langle c', pc', s', h', t' \rangle$ and $(m', w') \simeq (s', h')$.*

Theorem 2.5.6 (Functional correctness with time reset). *Consider memory m and heaps h such that (m, h) is free of dangling pointers. If $\langle c, pc, m, h, t \rangle \xrightarrow{mx} \langle c', pc', m', h', t' \rangle$ then $\langle c, pc, m, h, t \rangle \dashrightarrow \langle c, pc, m, w, t'' \rangle$ and $\langle c, pc, m, w, t \rangle \xrightarrow{mx} \langle c', pc', s', w', t' \rangle$ and $(s', w') \simeq (m', h')$.*

2.5.4 GC or Normal Steps

As a final element in this section, we define a top-level GC or normal step as a transition function that nondeterministically interleaves normal and collection steps.

$$\begin{array}{c} \text{NORMAL-STEP} \\ \frac{\langle c, pc, m, h, t \rangle \xrightarrow{mx} \langle c', pc', m', h', t' \rangle}{\langle c, pc, m, h, t \rangle \xrightarrow{mx} \langle c', pc', m', h', t' \rangle} \end{array} \quad \begin{array}{c} \text{GC-STEP} \\ \frac{\langle c, pc, m, h, t \rangle \dashrightarrow \langle c, pc, m, h', t' \rangle}{\langle c, pc, m, h, t \rangle \xrightarrow{mx} \langle c, pc, m, h', t' \rangle} \end{array}$$

We use this top-level relation in studying security properties of our programs in Section 2.7.

2.6 Type System

In addition to the secure garbage collection described in the previous section, our enforcement mechanism additionally relies on a typing discipline. The typing discipline is mostly standard for an imperative security-typed language with arrays [33, 34, 83, 103, 130], with a few minor technical deviations that we explain below. In particular, the type system ensures not only confidentiality, but also integrity by viewing values that depend on time commands as tainted. This restricts the extent to which the result of the time command affects the control flow or the heap shape of the program.

2.6.1 Time Taint and Generalized Security Levels

We introduce a *time lattice*; a two-point lattice with the elements \circ and \bullet . Here, \circ corresponds to *untainted* values, and \bullet corresponds to *tainted* values. We let ι range over elements of this lattice, and define an ordering \sqsubseteq^t such that for all $\iota \in \{\circ, \bullet\}$ it holds that $\iota \sqsubseteq^t \iota$, and $\circ \sqsubseteq^t \bullet$. We define the corresponding least upper bound operator as \sqcup^t .

A *generalized security level* ξ is a combination of both a confidentiality level ℓ and a taint level ι . With this, we have the following grammar for the security levels in the type system.

$$\begin{aligned}\iota &::= \circ \mid \bullet \\ \xi &::= (\ell, \iota)\end{aligned}$$

We lift lattice operations to generalized security levels, and denote the resulting ordering and least upper bound operations as \preceq and γ , respectively.

Types are given by the following grammar

$$\begin{aligned}\sigma &::= \mathbf{int} \mid \mathbf{array}_\ell[\tau] \\ \tau &::= \sigma \xi\end{aligned}$$

Here, τ is a security annotated type that consist of a base type with a security level. Base types σ are either integers or arrays of some type τ that specify the confidentiality level of the partition where the array lives.

Given a base type σ and security levels ξ_1, ξ_2 , define the operator for raising of the type $\sigma \xi_1$ to level ξ_2 as

$$(\sigma \xi_1)^{\xi_2} \triangleq \sigma (\xi_1 \gamma \xi_2)$$

The lattice ordering \preceq induces a subtyping relation on the types

$$\frac{\xi_1 \preceq \xi_2 \quad \sigma_1 = \sigma_2}{\sigma_1 \xi_1 <: \sigma_2 \xi_2}$$

$$\begin{array}{c}
 \text{WF-INT} \\
 \hline
 \ell \vdash_{\text{wf}} \mathbf{int} \xi
 \end{array}
 \qquad
 \begin{array}{c}
 \text{WF-ARRAY} \\
 \hline
 \ell \sqsubseteq \ell_p \quad \ell_{\text{ref}} \sqsubseteq \ell_p \quad \ell_p \vdash_{\text{wf}} \tau \\
 \hline
 \ell \vdash_{\text{wf}} \mathbf{array}_{\ell_p} [\tau] (\ell_{\text{ref}}, \circ)
 \end{array}$$

Figure 2.11: Type well-formedness

Note the invariance in the base types, even if the base type is $\mathbf{array}_{\ell_p} [\tau] (\ell_{\text{ref}}, \circ)$, is required because the arrays are mutable [82].

2.6.2 Well-Formedness of Reference Types

Figure 2.11 presents well-formedness conditions of types w.r.t. references. The security level on the left-hand side of the turnstile is a lower bound on the heap level that can store values of type τ . A type τ is well-formed when $\perp \vdash_{\text{wf}} \tau$. These rules prevent creation of references from the high partitions into the low ones, and are later lifted to define well-formedness of configurations.

2.6.3 Typing Rules

We assume a memory typing environment Γ that maps variable names to types. In the remaining parts of the paper, we require that the memory typing environments are well-formed w.r.t. all types defined in it. The typing judgment for expressions has form $\Gamma \vdash e : \tau$. Figures 2.12 presents the typing rules for expressions.

The typing judgment for commands has form $\Gamma, pc \vdash c$. Figure 2.14 present the typing rules for commands, where pc is the static program counter level. The rule (T-SKIP) is trivial. Rule (T-ASSIGN) is standard in how it prevents both implicit and explicit information flows using the program counter level. The rule (T-TIME) requires the assigned variable to be marked as tainted, and is otherwise similar to assignment in its treatment of implicit flows. Rule (T-IF) is slightly non-standard. First, it prevents branching on high data if the pc is low. Note that the rule does not raise the level of the program counter label in the branches. Instead, high conditionals must occur syntactically in a scope where the pc -level is explicitly raised using `at` command. Second, branching on high values is allowed only if the value is not tainted by time commands. This is visualized in Figure 2.13. Rule (T-WHILE) imposes a similar restriction, and is otherwise standard.

Rule (T-NEW) requires that both pc -level and the level of the expression that determines the size of the array flow to the variable that stores the reference.

$\frac{}{\Gamma \vdash n : \mathbf{int} \xi}$ <p>T-INT</p>	$\frac{\perp \vdash_{wf} \mathbf{array}_\ell[\tau](\ell_{ref}, \circ)}{\Gamma \vdash \mathbf{null} : \mathbf{array}_\ell[\tau](\ell_{ref}, \circ)}$ <p>T-NULL</p>	$\frac{}{\Gamma \vdash x : \Gamma(x)}$ <p>T-VAR</p>
$\frac{\Gamma \vdash e_i : \mathbf{int} \xi_i \quad i = 1, 2}{\Gamma \vdash e_1 \oplus e_2 : \mathbf{int}(\xi_1 \vee \xi_2)}$ <p>T-OP</p>		

Figure 2.12: Typing rules for expressions

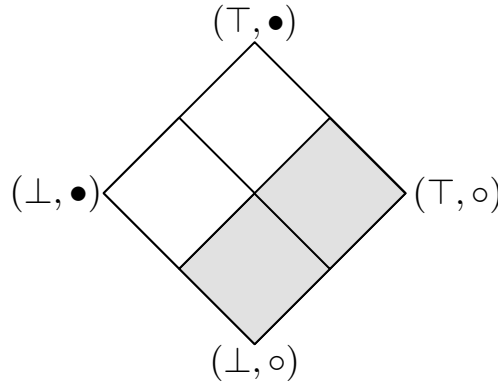


Figure 2.13: Lattice for confidentiality and integrity. Branching and heap manipulation is allowed on values whose type is in the gray area.

This prevents the size of a high array from depending on low values. The level ℓ on the command that is interpreted by the allocation semantic (cf. Rule (S-New) in Section 2.5) must be as restrictive as the level ℓ_2 of the reference.

Rule (T-SET) requires that both the pc -level and the expression used for indexing flow to the level of the array reference. This prevents indexing into a low array using high expressions. It also requires the type of the right-hand side expression to flow to the type of the array on the left, taking implicit flows via pc -level into account.

Rule (T-GET) is similar. It requires that the pc -level and the level of the index expression flows to the level of the array reference, and rules out both explicit and implicit flows in the assignment.

Finally, rules (T-AT) raises the level of pc . Furthermore, an explicit time bound is provided for this command, which allows the programmer to control the time consumed by commands when the pc is high.

$$\begin{array}{c}
\text{T-SKIP} \\
\hline
\Gamma, pc \vdash \text{skip}
\end{array}
\qquad
\begin{array}{c}
\text{T-ASSIGN} \\
\hline
\frac{\Gamma \vdash e : \sigma \xi \quad \sigma(\xi \vee (pc, \circ)) <: \Gamma(x)}{\Gamma, pc \vdash x := e}
\end{array}$$

$$\begin{array}{c}
\text{T-IF} \\
\hline
\frac{\Gamma \vdash e : \mathbf{int}(\ell, \circ) \quad \ell \sqsubseteq pc \quad \Gamma, pc \vdash c_i \quad i = 1, 2}{\Gamma, pc \vdash \text{if } e \text{ then } c_1 \text{ else } c_2}
\end{array}
\qquad
\begin{array}{c}
\text{T-WHILE} \\
\hline
\frac{\Gamma \vdash e : \mathbf{int}(\ell, \circ) \quad \ell \sqsubseteq pc \quad \Gamma, pc \vdash c}{\Gamma, pc \vdash \text{while } e \text{ do } c}
\end{array}
\qquad
\begin{array}{c}
\text{T-SEQ} \\
\hline
\frac{\Gamma, pc \vdash c_i \quad i = 1, 2}{\Gamma, pc \vdash c_1; c_2}$$

$$\begin{array}{c}
\text{T-NEW} \\
\hline
\frac{\Gamma \vdash e_{size} : \mathbf{int}(\ell_{size}, \circ) \quad \Gamma \vdash e_{init} : \tau \quad \Gamma(x) = \mathbf{array}_{\ell}[\tau](\ell_x, \circ) \quad pc \sqcup \ell_{size} \sqsubseteq \ell_x}{\Gamma, pc \vdash x := \text{new}_{\ell}(e_{size}, e_{init})}
\end{array}$$

$$\begin{array}{c}
\text{T-SET} \\
\hline
\frac{\Gamma \vdash e_{idx} : \mathbf{int} \xi_{idx} \quad \Gamma \vdash e : \sigma \xi \quad \Gamma(x) = \mathbf{array}_{\ell}[\tau] \xi_x \quad \xi_{idx} \vee (pc, \circ) \preceq \xi_x \quad \sigma(\xi_x \vee \xi) <: \tau}{\Gamma, pc \vdash x[e_{idx}] := e}
\end{array}$$

$$\begin{array}{c}
\text{T-GET} \\
\hline
\frac{\Gamma \vdash e_{idx} : \mathbf{int} \xi_{idx} \quad (pc, \circ) \vee \xi_{idx} \preceq \xi_y \quad \Gamma(y) = \mathbf{array}_{\ell}[\sigma \xi] \xi_y \quad \sigma(\xi \vee \xi_y) <: \Gamma(x)}{\Gamma, pc \vdash x := y[e]}
\end{array}
\qquad
\begin{array}{c}
\text{T-TIME} \\
\hline
\frac{\Gamma(x) = \mathbf{int}(\ell, \bullet) \quad pc \sqsubseteq \ell}{\Gamma, pc \vdash x := \text{time}()}$$

$$\begin{array}{c}
\text{T-AT} \\
\hline
\frac{\Gamma \vdash e : \mathbf{int}(\ell', \circ) \quad \ell' \sqsubseteq pc \sqsubseteq \ell \quad \Gamma, \ell \vdash c}{\Gamma, pc \vdash \text{at } \ell \text{ with bound } e \text{ do } c}
\end{array}$$

Figure 2.14: Typing rules for the surface language commands

Because the information flow constraints imposed by the type system are standard, they can also be enforced using dynamic or hybrid monitors [15, 106].

Note that the type system permits time measurements in both low and high context; this directly models attacker capability to make internal timing measurements (cf. Section 2.3); weaker attacker models, i.e., the ones where attacker does not have access to system clock but only to network messages, can be addressed in a similar manner.

Properties of the Type System

The type system ensures two important properties. To state these, we need a heap typing environment [126] that maps allocated locations to types.

Definition 2.6.1 (Heap typing environment). *A heap typing is a partial function $\Sigma : \text{Loc} \rightarrow \tau$ that maps heap locations to their types.*

The intuition for Σ is that given a location l , allocated by a command $x := \text{new}_\ell(e, e_{\text{init}})$, where $\Gamma \vdash e_{\text{init}} : \tau$, we have $\Sigma(l) = \tau$. Similarly to memory typing environments, we assume that types defined by the heap typing environment are well-formed.

We can now state the first property, which we split into two sub-properties: one for typing environments Γ , and one for heap typing environments Σ .

First, the typing environment Γ gives us an adequate view of the heap level of locations. More specifically, if a variable x points to a location l , then the heap level of l equals to the partition level specified by $\Gamma(x)$.

Similarly, we can state this property for a heap typing environment Σ . Let l be a location. Then $\Sigma(l)$ records the type of the *content* stored at l . So, if $\Sigma(l) = \mathbf{array}_{\ell_p}[\tau] (\ell_{\text{ref}}, \circ)$ it means that the content of location l has a heap level of ℓ_p . That is, following l twice leads to a heap allocation with a heap level equal to ℓ_p .

The second property is that the type system prevents creating pointers from high heap levels into the low heap levels. This property is important because a pointer from a high heap level to a low heap level would allow modifying low heap level pointers in a high program context.

Definition 2.6.2 (Memory and heap well-formedness). *Given a memory m , a heap h , an typing environment Γ and a heap typing environment Σ we say that (m, h) is well-formed wrt. (Γ, Σ) if*

- 1a) For all variables x s.t. $\Gamma(x) = \mathbf{array}_{\ell_p}[\tau](\ell_{ref}, \circ)$, $m(x) = l$, and $h(l) = (\ell, \mu)$, it holds that $\ell = \ell_p$.
- 1b) For all locations l s.t. $\Sigma(l) = \mathbf{array}_{\ell_p}[\tau](\ell_{ref}, \circ)$, $h(l) = (\ell, \mu)$, $\mu(n) = l'$, and $h(l') = (\ell', \nu)$ it holds that $\ell' = \ell_p$.
- 2) If $h(l_1) = (\ell_1, \mu_1)$, and $\mu_1(n) = l_2$ for some $n \in \mathbb{N}$, and $h(l_2) = (\ell_2, \mu_2)$ then $\ell_1 \sqsubseteq \ell_2$.

We define well-formed configurations to be configurations in which the command is well-typed, the memory and heaps are heap level bound and are free of dangling pointers. Finally we also capture the intuition about the relation between Γ and Σ : That Σ contains the type of the “content” of a location, whereas Γ contains the type of the location.

Definition 2.6.3 (Well-formed configuration). *Given a configuration $\langle c, pc, m, h, t \rangle$, a typing environment Γ and a heap typing environment Σ , say that the configuration is well-formed w.r.t. Γ, Σ , if*

1. $c \neq \text{stop} \implies \Gamma, pc \vdash c$
2. (m, h) is free of dangling pointers.
3. (m, h) is well-formed wrt. (Γ, Σ) .
4. If $\Gamma(x) = \mathbf{array}_{\ell_p}[\tau](\ell_{ref}, \circ)$ and $m(x) = l$ then $\Sigma(l) = \tau$.

By a standard proof of preservation [135] the semantics can be shown to preserve the well-formedness of configurations.

2.7 Security Guarantees

This section presents the security guarantees obtained by combining the properties of partitioned allocation semantics (Section 2.4), constrained garbage collection (Section 2.5), and security types (Section 2.6).

For garbage collection in isolation we obtain a property of timing-sensitive noninterference.

For programs in general, the semantic security property obtained in this section is *termination-insensitive timing-sensitive noninterference*. While this may appear unorthodox, given the usual expectation that timing-sensitivity implies termination-sensitivity, we believe it makes sense in our setting, where the attacker has access to the internal clock of the computation, yet there

$$\begin{array}{c}
\Gamma(x) = \mathbf{array}_{\ell_p}[\tau](\ell_{ref}, \circ) \\
\hline
\ell_{ref} \sqsubseteq \ell \quad m(x) = l \\
\hline
\text{reach}_{\ell}(l, \Gamma, \Sigma, m, h)
\end{array}
\qquad
\begin{array}{c}
\text{reach}_{\ell}(l, \Gamma, \Sigma, m, h) \\
\Sigma(l) = \mathbf{array}_{\ell_p}[\tau](\ell_{ref}, \circ) \\
\hline
\ell_{ref} \sqsubseteq \ell \quad h(l) = (\ell', \mu) \quad \mu(n) = l' \\
\hline
\text{reach}_{\ell}(l', \Gamma, \Sigma, m, h)
\end{array}$$

Figure 2.15: Low-reachability

are many ways via which the program may diverge. The sources of divergence may be infinite loops – that we allow, heap exhaustion – also possible in our semantics, or other fatal errors that we do not currently model. In that light, even though we allow termination channels in this work, it remains a channel that cannot be efficiently magnified [13]. Note however that termination-insensitivity is not a fundamental restriction in our work. Because our type system is relatively standard, it should be possible to apply orthogonal techniques [76] to obtain termination or progress-sensitive security.

Our notion of noninterference is parametrized over the heap size, because of the parametrization of the semantics.

2.7.1 Low-Equivalence

In order to formalize our security conditions, we introduce ℓ -equivalence [143] for memories and heaps.

We define the set of low-reachable locations, written $\text{reach}_{\ell}(\Gamma, \Sigma, m, h)$, as the set of locations l satisfying the predicate $\text{reach}_{\ell}(l, \Gamma, \Sigma, m, h)$, which is specified in Figure 2.15. Intuitively, this is the set of locations reachable by following only pointers with a low confidentiality level according to the memory and heap typing environments. An important property of low reachability is that, when a location maps to a low heap level, then the low reachability of that location coincides with its reachability as per Definition 2.5.2.

Lemma 2.7.1 (Adequacy of low reachability). *Let (m, h) be well-formed wrt. (Γ, Σ) and let ℓ be a security level. If $h(l) = (\ell', \mu)$ and $\ell' \sqsubseteq \ell$ then*

$$\text{reach}_{\ell}(\Gamma, \Sigma, m, h) = \text{reach}(m, h)$$

Lemma 2.7.1 will be a crucial part of the proof of garbage collection noninterference in Section 2.7.2.

Memory low-equivalence We first consider a definition of low-equivalence for memories. This relation is induced by Γ .

$$m \sim_{\ell}^{\Gamma, \phi} s \triangleq \forall x \in \text{dom}(\Gamma) . m(x) \stackrel{\Gamma(x)}{=}_{\phi, \ell} s(x)$$

Whenever Γ is clear from the context, we omit it for clarity.

Memory and heap low-equivalence As we wish to reason about garbage collection, which identifies and removes unreachable locations, reasoning about locations that are low reachable is not sufficient to prove the desired non-interference results. Thus, we define the set of *low locations* as not only the low reachable ones, but also the locations which has a low heap level. This is captured in the following definition.

Definition 2.7.1 (Low locations). *Given security typing Γ , heap typing environment Σ , memory m , and heap h , the set $\text{Low}_{\ell}(\Gamma, \Sigma, m, h)$ is the smallest set such that*

1. *Low reachable locations are contained in the set:*

$$\text{reach}_{\ell}(\Gamma, \Sigma, m, h) \subseteq \text{Low}_{\ell}(\Gamma, \Sigma, m, h)$$

2. *Locations with a low heap level are contained in the set:*

$$\forall l . h(l) = (\ell', \mu) \wedge \ell' \sqsubseteq \ell \implies l \in \text{Low}_{\ell}(\Gamma, \Sigma, m, h).$$

We can now define heap low equivalence similarly to memory low-equivalence. Crucial for the non-interference is the idea that if two locations are related by a bijection ϕ , then one location is reachable if and only if the other location is reachable. This is captured in the following definition.

$$\begin{aligned} \text{reach-iff}_{\phi, \ell}^{\tau}(\Sigma_1, \Sigma_2, m, h, s, w) &\triangleq \\ \forall l_1 l_2 . \phi(l_1) = l_2 &\implies \\ l_1 \in \text{reach}_{\ell}(\Gamma, \Sigma_1, m, h) &\iff l_2 \in \text{reach}_{\ell}(\Gamma, \Sigma_2, s, w) \end{aligned}$$

The final relation specified by STATE-LOW-EQ, written $(m, h) \stackrel{\Gamma, \Sigma_1, \Sigma_2}{\approx}_{\phi, \ell} (s, w)$, is parametrized by the level ℓ of the low-equivalence (typically low), the typing environment Γ , the two heap typing environments Σ_1 and Σ_2 , and the substitution ϕ that witnesses the isomorphism between the heaps h and w . This is the main relation of interest during execution, as this what relates the parts of the environments that the attacker can observe. It relates the low reachability of the two environments, and specifies that the memories

$$\begin{array}{c}
\llbracket \text{int} \rrbracket = \mathbb{Z} \qquad \text{VAL-LOW-EQ} \\
\llbracket \text{array}_\ell[\tau] \rrbracket = \text{Loc} \qquad \frac{\tau = \sigma(\ell', \iota) \quad v_i \in \llbracket \sigma \rrbracket \quad i = 1, 2 \quad \ell' \sqsubseteq \ell \implies \phi(v_1) = v_2}{v_1 \stackrel{\tau}{\approx}_{\phi, \ell} v_2} \\
\\
\text{HEAP-LOC-EQ-NON-REACH} \\
\frac{l_i \notin \text{reach}(m_i, h_i) \text{ for } i = 1, 2 \quad \text{dom}(\mu_1) = \text{dom}(\mu_2) \quad h_i(l_i) = (\ell', \mu_i) \text{ for } i = 1, 2}{\text{heap-loc-eq}_{\phi, \ell}^\tau(m_1, h_1, m_2, h_2, l_1, l_2)} \\
\\
\text{HEAP-LOC-EQ-REACH} \\
\frac{l_i \in \text{reach}(m_i, h_i) \text{ for } i = 1, 2 \quad \text{dom}(\mu_1) = \text{dom}(\mu_2) \quad h_i(l_i) = (\ell', \mu_i) \text{ for } i = 1, 2 \quad \forall n. \mu_1(n) \stackrel{\tau}{\approx}_{\phi, \ell} \mu_2(n)}{\text{heap-loc-eq}_{\phi, \ell}^\tau(m_1, h_1, m_2, h_2, l_1, l_2)} \\
\\
\text{STATE-LOW-EQ} \\
\frac{m_1 \stackrel{\Gamma, \phi}{\sim}_\ell m_2 \quad \text{reach-iff}_{\phi, \ell}^\tau(\Sigma_1, \Sigma_2, m_1, h_1, m_2, h_2) \quad (\forall l_1, l_2. \phi(l_1) = l_2 \wedge \Sigma_i(l_i) = \tau \wedge l_i \in \text{Low}_\ell(\Gamma, \Sigma_i, m_i, h_i) \ i = 1, 2) \implies \text{heap-loc-eq}_{\phi, \ell}^\tau(m_1, h_1, m_2, h_2, l_1, l_2)}{(m_1, h_1) \stackrel{\Gamma, \Sigma_1, \Sigma_2}{\approx}_{\phi, \ell} (m_2, h_2)}
\end{array}$$

Figure 2.16: Low-equivalence on memories and heaps

are related, and finally the last relation expresses that if two low locations are related by ϕ and have data labeled at a common type τ then they must have the same heap level, and if the locations are reachable then the values associated with this location (i.e. the codomain of μ_1 and μ_2 in **HEAP-LOC-EQ-REACH**) have to be equivalent at type τ . When Γ and Σ_1, Σ_2 are irrelevant, we write $(m, h) \approx_{\phi, \ell} (s, w)$ for $(m, h) \stackrel{\Gamma, \Sigma_1, \Sigma_2}{\approx}_{\phi, \ell} (s, w)$.

2.7.2 Noninterference for Garbage Collection

Using the definition of memory and heap low-equivalence we can formulate the noninterference result for the garbage collector. Our definition of noninterference for garbage collection is possibilistic [73] in its nature. Intuitively, it states that for a GC-transition that takes some time there is a GC-transition that takes as much time and yields a low-equivalent resulting heap and memory.

Theorem 2.7.2 (Garbage collection noninterference). *Assume typing environment Γ , level ℓ and heap typing environments Σ_1 and Σ_2 . Consider two well-formed*

configurations $\langle c_1, pc, m, h, t \rangle$ and $\langle c_2, pc, s, w, g \rangle$ wrt. Γ, Σ_1 , and Γ, Σ_2 , a substitution ϕ such that $(m, h) \stackrel{\Gamma, \Sigma_1, \Sigma_2}{\approx}_{\phi, \ell} (s, w)$. Assume $pc \sqsubseteq \ell$. If $\langle c_1, pc, m, h, t \rangle \dashrightarrow \langle c_1, pc, m, h', t + \delta \rangle$ then there is w' and ψ such that

$$\langle c_2, pc, s, w, g \rangle \dashrightarrow \langle c_2, pc, s, w', g + \delta \rangle$$

and $(m, h') \stackrel{\Gamma, \Sigma_1, \Sigma_2}{\approx}_{\psi, \ell} (s, w')$.

Proof sketch. Unfolding we have $h = h^{=pc} \uplus h^{\neq pc} \uplus h^{gc}$, where h^{gc} is the sub-heap being collected. Pick

- $w^{=pc} = \phi(h^{=pc})$,
- $w^{gc} = \phi(h^{gc})$.

By definition no location in h^{gc} is reachable, and so they are not low-reachable either. Then (since $\text{reach-iff}_{\phi, \ell}^{\tau}(\Sigma_{1,2}, m, h, s, w)$) no location in $w^{gc} = \phi(h^{gc})$ must be low-reachable and since all locations in w^{gc} have heap level $pc \sqsubseteq \ell_{\text{adv}}$ it follows that no location in w^{gc} is reachable, meaning that it is safe to GC this part of the heap.

By definition of GC we have $h^{=pc} \uplus h^{gc} \rightsquigarrow_{\delta}^m h^{=pc}$ and so $w^{=pc} \uplus w^{gc} \rightsquigarrow_{\delta}^s w^{=pc}$ by the GC assumption from Section 2.5. \square

A property of the state low equivalence relation is that garbage collection when $pc \not\sqsubseteq \ell_{\text{adv}}$ results in a state which is low equivalent to the state before garbage collection.

Lemma 2.7.3 (High garbage collection). *Assume typing environment Γ and level ℓ . Consider configuration $\langle c, pc, m, h, t \rangle$ and assume $pc \not\sqsubseteq \ell$. If*

$$\langle c, pc, m, h, t \rangle \dashrightarrow \langle c, pc, m, h', t' \rangle$$

then $(m, h) \approx_{\text{id}, \ell} (s, h')$.

2.7.3 Noninterference for Programs

Formal attacker observations To simplify the technical presentation we assume that the secrets in the computation are all stored in the initial memory.

We present our noninterference condition using the notion of attacker knowledge [12, 36]. The attacker knowledge is the set of possible memories that are consistent with the memory after a sequence of program transitions. We assume that programs start with empty heap \emptyset and an initially low pc level \perp .

Definition 2.7.2 (Attacker knowledge at level ℓ). *Given program c , initial and final memories m and m' , final heap h' , security level ℓ and maximum heap size function mx , define attacker knowledge as*

$$k_\ell^{mx}(c, m, m', h') \triangleq \left\{ s \mid \begin{array}{l} m \stackrel{id}{\sim}_\ell s \wedge \langle c, \perp, s, \emptyset, 0 \rangle \xrightarrow{mx}^* \langle stop, pc', s', w', t' \rangle \\ \wedge \exists \phi. (m', h') \approx_{\phi, \ell} (s', w') \end{array} \right\}$$

Note that the larger attacker knowledge set corresponds to attacker obtaining less information. Smaller knowledge sets correspond to more precise information. Singleton knowledge set means the attacker knows the exact initial memory with which the execution started.

Definition 2.7.3 (Set of terminating memories). *Given a program c we define $M_\ell^{mx}(c, m)$ as the set of initial ℓ -equivalent memories that lead to a terminating configuration when the heap is bounded by mx .*

$$M_\ell^{mx}(c, m) \triangleq \left\{ s \mid m \stackrel{id}{\sim}_\ell s \wedge \langle c, \perp, s, \emptyset, 0 \rangle \xrightarrow{mx}^* \langle stop, pc', m', h', t' \rangle \right\}$$

Using attacker knowledge and the set of initial memories we can define the noninterference policy [46]. Intuitively, a program satisfies noninterference if any memory and heap produced by a terminating sequence of program steps does not exclude any possible initial memory.

Definition 2.7.4 (Termination-Insensitive Noninterference at ℓ for heap size mx). *Given a heap bounding function mx , a program c satisfies mx -noninterference up to level ℓ if for all initial memories m such that*

$$\langle c, \perp, m, \emptyset, 0 \rangle \xrightarrow{mx}^* \langle stop, pc', m', h', t' \rangle$$

implies $k_\ell^{mx}(c, m, m', h') \supseteq M_\ell^{mx}(c, m)$.

Intuitively, this definition says that a program c satisfies termination-insensitive noninterference when, given two terminating executions of the same program with low equivalent initial memories m and s and final memories m' and s' , it is possible to construct a terminating execution of c starting at initial memory s which results in a final memory s'' such that s'' is low equivalent to s' .

Theorem 2.7.4 (Soundness of the enforcement). *Given a program c , if $\Gamma, pc \vdash c$ then c satisfies noninterference for unbounded semantics for all levels ℓ .*

Proof sketch. Given two terminating executions of program c :

$$\begin{aligned} \langle c, \perp, m, \emptyset, 0 \rangle &\xrightarrow{\text{mx}}^* \langle \text{stop}, pc_1, m', h', t' \rangle \\ \langle c, \perp, s, \emptyset, 0 \rangle &\xrightarrow{\text{mx}}^* \langle \text{stop}, pc_2, s^*, w^*, g^* \rangle \end{aligned}$$

(call these executions A and B resp.) our goal is to construct an alternative run (call this execution C)

$$\langle c, \perp, s, \emptyset, 0 \rangle \xrightarrow{\text{mx}}^* \langle \text{stop}, pc_1, s', w', t' \rangle$$

where (m', h') is \approx -equivalent to (s', w') .

To show this, we start by defining an auxiliary “bridge” relation on pairs of configurations. The relation records that starting from some configuration C_1 , the execution “steps over” a number of intermediate steps, that do not modify the low parts of the memory or the heap, or terminates, resulting in configuration C_2 . Each of the A and B executions can be broken down into a sequence of consecutive “bridging” steps. We construct the execution C one bridge-step at a time, starting from the initial configurations. The key invariant used in the proof is that bridging configurations in C execution are low-equivalent with the respective configurations in A and “taint”-equivalent with the respective configurations in B. Our workhorse bridge noninterference shows that whenever a pair of related configurations in A and B can take a bridge step, it is possible to construct a matching bridge step in C that “mimics the timing behavior” of A. i.e., it generates the same events and takes the same execution time as A. This is proved by induction on the number of the intermediate steps, followed by an induction over the structure of command c . In order to show the termination of the high commands constructed in C run, we observe that the taint equivalence of B and C configurations implies that they agree on the control flow, and one can further construct high GC steps in C to match the high GC steps in B.

□

2.8 Connections to Real Time Garbage Collection

Wadler [132] defined the term *real time garbage collection* as any garbage collection system guaranteeing that the execution is not suspended for long periods of time. Many such collectors have been presented previously [44, 58, 102, 121], and they are a crucial part of building real time systems in managed languages [48]. As the main goal of real time garbage collectors is to reduce the amount of time the garbage collector suspends the program

execution, such collectors could be seen as an effective mitigation against the attacks presented in Section 2.3.

While we have not performed experiments against existing real time garbage collectors at this point, we expect that real time garbage collectors are not sufficient for mitigating these attacks. To see this, consider the second program from Section 2.5.1 that illustrates the danger of collecting **H** in **L**. An “eager” real-time garbage collector may manifest the behavior that this example warned against. Additionally, if a practical implementation of real-time garbage collection algorithm occasionally stops the world for collection, this allows attacks similar to the first example in Section 2.5.1.

2.9 Related Work

Conceptually, this work fits into the framework introduced by Zhang et al. [143], where the interaction between the language semantics and the underlying abstract runtime happens via security labels. The work by Zhang et al. does not however consider automatic memory management.

Many modern programming language features can be used to create timing channels, and programming language designers must secure the entire execution stack. Buiras and Russo [25] show that a programming language with lazy evaluation leak information because of sharing. Buiras and Russo [25] break the information flow control of the Haskell library LIO by leveraging the way thunks are shared between threads. They present a method for leaking one bit of information, along with a technique to amplify the attack. As a solution, they propose a restriction on sharing of thunks between threads, but do not prove noninterference.

Secure-multi-execution [35] guarantees timing-sensitive noninterference by running multiple copies of the program, at the cost of changing the semantics of insecure programs. In order for the secure-multi-execution to provide guarantees against leakage via memory management it must be necessary to run each copy with a separate collector, effectively enforcing the constraints of Section 2.5. Without such isolation, the shared GC is likely to represent a source of timing channels.

The idea of partitioning heaps based on security levels appears in the work on Relational Hoare Type Theory (RHTT) by Nanevski et al. [88] that provides two different allocation primitives: `lalloc` for public allocations and `alloc` for secret ones. Locations obtained by different primitives return disjoint values, which is necessary for defining heap low-equivalence, in many ways similar to our work. However, Nanevski et al. [88] do not consider neither automatic collection nor timing-sensitivity.

The idea of the runtime pc-level is related to the floating label concept in Haskell LIO library [115] that relies on lightweight threads to eliminate internal timing leaks in applications. It is not clear how to combine secure scheduling necessary for concurrent information flow with restrictions on garbage collection that we present here.

Recent years have seen a surge of efforts on verification of garbage collector algorithms and implementations [45, 72]. While these project focus on basic safety properties, they may provide a foundation for design of implementations that would satisfy our security requirements.

2.10 Conclusion

This paper presents a series of examples that demonstrate feasibility of information leaks via garbage collection. To effectively control such leaks, a tight integration between runtime and the source-level language is needed. We observe that even despite drastic simplifications in the design of the language to simplify aspects such as direct timing attacks, closing leaks via garbage collector requires strong assumptions from the language implementors.

Chapter 3

Static Enforcement of Security in Runtime Systems

MATHIAS V. PEDERSEN, AARHUS UNIVERSITY, DENMARK
ASLAN ASKAROV, AARHUS UNIVERSITY, DENMARK

In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, 2019.

Abstract

Underneath every modern programming language is a runtime environment (RTE) that handles features such as automatic memory management and thread scheduling. In the information-flow control (IFC) literature, the RTE is often part of the trusted computing base (TCB), and there has been little focus on applying IFC to the implementation of the RTE itself. In this paper we address this problem by designing an IFC language, Zee, for implementing secure RTEs, thereby removing the RTE from the TCB. We implement Zee and design and implement secure versions of garbage collectors and thread schedulers using Zee. We also prove that a faithful calculus of Zee satisfies a strong variant of timing-sensitive noninterference.

3.1 Introduction

Modern programming languages offer many abstraction mechanisms to simplify the development of programs, increase their throughput, or reduce their resource consumption at runtime. Such abstractions are often implemented by compiler writers in a specialized program called the runtime

environment (RTE). A runtime environment is a program (possibly written in a different language) that is running alongside programs written by the user, and may use knowledge about the implementation details of the language to perform its tasks. Examples of features commonly associated with the RTE include thread scheduling and automatic memory management. For a user of the language, implementing such functionality is difficult as it often requires breaking abstractions enforced by the language designer, and careful reasoning must be done by the developers of the runtime environment to ensure that the guarantees offered by the language are not violated by the runtime environment. For instance, a garbage collector must determine which heap allocations are reachable by following pointers through the heap, starting from a set of *root* pointers. Root pointers typically include the local variables stored on the call stack, and thus the RTE must traverse every stack frame currently stored on the call stack, potentially breaking local state encapsulation [113]. Even worse is the situation from a security perspective, where operations carried out by an RTE might reveal confidential information about the data handled by the user-written programs through storage- or timing channels [92, 109, 116, 127].

Language-based information-flow control (IFC) is a popular approach to solve the problem of ensuring integrity and confidentiality of data [107]. This approach uses programming language techniques to analyze information flows in potentially untrusted programs before and/or during their execution, and verifies that the execution will not leak sensitive information.

In this paper we design and implement Zee: an IFC programming language for implementing secure runtime environments. Zee supports secure and type-safe programming on heterogeneous data (e.g., data at multiple security levels). We also define a faithful calculus of Zee and prove that well-typed programs satisfy timing-sensitive noninterference.

The calculus, and its implementation, is defined over an abstract *instantiation language*, which allows for Zee to be extended with additional features without redoing many of the formal proofs. Instead, each instantiation must only prove that a certain semantic interface is satisfied.

In summary, this paper makes the following contributions:

- It identifies access to the call stack and runtime type analysis as core features necessary for a practical language for programming runtime environments.
- It presents the design and implementation of Zee: an extensible language for implementing RTEs, and proves that the combination of

Zee’s type system and runtime semantics enforces timing-sensitive noninterference.

- It uses Zee’s extensible semantics to design and implement a secure garbage collector, and a secure thread scheduler.

The rest of the paper is structured as follows: Section 3.2 gives an introduction to Zee through examples, and Section 3.3 formally defines the syntax and semantics of Zee. Section 3.4 formalizes the attacker model and the security guarantees enforced by Zee’s type system. Section 3.5 presents two case studies on how Zee can be used to implement well-typed (and hence secure) garbage collectors and thread schedulers. Section 3.6 describes the implementation of Zee, and Section 3.7 discusses related work. The Appendix contains full definitions and the accompanying technical report contains full proofs.

3.2 Programming in Zee

This section introduces Zee using three example programs. The first example demonstrates how Zee uses existentially quantified type variables to compute securely on values with different security levels located in the same data structure.¹ The second example demonstrates how a similar technique can be used to securely inspect the call-stack at runtime. Finally, the third example demonstrates how Zee uses implicit revocation of expired pointers to prevent reuse of invalidated memory.

3.2.1 Computing on Heterogeneous Values

The first example uses a simple two-point lattice consisting of two elements L and H with the partial order $L \sqsubseteq H$, and $L \sqcup L = L$ and $\ell_1 \sqcup \ell_2 = H$ otherwise. We call L the “public” level and H the “secret” level. We use a level-partitioned heap [92] and so, in the two-point lattice, the heap consists of two partitions which we denote as the L partition and the H partition. As is standard for type systems for security, types are augmented with security levels, i.e., the type $\text{int } H$ is the type of integers at security level H .

The type system also tracks which partitions pointers point into. The type $(\ell_1 \mapsto s) \ell_2$ describes pointers that point into the ℓ_1 section, containing data of type s , and where the size of the allocation depends on information up to level ℓ_2 .

¹This example uses a specific instantiation language defined in Section 3.5.1.

Zee uses existential types [124] and runtime type analysis [53] for secure handling of heterogeneous data. Traditional existential types are written as $\exists \alpha : \mathbf{type}. s$, and a value of type $\exists \alpha : \mathbf{type}. s$ is a pair (τ, v) where τ is a runtime representation of a type,² and v is a value of type $s[\tau/\alpha]$. That is, v is of type s , but where the free type variable α has been replaced by τ . A value of an existential type can be introduced using the expression $\text{pack } (s, e)$ as $\exists \alpha : \mathbf{type}. s$, which evaluates s to τ and e to v before returning a pair (τ, v) . Dually, given an expression e of type $\exists \alpha : \mathbf{type}. s$, it can be eliminated using the command $\text{let } (\alpha : \mathbf{type}, x : s) := e \text{ in } c$, that brings the type variable α , and the variable x into scope for the evaluation of command c . For information-flow control, the existential type is augmented with security levels, and a value (τ, v) is of type $(\exists \alpha : \mathbf{type} \ell_1. s) \ell_2$ if τ depends only on information up to level ℓ_1 . Introduction and elimination rules are augmented accordingly as $\text{pack } (s, e)$ as $\exists \alpha : \mathbf{type} \ell_1. s$, and $\text{let } (\alpha : \mathbf{type} \ell_1, x : s) := e \text{ in } c$ respectively. We call the latter command an *unpacking* command.

Figure 3.1 shows a Zee function `compute` that, given an array `xs` with elements of different security types, computes the sum of all the public integers in the array, revealing no information about the secret values in `xs`. The array `xs` is annotated with the type $(\mathbf{L} \mapsto (\exists \alpha : \mathbf{type} \mathbf{L}. \alpha) \mathbf{L}) \mathbf{L}$, representing an array of heterogeneous data in the \mathbf{L} partition of public length.

Function `compute` also specifies two additional labels (both of which are \mathbf{L} in this example): the bottom label is a lower bound on the program counter label, which is classic in IFC literature [83]. The top label is novel: it represents an upper bound on the sensitivity of the information that can be learned by knowing the type of a local variable in the current activation frame. We defer the discussion of this label until Section 3.3.3.

On lines 5 to 9, function `compute` loops over the elements of `xs`, and on lines 6 and 7 it extracts the witness α and the value of type α . On line 8 the code performs runtime type analysis on the value of α , and it matches the pattern `int L` (i.e., the type of public integers), the value `y` is known to be of type `int L` on line 8, and can be added to the public variable `sum`, without leaking sensitive information. Finally, if α is of any other type, this value is omitted from the final sum.

This example demonstrates how Zee securely computes on heterogeneous values using existential types and runtime type analysis. In the next example, we extend such use of existential types to access the call stack while guaranteeing type-safety and security. We do this by treating the frame pointer as a pointer to an array of existentially quantified types, similar to

²The value τ is often called the *witness* of the type s .

```

1 compute(xs : (L → (∃α: type L . α) L) L) =L
2   let n : int L := length xs in
3   let i : int L := 0 in
4   let sum : int L := 0 in
5   while i < n do
6     let x : (∃α: type L . α) L := *(xs + i) in
7     let (α : type L, y : α) := unpack x in
8     match α with int L ⇒ sum := sum + y
9       | _           ⇒ skip;
10    i := i + 1

```

Figure 3.1: Zee program demonstrating computations on heterogeneous values.

the type of `xs` in Figure 3.1. For the remaining examples in the paper we elide some type annotations, but all of the missing type annotations are inferred by our prototype implementation of Zee.

3.2.2 Computing with the Call-Stack

Zee allows fine-grained reasoning about the call stack, which is important for operations such as stack traversal for many garbage collection algorithms, or unwinding the stack to handle exceptions. Figure 3.2 shows the structure of the call stack during an execution with two activation frames belonging to functions `g` and `f` respectively. Function `g` pushes two arguments on the stack: a value of type $(H \mapsto \text{int } H) H$, and a value of type $\text{int } H$, and then invokes `f`. Function `f` then establishes a new frame by pushing the value of the old frame pointer onto the stack, so that the previous frame can be restored upon returning from `f`. Furthermore, `f` modifies the frame pointer to point to its local variables, and modifies the stack pointer to point to the next free address on the stack. Finally, `f` allocates its local variables and proceeds with computation in the newly established activation frame.

As Zee features runtime type analysis, which is crucial for the implementation of many useful programming language features, the types themselves must be protected using security labels. To accomplish this, we introduce the notion of a frame label fr , which represents an upper bound on the sensitivity of the information that can be learned by knowing the type of a local variable in the current frame.

In Zee the expression `FP` returns a pointer to the beginning of the current activation frame. The type assigned to `FP` is a recursive type,³ that reflects

³We will explain the precise typing of the frame pointer in detail in Section 3.3.3.

```

1  f(p : (H ↦ int H)H, h : int H) =HL
2    let a : int L := 0 in
3    let b : (L ↦ int H) L := null in
4    let c : int H := h in skip
5  g() =HL ...; f(null, 42); ...
6

```

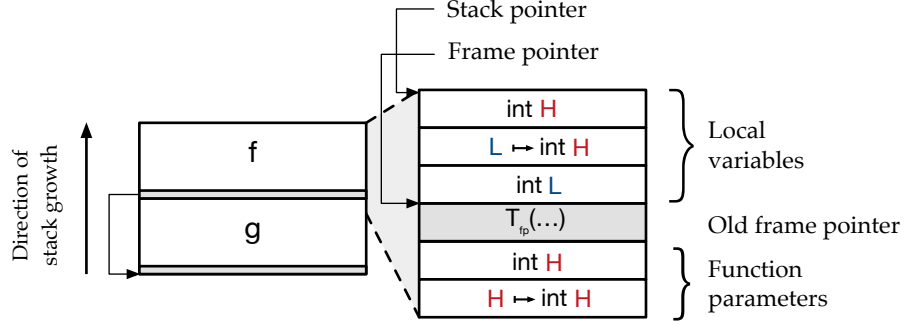


Figure 3.2: Top: two functions g and f . Bottom: the structure of the call stack just before executing `skip` in f .

the layout of the stack (cf. Figure 3.2) and contains the types of the function parameters and local variables.

To understand the need for the frame label, consider the program in Figure 3.3, showing a function `inspect` that inspects its own frame. Unlike the `compute` function, the `inspect` function is parameterized by a secret label κ , which the caller provides when invoking `inspect`. In this example, the first element of the frame layout consists of an integer x with the security label κ , as declared on line 2. Using the `FP` construct on line 3, the type variable α_{args} is assigned a tuple type representing the types of the arguments passed to `inspect`,⁴ and on line 4, α_{locals} is assigned the types of the local variables of `inspect`.

As no arguments has been passed to `inspect`, the value of α_{args} during execution is the unit type ε . The value of α_{locals} is $\text{int } \ell' \cdot \tau_{\text{tail}}$, where ℓ' is the value of κ that has been provided when `inspect` was called, and τ_{tail} is the rest of the frame layout. After obtaining the type of the local variables, line 5 performs a pattern match on α_{locals} , and if the pattern $\text{int } \ell \cdot _$ matches the value of α_{locals} , it follows that ℓ' is equal to ℓ , which is information classified at H . So to track the indirect flow from the label κ to the match command,

⁴The unroll expression is needed as Zee uses isorecursive types to represent the type isomorphism between $\mu \alpha : \text{type}. s$ and $s[\mu \alpha : \text{type}. s/\alpha]$ (i.e., between a recursive type and its unrolling). We will omit unroll expressions throughout the paper.

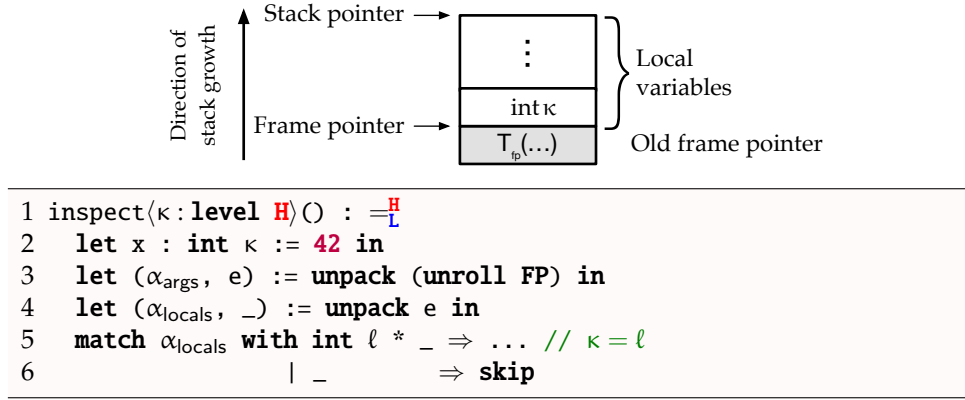


Figure 3.3: Bottom: Indirect information flow from κ to α_{locals} when `inspect` inspects its own stack frame. Top: The activation frame for function `inspect`.

we introduce the frame label fr , which assigns α_{locals} the security level fr (which is equal to **H** as declared by the `inspect` function), properly tracking the dependency of κ in α_{locals} .

3.2.3 Fail-Stop Revocation of Expired Pointers

This section demonstrates how Zee uses a dynamic enforcement technique to prevent the reuse of pointers that point to “expired” data. Expired data include local variables of functions that has returned control to its caller, or heap data that has been reclaimed by a garbage collector. To do this we introduce a technique similar to identifier-based temporal checking [87] (and the mechanism proposed by Tsampas et al. [123] for the CHERI architecture [134]) that we simply refer to as *versioning*. At runtime, every pointer is assigned natural number v called the version number. Similarly, every stack frame is assigned a version number. When a pointer to a local variable on a stack frame is created, the pointer is assigned the version number of that stack frame. When reading data pointed to by a pointer with version v , the system checks that the stack frame, which is read from, has a version number γ such that $\gamma \leq v$, ensuring that this stack frame was “live”⁵ when the pointer was created, and a similar check is done for writes through pointers.

Figure 3.4 demonstrates how an illegal flow could be constructed without versioning: on line 6, x is allocated on the stack and initialized to null.⁶

⁵A live stack frame refers to a frame of a computation that is still ongoing.

⁶For now the type $@s$ can be read as “pointer to a value of type s ”, but we will define a more general version of this type in Section 3.3.

```

1 f(px : (@(@ int L) L) L) =L
2   let y : int L := 0 in *px := &y
3 g(z : (@ int L) L) =L
4   let h : int H := secret in
5   let low : int L := *z in skip
6 let x : (@ int L) L := null in f(&x); g(x)

```

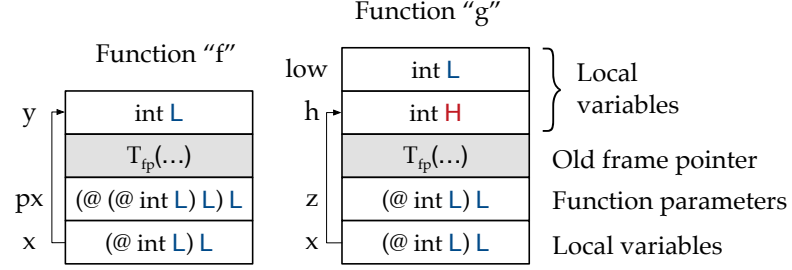


Figure 3.4: Attempting to read x in function g will fail: the pointer has been implicitly revoked when f no longer was live.

Then, function f is called and writes the address of its local variable y into x through the passed pointer px on line 2. When f returns, the value of x is a pointer, pointing to the local variable y from the "dead" stack frame of f . Then, when g is called, the value of `secret` is stored in h , which might be located in the same offset from the base pointer as y was in function f (cf. bottom of Figure 3.4). So when g reads x , the value of `secret` is stored in low , violating the type ascribed to the variable.

Versioning prevents this leak by assigning a version number $v \in \mathbb{N}$ to x and the stack frame allocated on line 6. The stack frames of f and g are assigned versions $v + 1$ and $v + 2$ respectively. When g reads from x , this variable has version $v + 1$, while the data it points to is located on g 's stack frame, which has version $v + 2$. This violates the version check and execution stops, successfully preventing the leak.

3.3 Language

This section presents a formalization of Zee. We assume a lattice \mathcal{L} of security levels with a least element \perp and let ℓ range over the elements of \mathcal{L} .

3.3.1 Syntax

Figure 3.5 defines the grammar for Zee. We give an informal description of each syntactic category, and delay the formal semantics until Section 3.3.2.

Commands

Commands c include standard constructs for imperative languages. Non-standard commands include commands $*e := e$ and $x := *e$ that respectively write to, and read from, a location on the stack. Command $\text{at } k$ with bound e do c raises the program counter label for command c during type checking, and also provides lightweight⁷ predictive mitigation [14, 142, 143]. The `match` command allows for runtime type analysis [1, 53], which is crucial for implementing many useful tasks associated with the RTE. We call the α being matched on the *scrutinee*, and the patterns are defined in the syntactic category *pat*. These include integer type patterns (`int` κ), stack pointer type patterns ($(\text{pat} @ \text{pat}) \kappa$), heap pointer type patterns ($(\kappa \mapsto \text{pat}) \kappa$), product type patterns ($\overline{\text{pat}}$) and a “catch all” pattern that names the type value (α). To facilitate traversing the stack at runtime, the language exposes the frame pointer, which can be obtained using the $x := \text{fp}$ command. A function can be invoked by supplying the function name with label⁸ parameters \bar{k} , type parameters \bar{s} , and expression parameters \bar{e} . Finally, both existentially quantified labels and types can be eliminated using the unpacking commands $\text{let } (\alpha : \mathcal{X} \kappa, x : s) := e \text{ in } c$ for $\mathcal{X} \in \{\text{type}, \text{level}\}$. While the language does not include an assignment command, this can be encoded using the “address-of” operator $\&$ and the command $*e := e$ for writing to the stack. For readability we assume the presence of an assignment operator in our example programs.

Expressions

The meta-variable e ranges over expressions which include numbers, variables, binary operations, a special pointer value `null`, and pack expressions for introducing existentially quantified labels and types. We include recursive types to give the language type-safe access to the call stack, and expressions `unroll e` and `roll e` allow for simple (i.e., isorecursive) typing rules for recursive types [101]. The size of a type can be calculated using the expression `sizeof s`, which is also used to facilitate stack traversal. Finally, given a variable x the address of x on the stack can be obtained as $\&x$.

Security Labels and Types

The meta-variable k ranges over security labels. As labels can be existentially and universally quantified, the category includes variables κ . Finally, one

⁷In particular, the predicted time for mitigation commands is given by the programmer, nor is it not automatically updated by the semantics.

⁸We distinguish between levels (i.e., elements of \mathcal{L}) and labels (i.e., expressions that evaluate to elements of \mathcal{L})

$$\begin{aligned}
 c &::= \text{skip} \mid \text{let } x : s := e \text{ in } c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \mid c; c \\
 &\mid *e := e \mid x := *e \mid \text{at } k \text{ with bound } e \text{ do } c \mid \text{if } (k \sqsubseteq k) \text{ then } c \text{ else } c \\
 &\mid \text{match } \alpha \text{ with } \overline{pat} \Rightarrow c \mid x := fp \mid f(\overline{k})\langle\overline{s}\rangle(\overline{e}) \\
 &\mid \text{let } (\alpha : \mathbf{type} \ k, x : s) := e \text{ in } c \mid \text{let } (\kappa : \mathbf{level} \ k, x : s) := e \text{ in } c \\
 e &::= n \mid x \mid e \oplus e \mid \text{null} \mid \text{unroll } e \mid \text{roll } e \mid \text{pack } (s, e) \text{ as } \exists \alpha : \mathbf{type} \ k. s \\
 &\mid \text{sizeof } s \mid \text{pack } (k, e) \text{ as } \exists \kappa : \mathbf{level} \ k. s \mid \&x \\
 k &::= \ell \mid \kappa \mid k \sqcup k \mid k \sqcap k \\
 s &::= tk \mid \alpha \mid \overline{s} \\
 t &::= \mathbf{int} \mid k \mapsto s \mid s @ s \mid \exists \alpha : \mathbf{type} \ k. s \mid \exists \kappa : \mathbf{level} \ k. s \mid \mu \alpha : \mathbf{type} \ k. s \\
 &\mid \mathbf{size}[s] \\
 pat &::= \mathbf{int} \ \kappa \mid (pat @ pat) \ \kappa \mid (\kappa \mapsto pat) \ \kappa \mid \overline{pat} \mid \alpha
 \end{aligned}$$

Figure 3.5: The syntax of Zee.

can form the join \sqcup or meet \sqcap of two security labels, representing the least upper bound, or greatest lower bound of two labels respectively.

The meta-variable s ranges over security types and include base types t with a security label k , written tk , type variables α , or product types \overline{s} . Base types are integers, heap pointers ($k \mapsto s$) representing a pointer into the heap partition associated with label k [92], stack pointers ($s_1 @ s_2$) representing pointers to a value of type s_2 that, on the stack, are located “above” a value of type s_1 [6, 99]. Base types also include the type of existentially quantified security types and labels, recursive types and singleton types [118] $\mathbf{size}[s]$ describing the size of the type s . A function definition is given as $f(\overline{\kappa} : \mathbf{level} \ k_1) \langle \overline{\alpha} : \mathbf{type} \ k_2 \rangle (\overline{x} : \overline{s}) =_{pc}^{fr} c$ that defines a function f with label parameters $\overline{\kappa}$, type parameters $\overline{\alpha}$, and value parameters \overline{x} . Label parameter κ_i can depend on the previous label parameters $\kappa_1, \dots, \kappa_{i-1}$, and type parameter α_i can depend on all label parameters and type parameters $\alpha_1, \dots, \alpha_{i-1}$. Finally, the types \overline{s} for the expression parameters can depend on all label and type parameters. Furthermore, pc is a lower bound on the side effects produced by f , and fr is an upper bound on the type of any local variable declaration. We revisit this label in Section 3.3.3.

3.3.2 Semantics

The semantics of Zee is given by a small-step relation \rightarrow on configurations C of the form $\langle c, M, P, q \rangle_\gamma$. We first define each component of the configuration before describing the small-step relation.

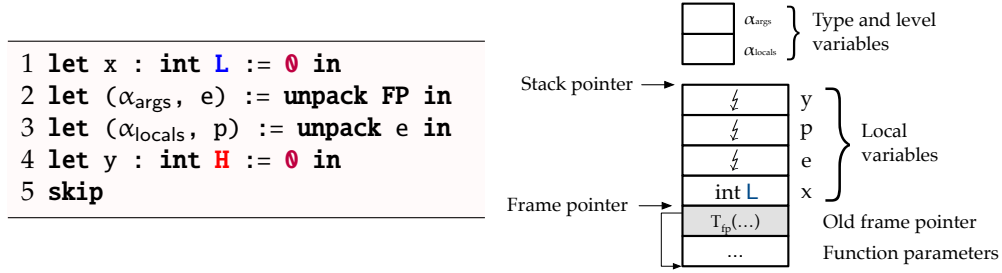


Figure 3.6: Left: the value of α_{locals} contains three nonsense type values, corresponding to the three variables e , p and y that has yet to enter the scope upon evaluating FP on line 2. Right: the stack layout when FP is evaluated.

Values

Figure 3.7 describes the syntax of values. A value v is either a number n , an address a with a *version number* $v \in \mathbb{N}$, a pair consisting of either a level and a value (ℓ, v) , or a security type value and a value (τ, v) . The meta-variable τ ranges over security type values and is either a base type value π with a security level ℓ , or a product of security type values.

Security type values also include a *nonsense* [79] type value ζ . To motivate the need for nonsense type values, consider the program in Figure 3.6. When evaluating FP on line 2, the variable x with value 0 is in scope, but none of the variables e , p or y have entered the scope, and their value on the stack are “garbage” values. So α_{locals} is a product type $\text{int } L \cdot \zeta \cdot \zeta \cdot \zeta$ containing three nonsense type values ζ as the variables e , p and y have yet to enter the scope and be assigned a value. The type variables α_{args} and α_{locals} live on a different stack (cf. top right of Figure 3.6) where extracting type-information is not possible, and therefore technically do not need a nonsense type value, as we do not track “the type of type variables” at runtime.

The meta variable π ranges over base type values, and contain the same constructs as the base types, but where security labels not under binders \exists and μ are fully evaluated, and labels on types and labels are erased (as they are only needed for type checking).

Exposed and Private Stack Frames

An exposed stack frame m is a triple $(\mathbb{I}, |m|, v)$ where $|m| : \mathbb{I} \rightarrow v$ is a partial function from a nonempty interval $\mathbb{I} \subset \mathbb{N}$ to values, and a frame version $v \in \mathbb{N}$. We call $\min(\mathbb{I})$ the frame pointer, written $\text{fp}(m)$, and $\max(\mathbb{I}) + 1$ the stack pointer, written $\text{sp}(m)$. Intuitively, $\text{fp}(m)$ is the minimum address in

$$\begin{aligned}
 v &::= n \mid a_v \mid (\ell, v) \mid (\tau, v) \\
 \tau &::= \pi \ell \mid \bar{\tau} \mid \not\downarrow \\
 \pi &::= \mathbf{int} \mid \ell \mapsto \tau \mid \tau @ \tau \mid \exists \alpha : \mathbf{type}. s \\
 &\quad \mid \exists \kappa : \mathbf{level}. s \mid \mu \alpha : \mathbf{type}. s \mid \mathbf{size}[\tau]
 \end{aligned}$$

Figure 3.7: Values in Zee.

the stack frame, corresponding to the usual notion of a frame pointer, and similarly $\text{sp}(m)$ is the address of the next available stack location. Given an exposed stack frame $m = (\mathbb{I}, |m|, v)$ we write $m[a \mapsto v]$ to mean $(\mathbb{I}, |m|[a \mapsto v], v)$, and $m(a)$ to mean $|m|(a)$. We call a list of exposed stack frames an exposed stack M .

A private stack frame p is a triple of partial functions $(p_{\text{var}}, p_{\text{arg}}, p_{\text{local}})$ where $p_{\text{var}} : \text{Var} \rightarrow \mathcal{L} \cup \tau$ and $p_{\text{arg}}, p_{\text{local}} : \text{Var} \rightarrow \tau$. Function p_{var} maps label- and type variables to levels and type values, p_{arg} maps function arguments to their security type, and p_{local} maps local variable names to their security type. The name “private” refers to the fact that this stack cannot be traversed and inspected at runtime (unlike the exposed stack). Finally, a private stack P is a list of private stack frames. We call a pair of a exposed and private stacks (M, P) a stack.

We distinguish between exposed and private stacks because Zee allows type-safe traversal of the exposed stack, but does not directly expose the private stack to programs.

As exposed stack frames include mappings from (subsets of) natural numbers, a translation from local variable names to addresses is needed. This translation is usually performed by a compiler, and many techniques exist for such translations [7]. We abstract away the specifics of such translations by assuming a global mapping $\delta : \text{Var} \rightarrow \mathbb{N}$ from variables names to offsets. Figure 3.8 demonstrates how a compiler might generate a mapping δ that maps variable names to activation frame offsets. At the moment we assume that the compiler does not coalesce stack locations when the lifetimes of two variables do not overlap. So given two local variables $x \neq y$ it holds that $\delta(x) \neq \delta(y)$. That is, different variables are stored at different locations on the stack. The expression $\delta(x) + \text{fp}(m)$ computes the address of the local variable x in the stack frame $\text{fp}(m)$.

```

1 fib(n : int L, r : (@ int L) L) =  $\perp$ 
2   if n <= 1 then *r := n
3   else let r1 : int L := 0 in
4         let r2 : int L := 0 in
5           fib(n-1, &r1); fib(n-2, &r2);
6         *r := r1 + r2

```

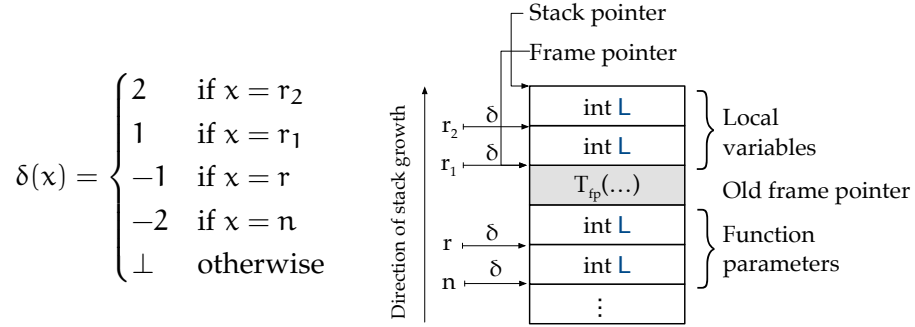


Figure 3.8: Top: an implementation of a function `fib` for computing a public Fibonacci number given a public input. Bottom right: the stack frame layout for `fib`. Bottom left: the mapping δ between local variable names and stack frame offsets.

Model of Time

We represent time as a number $q \in \mathbb{N}$ that counts the number of operational steps in the computation. This simple model is sufficient to demonstrate that runtime environment tasks can be computed in a timing-sensitive security setting. Naturally, a realistic implementation would need to soundly relate the operational steps with the wall-clock, but that is outside of the scope of the current work.

Version Counter

Finally, the configuration contains a version counter that keeps track of the next free version number. This is needed when constructing new stack frames, as each new frame is given a fresh version number.

Big-Step Evaluation for Expressions

The big-step evaluation for expressions is defined on configurations of the form $\langle e, m, p \rangle$, where m is an exposed stack frame and p is a private stack frame. The evaluation of expressions need both the exposed, and the private stack frame, as both expression variables and type- and level variables might appear in expressions. Figure 3.9 shows excerpts of the big-step eval-

$$\begin{array}{c}
 \text{E-Num} \\
 \langle n, m, p \rangle \Downarrow n \\
 \\
 \text{E-Pack-Ty} \\
 \frac{\langle s, p \rangle \Downarrow_{\text{type}} \tau \quad \langle e, m, p \rangle \Downarrow v}{\langle \text{pack } (s, e) \text{ as } \exists \alpha : \mathbf{type} \ k. s', m, p \rangle \Downarrow (\tau, v)} \\
 \\
 \text{E-Var} \\
 \frac{m(\delta(x) + \text{fp}(m)) = v}{\langle x, m, p \rangle \Downarrow v}
 \end{array}$$

Figure 3.9: Excerpts of the big-step evaluation of expressions.

uation semantics for expressions. Rule E-Num evaluates a literal, and E-Var evaluates a variable by looking up its value on the stack using the global mapping δ . Finally, rule E-Pack-Ty evaluates a pack expression containing a security type s and an expression e to a pair (τ, v) . The remaining rules are found in the technical appendix [97].

Small-Step Relation for Commands

Figure 3.10 shows an excerpt of the small-step relation, and the full semantics is in Figure A.4 in the Appendix. Rule S-AsGN evaluates an expression e and inserts the resulting value v in the memory at offset $\delta(x)$ of the current frame pointer. Rule S-FP stores the frame pointer in the variable x . In addition to the value of the frame pointer $\text{fp}(m)$, the value v contains the list of types $\text{cod}(p.\text{arg})$ of the arguments passed to the current function, and the list of types $\text{cod}(p.\text{local})$ of the local variables. Rule S-Let declares a new local variable x and (1) updates the stack location to contain the initial value of x , and (2) updates the private stack frame to contain type information about the type of x . This causes the type value of the variable x to be updated from a nonsense type value \perp to a meaningful type value τ , which is the result of evaluating the type s . After executing c , a command $\text{unscope}(x)$ removes the type information of x from the private stack p .⁹ Rule S-UNPACK-Ty unpacks an existential value containing a security type value and a regular value. Several maps are updated: the private stack frame is updated to contain the security type value and the type information about the newly allocated local variable. Finally, the exposed stack frame is updated to contain the regular value. Rule S-MATCH evaluates the scrutinee α to a security type value and performs type analysis according to a list of patterns $\overline{\text{pat}}$ using the relation $\tau \lesssim \text{pat}$ containing rules such as $\mathbf{int} \ell \lesssim \mathbf{int} \kappa$ and $\tau \lesssim \alpha$ ¹⁰ (i.e., integer patterns matches integer types, and name patterns matches any security type

⁹The semantics of unscope is defined in the Appendix.

¹⁰The complete definition of matching is found in the Appendix.

value). Upon finding the first (due to argmin) matching pattern the private stack frame is updated by binding relevant parts of the security type value to type variables, and execution proceeds with the command associated with the pattern.

Rule S-READ reads from a stack location a_γ . The version number γ is used to prevent attacks based on pointer reuse as was described in Section 3.2.3. Dually, S-WRITE writes a value v to an address a , requiring the same relation between the versions of the target stack frame and the address being read from.

Rules S-AT and S-DELAY implement simple predictive mitigation of direct timing channels, i.e., channels represented directly in the control-flow of the program: S-AT reduces to the underlying command c , but ensures that the command terminates in exactly n steps, where n is the result of evaluating expression e , by delaying further commands until the command delay n has terminated.

3.3.3 Type System

We now describe a type system for Zee, which we will show ensures secure information-flow in Section 3.4. The type system integrates previous work on type safety in stack-based languages [6, 99] with dynamic security labels [145] and existential types for information-flow control [124] into a single language that is able to express complex indirect data dependencies. The typing judgment for commands has the form $\Gamma, \Pi, \phi, pc, fr \vdash c$, and the typing judgment for expressions has the form $\Gamma, \Pi, \phi \vdash e : s$. We explain each component of the judgment before presenting the judgment rules.

Function $\Gamma : \text{Var} \rightarrow s$ maps regular variable names to security types, and similarly $\Pi : \text{Var} \rightarrow \{\mathbf{type}, \mathbf{level}\} \times k$ maps type variables to **type** k and label variables to **level** k . Formula ϕ is a finite conjunction of (possibly negated) flow relations such as $\kappa_1 \sqcup \kappa_2 \sqsubseteq \kappa_3 \wedge \kappa_3 \not\sqsubseteq \kappa_4$. These formulas are gathered during type-checking in such a way that the constructed formulae always represents flows that will be true at runtime. Adding such formulae to the typing relation improves the expressiveness of static information-flow control in the presence of dynamic security labels [69, 144, 145]. Finally, the type system tracks two labels: the program counter label pc and the frame label fr . We now explain the typing judgments involved in typing Zee programs.

$$\begin{array}{c}
\text{S-FP} \\
\frac{v = (\text{cod}(p.\text{arg}), (\text{cod}(p.\text{local}), \text{fp}(m)_v)) \quad m = (\mathbb{I}, |m|, v) \quad m' = m[\delta(x) + \text{fp}(m) \mapsto v]}{\langle x := \text{fp}, m \cdot M, p \cdot P, q \rangle_v \rightarrow \langle \text{stop}, m' \cdot M, p \cdot P, q + 1 \rangle_v} \\
\\
\text{S-LET} \\
\frac{\langle s, p \rangle \Downarrow_{\text{type}} \tau \quad \langle e, m, p \rangle \Downarrow v \quad c' = c; \text{unscope}(x) \quad m' = m[\delta(x) + \text{fp}(m) \mapsto v] \quad p' = p[p_{\text{local}} \mapsto p_{\text{local}}[x \mapsto \tau]]}{\langle \text{let } x : s := e \text{ in } c, m \cdot M, p \cdot P, q \rangle_v \rightarrow \langle c', m' \cdot M, p' \cdot P, q + 1 \rangle_v} \\
\\
\text{S-UNPACK-TY} \\
\frac{\langle e, m, p \rangle \Downarrow (\tau_1, v_2) \quad \langle s, p' \rangle \Downarrow_{\text{type}} \tau \quad p' = p[\text{var} \mapsto p.\text{var}[\alpha \mapsto \tau_1]] \quad p'' = p'[\text{local} \mapsto p'.\text{local}[x \mapsto \tau]] \quad m' = m[\delta(x) + \text{fp}(m) \mapsto v_2]}{\langle \text{let } (\alpha : \mathbf{type} \, k, x : s) := e \text{ in } c, m \cdot M, p \cdot P, q \rangle_v \rightarrow \langle c; \text{unscope}(x), m' \cdot M, p' \cdot P, q + 1 \rangle_v} \\
\\
\text{S-READ} \\
\frac{m_i = (\mathbb{I}, |m_i|, v_i) \in m \cdot M \quad a \in \mathbb{I} \quad v_i \leq \gamma \quad \langle e, m, P \rangle \Downarrow a_\gamma \quad m' = m[\delta(x) + \text{fp}(m) \mapsto m_i(a)]}{\langle x := *e, m \cdot M, P, q \rangle_v \rightarrow \langle \text{stop}, m' \cdot M, P, q + 1 \rangle_v} \\
\\
\text{S-WRITE} \\
\frac{m_i = (\mathbb{I}, |m_i|, v) \quad M = M_1 \cdot m_i \cdot M_2 \quad a \in \mathbb{I} \quad v_i \leq \gamma \quad \langle e_1, M, P \rangle \Downarrow a_\gamma \quad \langle e_2, M, P \rangle \Downarrow v \quad m'_i = m_i[a \mapsto v]}{\langle *e_1 := e_2, M, P, q \rangle_v \rightarrow \langle \text{stop}, M_1 \cdot m'_i \cdot M_2, P, q + 1 \rangle_v} \\
\\
\text{S-AT} \qquad \qquad \qquad \text{S-DELAY} \\
\frac{\langle e, M, P \rangle \Downarrow n}{\langle \text{at } k \text{ with bound } e \text{ do } c, M, P, q \rangle_v \rightarrow \langle c; \text{delay } n, M, P, q + 1 \rangle_v} \qquad \frac{n \leq q}{\langle \text{delay } n, M, P, q \rangle_v \rightarrow \langle \text{stop}, M, P, n + 1 \rangle_v} \\
\\
\text{S-MATCH} \\
\frac{\text{argmin}_{i=1, \dots, n}(\tau \preceq pat_i) = j \quad \langle \alpha, p \rangle \Downarrow_{\text{type}} \tau \quad \llbracket pat_j \rrbracket(p, \tau) = p'}{\langle \text{match } \alpha \text{ with } (pat_i \Rightarrow c_i)_{i=1, \dots, n}, M, p \cdot P, q \rangle_v \rightarrow \langle c_j, M, p' \cdot P, q + 1 \rangle_v}
\end{array}$$

Figure 3.10: Semantics of Zee: commands.

$$\begin{array}{c}
\text{T-NUM} \\
\hline
\Gamma, \Pi, \phi \vdash n : \mathbf{int} \perp \\
\\
\text{T-PACK-TY} \\
\frac{\Pi, \phi \vdash_{\text{type}} s_2 \quad \Gamma, \Pi, \phi \vdash e : s_2[s_1/\chi] \quad \Pi, \phi \vdash_{\text{type}} s_1 : k_1 \quad t = \exists x : \mathbf{type} \ k_1. s_2}{\Gamma, \Pi, \phi \vdash \text{pack}(s_1, e) \text{ as } t : t \perp} \\
\\
\text{T-VAR} \\
\hline
\Gamma, \Pi, \phi \vdash x : \Gamma(x) \\
\\
\text{T-SIZEOF} \\
\frac{\Pi, \phi \vdash_{\text{type}} s : k}{\Gamma, \Pi, \phi \vdash \text{sizeof } s : \mathbf{size}[s] k} \\
\\
\text{T-BINOP} \\
\frac{\Gamma, \Pi, \phi \vdash e_i : s_i \quad s_1 \llbracket \oplus \rrbracket s_2 \rightarrow s}{\Gamma, \Pi, \phi \vdash e_1 \oplus e_2 : s}
\end{array}$$

Figure 3.11: Excerpts of the typing rules for expressions.

Typing Judgment for Expressions

Figure 3.11 shows excerpts of the typing rules for expressions. Rules T-NUM and T-VAR are standard rules for literals and variables, and T-PACK-TY is the standard rule for introducing an existentially quantified type [124]. Finally, rule T-SIZEOF assigns a singleton-type $\mathbf{size}[s]$ to an expression `sizeof s`, representing that the expression will evaluate to the size of the type s at runtime. Such expressions are crucial for secure and type-safe operations in a language with heterogeneous data like in Zee: they allow the type system to track how pointer arithmetic changes the type of the pointer. This becomes clear in rule T-BINOP, which assigns types to the result of binary expressions using the relation $s_1 \llbracket \oplus \rrbracket s_2 \rightarrow s$. This states that applying operator \oplus to expressions of type s_1 and s_2 results in an expression of type s . The full judgment is shown in the Appendix, and excerpts of this relation include

$$\mathbf{int} \ k_1 \llbracket \oplus \rrbracket \mathbf{int} \ k_2 \rightarrow \mathbf{int} \ (k_1 \sqcup k_2) \quad (3.1)$$

$$(s_1 @ s \cdot s_2) \ k_1 \llbracket + \rrbracket \mathbf{size}[s] \ k_2 \rightarrow (s_1 \cdot s @ s_2) \ (k_1 \sqcup k_2) \quad (3.2)$$

$$(s_1 \cdot s @ s_2) \ k_1 \llbracket - \rrbracket \mathbf{size}[s] \ k_2 \rightarrow (s_1 @ s \cdot s_2) \ (k_1 \sqcup k_2) \quad (3.3)$$

In words, performing a binary operation on two integers results in an integer labeled with the join of the two labels (3.1), adding the size of a type s to a pointer pointing to a value of type $s_1 @ s \cdot s_2$ results in a pointer to a value of type $s_1 \cdot s @ s_2$, and the labels are raised accordingly. Dually, one can subtract the size of a type s from a pointer of type $s_1 \cdot s @ s_2$, and obtain a value of type $s_1 @ s \cdot s_2$.

Typing Judgment for Commands

Figure 3.12 shows excerpts of the typing rules for commands. Rule T-LET states that a variable declaration $\text{let } x : s := e \text{ in } c$ is well-typed if the type of e is a subtype of s . The subtype relation is standard for imperative languages for information-flow [83]. To prevent implicit flows, the program counter label should also flow to s . Finally, x is added to the typing context Γ , and the frame label fr is raised to reflect the fact that the frame layout has been influenced by the variable declaration.

Rule T-IF states that a command $\text{if } e \text{ then } c_1 \text{ else } c_2$ is well-typed when e is an expression of type $\text{int } pc$, and both branches can be shown to be well-typed. Readers familiar with IFC type systems may wonder why it is necessary to restrict the label on the type of e to be pc . This is done to facilitate predictive mitigation of direct timing channels: the pc must be explicitly raised using an at command. Rule T-AT states that a command $\text{at } k \text{ with bound } e \text{ do } c$ is well-typed when the label k and the computation time e only depends on information up to pc . Furthermore, the command must not lower the program counter label, and the command c must be well-typed under the raised program counter label k .

The command $x := fp$ is well-typed when x is a subtype of the type $T_{\text{st}}(pc, fr, k)$, which abbreviates the type

$$(\mu \alpha : \text{type } k. (\exists \beta : \text{type } fr. (\exists \gamma : \text{type } fr. (\beta \cdot \alpha @ \gamma) pc) \perp) \perp) \perp$$

This type reflects the layout of the stack at runtime (cf. Figure 3.2). Each frame consists of some type β representing the type of the arguments given to the function, followed by a pointer to the previous stack frame (which is represented as the recursive type variable α), and finally the type γ representing the types of the local variables. By assigning each existentially quantified type the label fr we ensure that no type leaks information, as fr represents the upper bound of the information that can be learned from knowing the value of the types.

Rule T-MATCH states when a command $\text{match } \alpha \text{ with } \overline{pat} \Rightarrow c$ is well-typed. First, α must be a type variable with a label that flows to pc , as the direct timing channels must be controlled using at commands. Judgment $\Pi \vdash pat_i \rightsquigarrow_k \Pi_i : s_i$ generates a type variable environment Π_i for type-checking the command for the i 'th pattern, and the type s_i to assign α in the command. Furthermore, any type and label variable bound by the new typing type variable environment Π_i is bound to the label k . Most of the rules of this judgment are of the form

$$\Pi \vdash \text{int } \kappa \rightsquigarrow_k \Pi[\kappa \mapsto \text{level } k] : \text{int } \kappa$$

which expresses that, in environment Π , when the pattern is **int** κ and the scrutinee depends on information up to label k , the environment is updated to $\Pi[\kappa \mapsto \mathbf{level} \ k]$ and the type of the scrutinee can be assumed to have type **int** κ in the command guarded by the pattern **int** κ . The full judgment can be found in the technical report. Finally, each command c_i is type-checked in the generated type- and variable environment.

Rule T-UNPACK-Ty states when an elimination of an existentially quantified type is well-typed. The rule follows previous work on existential types for security-typed languages [124]: the type r , in which α may appear free, must be a subtype of the declared type s , and to prevent implicit flows the program counter label must also flow to the label on s . Furthermore, the frame label is raised to reflect that two new variables, each of which has a type that may depend on sensitive information, is now part of the frame layout. Finally, the command is type-checked in the updated environments with the raised frame label. Rule T-FlowsTo branches on the runtime relation between the two labels k_1 and k_2 . Each command is checked in the extended formula capturing whether $k_1 \sqsubseteq k_2$ holds at runtime.

Typing Judgment for Types and Labels

The typing judgments for security types and labels are straightforward. Figure 3.13 shows an excerpt of the typing judgment for security types, and the judgment for labels is similar. Rule T-INT says that, if the label k depends on information up to label k' then **int** k depends on information up to label k' as well. Rule T-MU states that a recursive type $(\mu \alpha : \mathbf{type} \ k_1. s) \ k_2$ depends on information up to label k if, assuming α depends on information up to label k , the type s can be shown to depend on information up to k and finally, both k_1 and k_2 must also not depend on information above k .

3.3.4 An Extensible Language

To allow the specification of additional operations in Zee, we include a *hole* $[\cdot]$ command:

$$c ::= \dots \mid [\cdot]$$

We call the language without the hole construct the *base* language, and the additional commands the *instantiation* language. We let c range over commands in the instantiation language, and write \mathcal{C} for the set of commands in the base language. Given an instantiation language \mathcal{D} we write $\mathcal{C}[\mathcal{D}]$ for the set containing commands from both the base language and the instantiation language.¹¹

¹¹Section 3.4.2 formally defines the notion of an instantiation language.

$$\begin{array}{c}
\text{T-LET} \\
\frac{\Gamma, \Pi, \phi \vdash e : r \quad \Pi, \phi \vdash_{\text{type}} s : k \quad \phi \vdash r^{pc} <: s \quad fr' = fr \sqcup k \quad \Gamma[x \mapsto s], \Pi, \phi, pc, fr' \vdash c}{\Gamma, \Pi, \phi, pc, fr \vdash \text{let } x : s := e \text{ in } c}
\end{array}
\quad
\begin{array}{c}
\text{T-AT} \\
\frac{\Pi; \phi \vdash_{\text{lab}} k : pc \quad \Gamma, \Pi, \phi \vdash e : \mathbf{int} pc \quad \phi \vdash pc \sqsubseteq k \quad \Gamma, \Pi, \phi, k, fr \vdash c}{\Gamma, \Pi, \phi, pc, fr \vdash \text{at } k \text{ with bound } e \text{ do } c}
\end{array}$$

$$\begin{array}{c}
\text{T-IF} \\
\frac{\Gamma, \Pi, \phi \vdash e : \mathbf{int} pc \quad \Gamma, \Pi, \phi, pc, fr \vdash c_i \ i = 1, 2}{\Gamma, \Pi, \phi, pc, fr \vdash \text{if } e \text{ then } c_1 \text{ else } c_2}
\end{array}
\quad
\begin{array}{c}
\text{T-FP} \\
\frac{\Pi; \phi \vdash_{\text{lab}} fr : k \quad \phi \vdash T_{\text{st}}(pc, fr, k)^{pc} <: \Gamma(x)}{\Gamma, \Pi, \phi, pc, fr \vdash x := \text{fp}}
\end{array}$$

$$\begin{array}{c}
\text{T-MATCH} \\
\frac{\Pi(\alpha) = \mathbf{type} k \quad \phi \vdash k \sqsubseteq pc \quad \Pi \vdash pat_i \rightsquigarrow_k \Pi_i : s_i \quad \Gamma[s_i/\alpha], \Pi_i[s_i/\alpha], \phi, pc, fr \vdash c_i[s_i/\alpha]}{\Gamma, \Pi, \phi, pc, fr \vdash \text{match } \alpha \text{ with } pat \Rightarrow c}
\end{array}$$

$$\begin{array}{c}
\text{T-UNPACK-TY} \\
\frac{\Gamma, \Pi, \phi \vdash e : (\exists \alpha : \mathbf{type} k_1. r) pc \quad \phi \vdash r^{pc} <: s \quad \Gamma' = \Gamma[x \mapsto s] \quad \Pi' = \Pi[\alpha \mapsto \mathbf{type}_{k_1}] \quad \Pi', \phi \vdash_{\text{type}} r : k_2 \quad fr' = fr \sqcup k_1 \sqcup k_2 \quad \Gamma', \Pi', \phi, pc, fr' \vdash c}{\Gamma, \Pi, \phi, pc, fr \vdash \text{let } (\alpha : \mathbf{type} k_1, x : s) := e \text{ in } c}
\end{array}$$

$$\begin{array}{c}
\text{T-FLOWS TO} \\
\frac{\Pi; \phi \vdash_{\text{lab}} k_i : pc \quad \Gamma, \Pi, \phi \wedge k_1 \sqsubseteq k_2, pc, fr \vdash c_1 \quad \Gamma, \Pi, \phi \wedge k_1 \not\sqsubseteq k_2, pc, fr \vdash c_2}{\Gamma, \Pi, \phi, pc, fr \vdash \text{if } (k_1 \sqsubseteq k_2) \text{ then } c_1 \text{ else } c_2}
\end{array}$$

Figure 3.12: Excerpts of the typing rules for commands.

$$\begin{array}{c}
\text{T-INT} \\
\frac{\Pi; \phi \vdash_{\text{lab}} k : k'}{\Pi, \phi \vdash_{\text{type}} \mathbf{int} k : k'}
\end{array}
\quad
\begin{array}{c}
\text{T-MU} \\
\frac{\phi \vdash k_1 \sqsubseteq k \quad \Pi[\alpha \mapsto \mathbf{type} k]; \phi \vdash_{\text{lab}} s : k \quad \Pi; \phi \vdash_{\text{lab}} k_1 : k \quad \Pi; \phi \vdash_{\text{lab}} k_2 : k}{\Pi, \phi \vdash_{\text{type}} (\mu \alpha : \mathbf{type} k_1. s) k_2 : k}
\end{array}$$

Figure 3.13: Excerpts of the typing relation for security types.

$$\begin{array}{c}
\text{S-LIFT} \\
\frac{\langle c, M, P, q \rangle_v \rightarrow \langle c', m', P', q' \rangle_{v'}}{\langle c, M, P, h, q \rangle_v \rightarrow \langle c', m', P', h, q' \rangle_{v'}} \\
\\
\begin{array}{cc}
\text{S-INST} & \text{T-INST} \\
\frac{\langle \mathbf{c}, M, P, h, q \rangle_v \rightarrow \langle \mathbf{c}', M', P', h', q' \rangle_{v'}}{\langle \mathbf{c}, M, P, h, q \rangle_v \rightarrow \langle \mathbf{c}', M', P', h', q' \rangle_{v'}} & \frac{\Gamma, \Pi, \phi, pc, fr \vdash \mathbf{c}}{\Gamma, \Pi, \phi, pc, fr \vdash \mathbf{c}}
\end{array}
\end{array}$$

Figure 3.14: Extending Zee with rules for modular extensions of the reduction semantics and the typing judgment.

We add a heap to the configuration, which can be modified by the instantiation language. A heap is a partial mapping $h : \mathbb{A} \rightarrow v$ from addresses to values. We write $\text{dom}(h)$ for the set of addresses currently allocated in h . We add an additional rule, S-LIFT, that lifts the semantics of base commands to configurations that include a heap.

Finally, we add a rule for specifying semantics of commands in $\mathcal{C}[\mathcal{D}]$: rule S-INST delegates reduction steps to the small-step semantics of the instantiation language. The new rules are shown in Figure 3.14. We extend the typing judgment with an additional rule T-INST that delegates typing to the typing relation for the instantiation language using the typing judgment \vdash provided by the instantiation language.

3.4 Security Guarantees

In this section we formalize the security guarantees obtained by adhering to the type system described in Section 3.3.3. Section 3.4.1 defines the attacker model, and Section 3.4.2 specifies the semantic interface that each instantiation language must satisfy. Finally, Section 3.4.3 defines *termination-insensitive timing-sensitive noninterference* (TINI) [46, 92] and shows that well-typed programs satisfy TINI.

3.4.1 Attacker Model

To precisely define the security condition we introduce an augmented semantics that adds observable events to the reduction rules. We associate the attacker with a fixed level $\mathcal{A} \in \mathcal{L}$, and now define what an attacker at level \mathcal{A} can observe and which values \mathcal{A} can distinguish.

$$\begin{aligned} ev ::= & \varepsilon \mid \text{rd}(x \leftarrow v, q) \mid [\cdot] \\ & \mid \text{unp}(\ell, x : \tau \leftarrow v, q) \mid \text{let}(x : \tau \leftarrow v, q) \end{aligned}$$

Figure 3.15: Grammar for events.

Events and Event Semantics

The grammar of events is shown in Figure 3.15. We assume that only commands that modify the stack generates an event, but nothing fundamental prevents adding a more fine-grained syntax of events. Event $\text{rd}(x \leftarrow v, q)$ describes obtaining a value v from the stack by reading a pointer, and assigning the value to variable x at time q . Event $\text{unp}(\ell, x : \tau \leftarrow v, q)$ describes an unpack command that declares a type (or level) variable at security level ℓ , and a variable of type τ with the initial value v at time q . Event $\text{let}(x : q \leftarrow v, q)$ describes the declaration of a regular variable x of type τ with initial value v at time q .

As our events capture the time at which the events are emitted, our definition of noninterference is timing-sensitive. Finally, like commands, the language of events can be extended with *instantiation events* using a hole construct $[\cdot]$, and we write the events of the instantiation language as ev .

We denote by \tilde{ev} an *event tuple* of the form (ev, Γ, P) where Γ is the typing environment and P is the private stack, and we define an event semantics $\xrightarrow{\tilde{ev}}$ over configurations that emits event tuples. Finally, we write \xrightarrow{t}^* for the reflexive, transitive closure of the event semantics relation that concatenates all event tuples into a trace t .

Attacker Observability

Given some level $\mathcal{A} \in \mathcal{L}$ we say that $\tau\ell$ is *observable* to \mathcal{A} if $\ell \sqsubseteq \mathcal{A}$, and *invisible* to \mathcal{A} otherwise. We lift observability to events as follows: given an event ev we write $\Gamma, P \vdash ev \sqsubseteq \mathcal{A}$ if \mathcal{A} can observe event ev given typing environment Γ and private stack P . We write $\tilde{ev} \sqsubseteq \mathcal{A}$ if $\tilde{ev} = (ev, \Gamma, P)$ and $\Gamma, P \vdash ev \sqsubseteq \mathcal{A}$. Section 3.4.2 places restrictions on the instantiation events which are necessary for the noninterference theorem to hold.

Attacker Equivalence

We say two values $v_i : \tau_i$ for $i = 1, 2$ are \mathcal{A} -equivalent given private stacks frames p_1 and p_2 , written $p_1, p_2 \vdash v_1 =_{\mathcal{A}} v_2 : \tau_1 \times \tau_2$, if an attacker \mathcal{A} is unable to distinguish them.

$$\lfloor \varepsilon \rfloor_{\mathcal{A}} = \varepsilon \qquad \lfloor \tilde{ev} \cdot t \rfloor_{\mathcal{A}} = \begin{cases} \tilde{ev} \cdot \lfloor t \rfloor_{\mathcal{A}} & \tilde{ev} \sqsubseteq \mathcal{A} \\ \lfloor t \rfloor_{\mathcal{A}} & \text{otherwise} \end{cases}$$

Figure 3.16: \mathcal{A} -projected trace.

We lift \mathcal{A} -equivalence of values to \mathcal{A} -equivalence of events and write $\Gamma \mid p_1, p_2 \vdash ev_1 =_{\mathcal{A}} ev_2$ when events ev_1 and ev_2 are \mathcal{A} -equivalent. The typing environment Γ is needed in the judgment to associate a type with the variable x in the case of an assignment event $\text{asn}(x \leftarrow v, q)$. All judgments are spelled out in the technical appendix.

Given a trace t , Figure 3.16 defines the \mathcal{A} -projected trace $\lfloor t \rfloor_{\mathcal{A}}$ containing only \mathcal{A} -observable events. We say two traces t_1 and t_2 are \mathcal{A} -equivalent, written $t_1 =_{\mathcal{A}} t_2$, if $\lfloor t_1 \rfloor_{\mathcal{A}}$ and $\lfloor t_2 \rfloor_{\mathcal{A}}$ are pairwise \mathcal{A} -equivalent. Finally, given two pairs of exposed and private stack frames (p_i, m_i) we write $\Gamma \vdash (p_1, m_1) =_{\mathcal{A}} (p_2, m_2)$ when an attacker \mathcal{A} is unable to distinguish their content.

We extend this judgment pointwise and obtain an \mathcal{A} -equivalence on exposed and private stacks, which we write as $\Gamma \vdash (P_1, M_1) =_{\mathcal{A}} (P_2, M_2)$.

3.4.2 Specification of an Instantiation Language

In this section we describe the specification that an instantiation language must satisfy. To define the requirements of the instantiation language, we define an augmented small-step semantics

$$\Gamma, \Pi, \phi, pc, fr \vdash C \rightarrow C' : \Gamma', \Pi', \phi', pc', fr'$$

This relation specifies that C steps to C' and updates the typing environments Γ, Π , constraints ϕ and labels pc and fr to $\Gamma', \Pi', \phi', pc'$ and fr' respectively. We lift this relation to the event semantics relation and write $\Gamma, \Pi, \phi, pc, fr \vdash C \xrightarrow{\tilde{ev}} C' : \Gamma', \Pi', \phi', pc', fr'$ when event tuple \tilde{ev} is emitted when evaluating $\Gamma, \Pi, \phi, pc, fr \vdash C \rightarrow C' : \Gamma', \Pi', \phi', pc', fr'$. A wellformedness relation $\Gamma, \Pi, \phi \models C$ in the technical report formalizes wellformed configurations, and given a private stack P and a constraint formula ϕ relation $P \models \phi$ specifying that ϕ is true when evaluating all labels in ϕ in the private stack P .

Formally, an instantiation language \mathcal{D} is a tuple 6-tuple $(\mathbf{c}, \rightarrow, \vdash, ev, =_{\mathcal{A}}, \sqsubseteq)$ where \mathbf{c} is a set of syntactically valid commands, relation \rightarrow is a small-step relation on configurations, and \vdash is a typing judgment. The set ev contains syntactically valid events, and relations $\Gamma \mid p_1, p_2 \vdash ev_1 =_{\mathcal{A}} ev_2$ and $\phi, P \vdash ev \sqsubseteq \mathcal{A}$ defines when two events ev_1 and ev_2 are considered

equivalent by an attacker at level \mathcal{A} , and when an event is observable to a \mathcal{A} respectively.

For the following properties, let \mathbf{c} be a command \mathbf{c} such that $\Gamma, \Pi, \phi, pc, fr \vdash \mathbf{c}$, and let (M, P) and h be a stack and a heap such that $\Gamma, \Pi, \phi \models \langle \mathbf{c}, M, h, P, q \rangle_\nu$ and $P \models \phi$. The following three properties must then be satisfied:

Property 3.4.1 (Single-run reduction properties). *If*

$$\Gamma, \Pi, \phi, pc, fr \vdash \langle \mathbf{c}, M, h, P, q \rangle_\nu \rightarrow \langle \mathbf{c}', M', h', P', q' \rangle_{\nu'} : \Gamma', \Pi', \phi', pc', fr'$$

it holds that $\Gamma \subseteq \Gamma', \Pi \subseteq \Pi', \nu \leq \nu'$, and $\phi' \implies \phi$. Finally it holds that $\Gamma', \Pi', \phi' \models \langle \mathbf{c}', M', h', P', q' \rangle_{\nu'}, P' \models \phi'$ and $\Gamma', \Pi', \phi', pc', fr' \vdash \mathbf{c}'$.

Property 3.4.1 formalizes the type-safety requirements of the instantiation language. Intuitively, the semantics of the instantiation language should preserve wellformedness of configurations (i.e., $\Gamma', \Pi', \phi' \models \langle \mathbf{c}', M', h', P', q' \rangle_{\nu'}$). Furthermore, the semantics should not prevent future use of variables that are already in scope by removing them from the typing environments Γ' or Π' (i.e., $\Gamma \subseteq \Gamma'$ and $\Pi \subseteq \Pi'$). To prevent the possibility of reusing version numbers the version counter ν' should not decrease (i.e., $\nu \leq \nu'$), and finally the semantics must not weaken the constraint formula ϕ' ($\phi' \implies \phi$), but should also not strengthen the formula to the point where it is not guaranteed to hold at runtime (i.e., $P' \models \phi'$).

Property 3.4.2 (Single-step noninterference). *If $\Gamma \vdash (P_1, M_1) =_{\mathcal{A}} (P_2, M_2)$ and $\phi \vdash pc \sqsubseteq \mathcal{A}$ and*

$$\Gamma, \Pi, \phi, pc, fr \vdash \langle \mathbf{c}, M_i, h_i, P_i, q \rangle_\nu \xrightarrow{\widetilde{ev}} \langle \mathbf{c}'_i, M'_i, h'_i, P'_i, q' \rangle_{\nu'} : \Gamma', \Pi', \phi', pc', fr'$$

for $i = 1, 2$ then $\Gamma'_1 = \Gamma'_2, \Pi'_1 = \Pi'_2, \Gamma'_1 \vdash (P'_1, M'_1) =_{\mathcal{A}} (P'_2, M'_2), q'_1 = q'_2$ and $\Gamma'_1 \mid P'_1, P'_2 \vdash \widetilde{ev}_1 =_{\mathcal{A}} \widetilde{ev}_2$.

Property 3.4.2 ensures that a command \mathbf{c} in \mathcal{A} -equivalent environments results in \mathcal{A} -equivalent observations for a single-step.

Property 3.4.3 (Confinement). *If $\phi \vdash pc \not\sqsubseteq \mathcal{A}$ and*

$$\Gamma, \Pi, \phi, pc, fr \vdash \langle \mathbf{c}, M, h, P, q \rangle_\nu \xrightarrow{\widetilde{ev}} \langle \mathbf{c}', M', h', P', q' \rangle_{\nu'} : \Gamma', \Pi', \phi', pc', fr'$$

then $\Gamma \vdash (P, M) =_{\mathcal{A}} (P', M')$ and $\phi, P \vdash \widetilde{ev} \not\sqsubseteq \mathcal{A}$.

Property 3.4.3 ensures that the semantics does not leak sensitive information through indirect flows. That is, when the reachability of a program point depends on sensitive information (i.e., $\phi \vdash pc \not\sqsubseteq \mathcal{A}$), no \mathcal{A} -observable event is emitted, and \mathcal{A} is unable to distinguish the environments before and after the execution of c .

When properties 3.4.1, 3.4.2 and 3.4.3 are satisfied we say that the instantiation language $(c, \rightarrow, \vdash, ev, =_{\mathcal{A}}, \sqsubseteq)$ is a well-formed instantiation language.

3.4.3 Security Guarantees

Finally, we show that well-typed Zee programs satisfy termination-insensitive timing-sensitive noninterference. This definition permits attackers to learn information by observing the termination-behavior of the program, but it does not permit an attacker to learn information due to the timing-behavior of terminating programs. This distinction between termination and timing is unusual compared to previous literature where timing-sensitivity implies termination-sensitivity [70] but in a setting like ours, where a program can fail to terminate in many different ways, i.e., by attempting to read invalid memory or by non-exhaustive pattern matching, this definition is suitable [92].

Theorem 3.4.4 (Soundness). *Let \mathcal{D} be a well-formed instantiation language and let $c \in \mathcal{C}[\mathcal{D}]$, and let Γ, Π be typing environments. Assume $\Gamma, \Pi \vdash c$ and for all function definitions*

$$f(\overline{\kappa : k_1}) \langle \overline{\alpha : k_2} \rangle (\overline{x : s}) \stackrel{fr}{=}_{pc} c_f$$

it holds that $\Gamma_f, \Pi_f, \top, pc, fr \vdash c_f$, where

$$\Gamma_f = \{\overline{x \mapsto s}\} \quad \Pi_f = \{\overline{\kappa \mapsto \mathbf{level} \ k_1}\} \cup \{\overline{\alpha \mapsto \mathbf{type} \ k_2}\}.$$

If $\Gamma \vdash (P_1, M_1) =_{\mathcal{A}} (P_2, M_2)$ and $\langle c, M_i, P_i \rangle \xrightarrow{t_i}^$ for $i = 1, 2$ then $t_1 =_{\mathcal{A}} t_2$.*

Theorem 3.4.4 states that, if each function definition is well-typed, and the command c is well-typed, then executing c with two \mathcal{A} -equivalent stacks will result in \mathcal{A} -equivalent traces t_1 and t_2 .

3.5 Case Studies

This section presents two case studies demonstrating realistic activities of a runtime system for a modern programming language. Both case studies, and all programs presented in the paper, are executable using our prototype implementation.

The first case study is an implementation of a segregated garbage collector (GC) that splits the heap into partitions indexed by security levels from a fixed lattice [92]. The algorithm is a modified version of a mark-and-sweep collector [59], and to the best of our knowledge, is the first GC algorithm that implements the abstract semantics formally proven secure in [92]. The security property obtained from the well-typedness (Theorem 3.4.4) of the GC implementation implies that the timing behavior of the garbage collector does not depend on the memory operations caused by handling sensitive information.

The second case study is an implementation of a simple cooperative thread scheduling algorithm. The security property obtained from the well-typedness of the thread scheduler implies that the scheduling of “public threads” is independent from the presence of threads spawned due to handling of sensitive information.

The programs in the case studies use a syntax more suitable for programming compared to the formal language, but it can be desugared into the core calculus presented in Section 3.3.

3.5.1 Secure Garbage Collection

The job of a GC is to reclaim memory that will not be used in the future by the program. This property is in general undecidable, and GC algorithms instead only reclaim memory that is not *reachable* by the program. The GC implementation is split into two phases: The *marking* phase and the *sweeping* phase. The marking phase starts when the RTE decides that a GC is needed. Our GC implementation is a stop-the-world collector: it stops the execution of the program and marks every heap allocation that is currently reachable by the program. Guaranteeing security and type-safety for such an operation is nontrivial as data of different types, and with different security policies, must be traversed and handled differently depending on the base type of the value, and its security label.

We proceed by defining the syntax of an instantiation language MS : a language for implementing secure mark-and-sweep garbage collectors. We then define the small-step semantics and the typing relation of MS . Finally, we show that MS is a well-formed instantiation language (cf. Section 3.4.2).

The Instantiation Language MS

We instantiate Zee with the instantiation language MS , whose commands are shown in Figure 3.17. We assume an operation on ℓ -indexed partitioning

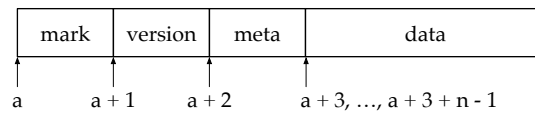
$c ::= \text{mark } e \mid x := \text{is_marked } e \mid \text{free } e \mid x := \text{length } e$
 $\quad \mid x := \text{alloc}(e, e, s, k, k) \mid x := \text{start } k \mid x := \text{next } e$
 $\quad \mid x := \text{read}(e) \mid \text{write}(e_1, e_2)$

Figure 3.17: Instantiation language for mark-and-sweep garbage collection.

of the heap and write $\mathbb{A} \upharpoonright \ell$ for the set of addresses belonging to security level ℓ .

Command $\text{mark } e$ marks a heap address, representing the information that the address is reachable in the heap. Command $x := \text{is_marked } e$ checks if an address, given by the evaluation of e , has previously been marked, and stores this information in variable x . Command $\text{free } e$ reclaims the memory pointed to by e , and $x := \text{alloc}(e_1, e_2, s, k_1, k_2)$ allocates e_1 number of entries, all initialized to the value e_2 , in the heap partition associated with the security level k_2 , and where the label k_1 denotes the sensitivity of the size of the allocation. Command $x := \text{length } e$ stores the length of an allocation pointed to by e in variable x . The final two commands are used in the sweep phase, and implement an abstract notion of heap parsability [59]: command $x := \text{start } k$ stores a pointer to the first allocation in the heap partition associated with the security level k in variable x , and $x := \text{next } e$ stores the next pointer in the same heap partition as e (i.e., the allocation with the smallest address that is larger e) in variable x .

Figure 3.19 shows the small-step semantics for allocation in *MS*. A heap allocation of size n is structured as follows:



That is, the first address stores the *mark* of the allocation, which is used during the marking phase of garbage collection to denote that the allocation is reachable by the program. Next to the mark is the *version* entry, which ensures that it is not possible to read stale values from the heap when addresses are reused. This is similar to the technique described in Section 3.2.3 for values stored on the exposed stack. The *meta* field stores type and label information using existentially quantified labels and types: Specifically, a value of type $\exists \kappa : \mathbf{level} \ell. \exists \alpha : \mathbf{type} \ell. \mathbf{int} \kappa$ is stored. Here, the level ℓ is the index of the partition in which the allocation is stored on the heap. The label κ stores the security label on the size of the array, which is used during the sweep phase, and the type α stores the type information about

```

1 let p : ( $\mathbf{L} \mapsto \text{int} \mathbf{L}$ )  $\mathbf{L} := \text{null}$  in
2 p := alloc( $\mathbf{10}$ ,  $\mathbf{0}$ , int  $\mathbf{L}$ ,  $\mathbf{L}$ ,  $\mathbf{L}$ );
3 free(p);
4 let q : ( $\mathbf{L} \mapsto \text{int} \mathbf{H}$ )  $\mathbf{L} := \text{null}$  in
5 q := alloc( $\mathbf{10}$ , h, int  $\mathbf{H}$ ,  $\mathbf{L}$ ,  $\mathbf{L}$ )

```

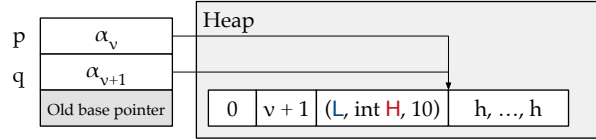


Figure 3.18: Top: a program that attempts to read the secret h through the pointer p when the RTE decides to reuse the recently freed memory for the allocation on line 4. Bottom: stack- and heap layout after executing line 5 of the program.

the type of elements in the array, which is used to traverse the heap during the marking phase, and the integer with security label κ stores the length of the allocation. Finally, the data of the array is stored at the end.

Rule S-ALLOC allocates such a data structure on the heap. First, an address a is found such that the range $a, \dots, a + n + 2$ is free (i.e., not part of the domain of the heap). A pointer to the first element of the array is then stored in stack variable x , and the address is given a fresh version count. Finally, the allocation structure is stored on the heap, and the version counter is incremented.

The version number in each allocation prevents leaks caused by dangling pointers and aliasing. To see an example of this, consider the program in Figure 3.18. On line 2 a pointer to an allocation containing public data is stored in p with a version number v . On line 3 the memory is freed, making it possible for S-ALLOC to reuse the memory from the allocation on line 2. The memory is reused on line 5, causing p to point to secret data, even though the type of p specifies that it points to public data. However, an updated version number $v + 1$ is stored when the memory is being reused, so any attempt to access the secret data through p will fail the version check.

Figure 3.19 also defines the typing judgment: first, the expression e_1 , which denotes the size of the allocation, must be of integer type with label at most k_1 . This ensures that the label argument k_1 correctly captures the security of the size of the allocation at runtime. Similarly, the expression e_2 , which denote the initial value of the array entries, must be typeable at type s , ensuring that the type argument s correctly captures the type information of the array at runtime. Finally, the partition label k_2 must be typeable at

$$\begin{array}{c}
\text{S-ALLOC} \\
\frac{
\begin{array}{l}
a = \text{argmin}([a, \dots, a + n + 2] \cap \text{dom}(h) = \emptyset) \\
a \in \mathbb{A} \text{ s.t. } a, \dots, a + n + 2 \in \mathbb{A} \upharpoonright \ell_2 \\
c = x := \text{alloc}(e_1, e_2, s, k_1, k_2) \quad \langle e_1, m, p \rangle \Downarrow n \\
P = p \cdot P' \quad \langle e_2, m, p \rangle \Downarrow v \quad \langle s, p \rangle \Downarrow_{\text{type}} \tau \\
\langle k_i, p \rangle \Downarrow_{\text{lab}} \ell_i \quad m' = m[\delta(x) + \text{fp}(m) \mapsto (a + 3)_v] \\
h' = \begin{array}{l} h[a \mapsto 0, a + 1 \mapsto v, a + 2 \mapsto (\ell_1, (\tau, n))] \\ \cup \{a + i + 3 \mapsto v \mid i \in 0, \dots, n - 1\} \end{array}
\end{array}
}{
\langle c, m \cdot M, P, h, q \rangle_v \rightarrow \langle \text{stop}, m' \cdot M, P, h', q + 1 \rangle_{v+1}
} \\
\\
\text{T-ALLOC} \\
\frac{
\begin{array}{l}
\Gamma, \Pi, \phi \vdash e_1 : \mathbf{int} \, k_1 \quad \Gamma, \Pi, \phi \vdash e_2 : s \\
\Pi; \phi \vdash_{\text{lab}} k_1 : k_2 \quad \Pi, \phi \vdash_{\text{type}} s : k_2 \\
\Pi; \phi \vdash_{\text{lab}} k_2 : \perp \quad \phi \vdash (k_2 \mapsto s) \, k_1 \sqcup pc <: \Gamma(x)
\end{array}
}{
\Gamma, \Pi, \phi, pc, fr \vdash x := \text{alloc}(e_1, e_2, s, k_1, k_2)
}
\end{array}$$

Figure 3.19: Static and dynamic semantics for allocation. The remaining judgments are defined in the technical report [97].

the label \perp . This guarantees that no information can be learned by knowing which partition the allocation happens in.

The events for MS is defined in the technical report, as well as the rest of the instantiation language. We conclude this section with the following lemma showing that MS satisfies all the requirements stated in Section 3.4.2.

Lemma 3.5.1. *The instantiation language MS is well-formed.*

We now describe how MS can be used to implement a secure mark-and-sweep garbage collector. This case study is developed in a setting of the two-point lattice $\mathbb{L} \sqsubseteq \mathbb{H}$ from Section 3.2.1.

Using the Instantiation Language

Figure 3.20 presents two functions representing the beginning of the marking phase. Function `mark_frames` is invoked by the `gc` function, which is invoked by the runtime.

```

1 gc ⟨pc : L, fr : L⟩()  $\stackrel{pc, fr}{=}$ 
2 let (αargs, e) := unpack FP in
3 let (αlocals, p) := unpack e in
4 let (αargs2, e2) := unpack *(p - sizeof Tst(pc, fr, L)) in
5 let (αlocals2, p2) := unpack e2 in
6 mark_frames⟨pc, fr⟩⟨αargs2, αlocals2⟩(p2); ...

```

Function `gc` starts by reading the frame pointer `FP`, on line 2, to obtain an existentially quantified pointer `e` of type

$$(\exists \alpha_{\text{locals}} : \mathbf{type} \text{ fr}. (\alpha_{\text{arg}} \cdot T_{\text{st}}(pc, fr, L) @ \alpha_{\text{locals}}) pc) L$$

On line 3 `e` is unpacked, revealing the pointer `p` of type

$$(\alpha_{\text{arg}} \cdot T_{\text{st}}(pc, fr, L) @ \alpha_{\text{locals}}) pc$$

pointing to the beginning of `gc`'s stack frame. Lines 4 to 6 then follows the same procedure to obtain a pointer `p2` to the beginning of the previous stack frame (i.e., the function that was executing before the GC occurred). This pointer is then passed to a recursive function `mark_frames` on line 7, which traverses each stack frame. This function is shown in Figure 3.20. On line 3 the function checks if the pointer `p` is non-zero (i.e., we are not at the last stack frame). It marks allocations reachable from the stack frame starting at `p` using the function `mark_frame` on line 4, and computes the pointer to the previous stack frame on lines 5 to 7. On line 8 the function then invokes itself recursively to mark the previous frame starting at `p2`.

Function `mark_frame` is also recursively defined, as it traverses each entry in a single stack frame. On line 3 the function performs runtime type analysis on the type α_{locals} . If the runtime representation of the type is a product type with a pointer type $(\kappa_p \mapsto \alpha) \kappa$ at its head, lines 5 to 10 are executed. Line 5 performs a dynamic “flows to” check¹² to ensure that it is secure to reclaim this allocation [92]. If so, the pointer is read off the stack on line 6, and line 7 marks the pointer if it is non-null. For simplicity we elide the code that marks objects recursively reachable from this object, but the full implementation is available in our technical appendix, and the code is executable using our prototype implementation of Zee.

After having marked all allocations reachable from `q` on line 7, `mark_frame` calculates the offset `n` to the address of the next entry in the stack frame on line 9, and then invokes itself recursively on line 10 with the new address `p + n` as its argument.

¹²The expression $\kappa_1 \equiv \kappa_2$ is shorthand for $\kappa_1 \sqsubseteq \kappa_2 \wedge \kappa_2 \sqsubseteq \kappa_1$.

```

1 mark_frames ⟨pc : L, fr : L⟩⟨αargs : L, αlocals : L⟩
2 (p : (αargs * Tst(pc, fr, L) @ αlocals)pc) =pcfr
3   if p then
4     mark_frame⟨pc, fr⟩⟨αargs, αlocals⟩(p);
5     let (αargs2, e) := unpack
6       *(p - sizeof Tst(pc, fr, L)) in
7     let (αlocals2, p2) := unpack e in
8     mark_frames⟨pc, fr⟩⟨αargs2, αlocals2⟩(p2)
9   else skip

```

```

1 mark_frame⟨pc : L, fr : L⟩⟨αargs : L, αlocals : L⟩
2 (p : (αargs * Tst(pc, fr, L) @ αlocals)pc) =pcfr
3   match αlocals with
4   (κp ↦ α)κ * β ⇒
5     if κp ≡ pc then
6       let q := *p in
7       if q then mark(q); ... else skip
8     else skip;
9     let n := sizeof (κp ↦ α)κ in
10    mark_frame⟨pc, fr⟩⟨αargs, β⟩(p + n)
11  | α * β ⇒
12    let n := sizeof α in
13    mark_frame⟨pc, fr⟩⟨αargs, β⟩(p + n)
14  | _ ⇒ skip

```

Figure 3.20: Snippets from the GC case study: The stack frames are traversed (top) and each frame is traversed looking for pointers into the heap (bottom).

If α_{locals} is not of the form $(\kappa_p \mapsto \alpha)_{\kappa} \cdot \beta$, but is still a product type $\alpha \cdot \beta$, lines 12 and 13 are executed. Line 12 computes the number of addresses that must be skipped in order to skip past the current entry in the stack frame, and line 13 then calls the function recursively with the next address $p + n$ to inspect. Finally, if α_{local} is not a product type, the frame has been completely traversed and line 14 is executed, and the function returns.

3.5.2 Secure Thread Scheduling

Once we have the possibility of allocating memory on the heap, we can use the same instantiation language MS to implement a thread scheduler. Concurrency has received a lot of attention in the literature on language-based security [61, 108, 109, 114, 116, 138], especially in the context of timing-channels. Several authors [61, 108, 116] propose special-purpose thread schedulers designed to close such timing-channels, and in this section we present an implementation of a secure cooperative thread scheduling algo-

rithm. For the purpose of this case study, each function written by the user is assumed to have been rewritten into continuation passing style (CPS), as is standard for many compilers for functional programming languages [7, 60], and defunctionalized into a form that contains no higher-order functions, i.e., closure is an identifier followed by a heterogeneous array of local variables. Each security level from some fixed lattice is associated with a queue of closures, and a thread schedules a function f to be invoked by enqueueing its closure in the queue associated with the program counter label of f . The scheduler is a function `schedule` that receives a queue for each security level, and a bound for how long to run sensitive computation. We have implemented a small security-typed queue datastructure in Zee that supports operations such as checking if the queue is empty, as well as queueing and dequeuing elements. We use a pseudocode-style description of the scheduling algorithm, and refer interested readers to the implementation for a precise description.

```

1 schedule(n : int L,
2   schedL : (L ↦ (∃ α : type L . α) L) L,
3   schedH : (L ↦ (∃ α : type L . α) H) H) =LH
4   while nonempty(schedL) do
5     let (α, proc) := unpack dequeue(&schedL)
6     in run(L)(α)(proc);
7     at H with bound n do
8       if nonempty(schedH) then
9         let (α, proc) := unpack dequeue(&schedH)
10        in run(H)(α)(proc);
11      else skip

```

For simplicity, the initial program counter label is L , meaning that threads with program counter L do not need a bound on their computation time, but the scheduler can only be called when the program counter label is L . The frame label, on the other hand, is set to H allowing any sensitive information to flow to the types of the data, but any attempts to compute information based on types will be assigned the label H .

The thread scheduler executes a *public quanta* (i.e., the execution of one closure in the `schedL` queue), followed by one *secret quanta* (i.e., the execution of one closure in the `schedH`). There are positives and negative things to point out about the design of this scheduler: on the positive side, the bound on secret computations n only needs to bound one quanta, and the function that calls `schedule` does not need to consider the number of closures in `schedH`. On the negative side, to guarantee timing-sensitive noninterference, each public quanta must be followed by n steps of computation, no matter

if there is any secret threads to execute or not. Furthermore, in order to run secret threads, there must also be public threads available, as the while loop terminates when the `schedL` queue is empty. An alternative strategy would be a scheduler that provides a bound on the total computation time on high threads. With this approach secret threads can run without the presence of public threads. Having developed this prototype, we leave the design of a more practical thread scheduler as future work.

3.6 Implementation

We have implemented a type checker and interpreter for Zee in Haskell in about 3600 lines of code, and the case studies consists of about 500 lines each. Providing an instantiation language corresponds to an implementation of a particular type class, and the type checking and evaluation of instantiation language constructs is delegated to the relations provided by the instantiation language, similar to how the judgments in the paper are defined. The implementation can be found at: <https://www.dropbox.com/s/bl2juns8nqukqhu/zee.zip>.

3.7 Related Work

Our work on securing runtime environments combines previous efforts of memory safety for unsafe programming languages, extensible reasoning about type systems and information-flow control. We review the relevant literature in each of these classes separately.

3.7.1 Stack Typing and Memory Safety

The work on typed assembly languages initiated by Morrisett et al. [80] paved the way for type systems for low-level programming languages. As the target language was expressed in continuation-passing style, there was no need for a stack. Morrisett et al. [79] introduced local stack variables, but as the goal of Morrisett et al. is type preserving compilation it does not support reasoning about stack traversal, and so the “previous frame pointer” is not available on the stack for accessing the previous stack frames. For this reason Morrisett et al. do not consider runtime type analysis.

Our stack typing discipline is inspired by the bunched adjacency logic of Ahmed and Walker [6]. They use logic formulae more^{\leftarrow} and $\text{more}^{\rightarrow}$ to describe the type of an infinite sequence of locations that increases “to the left” and “to the right” respectively, similar to our use of stack pointer types.

Our version-based enforcement mechanism is inspired by CETS’s [87] identifier-based temporal checking. CETS is a program transformation that adds temporal memory safety checking capable of detecting dangling pointer dereferences and double frees errors at runtime. Our version-based enforcement mechanism could be replaced with static reasoning about regions. Regions were introduced by Tofte and Talpin [122] and later used to provide memory safety for a safe dialect of C [50]. We believe the use of regions is orthogonal to our choice of a dynamic enforcement mechanism.

3.7.2 Attacks on Runtimes

The work on observational determinism by Zdancewic and Myers [138] contains a detailed collection of common scheduler-related attacks. Other parts of the RTE that has been attacked include garbage collectors [92]. Pedersen and Askarov [92] present a series of attacks on the garbage collectors of the Java virtual machine and the V8 JavaScript engine, and design a type system and a small-step semantics for a high-level language with automatic memory management for which they prove a noninterference result similar to ours. Finally, Vassena et al. [127] present attacks that combine concurrent execution and lazy evaluation for leaking sensitive information. They propose a new construct for Haskell called *lazydup*, which lazily duplicates thunks on the heap when entering secret contexts (i.e., when the program counter label is **H**, as they only consider a two-point lattice).

3.7.3 Securing Runtimes

Vassena et al. [128] present a new foundation for a dynamic information flow control parallel runtime system. The goal of their work is securing the execution platform of LIO [115], a dynamic information flow control library for Haskell. Similar to our work, Vassena et al. consider a setting in which an attacker can obtain the current global time as a natural number counting execution steps, and (unlike our model) the current size of the heap. They design a system for hierarchically managing space and time resources with some amount of burden on the programmer: a parent thread has to manually kill their child thread to reclaim resources. Their end goal is an implementation of a modified GHC runtime system, but such a modified runtime has yet to be implemented.

The work by Sabelfeld and Sands [108] contains an interesting observation:

Abstractly we will take a scheduler to be a mechanism for selecting threads which itself satisfies some noninterference property, i.e., its behaviour is independent of high data.

This is exactly the approach we have taken: a thread scheduler is a program written in our language, and Theorem 3.4.4 proves that, since this program is well-typed, its public observable behavior is independent of high data.

3.7.4 Static Information-Flow Control

There is a large body of literature focusing on static information flow control, starting with the seminal work by Denning and Denning [34] and later formulated as a type system by Volpano et al. [130]. Sabelfeld and Myers [107] survey the different enforcement techniques and security definitions. Zheng and Myers [145] introduce the technique of including a formulae expressing which flows are guaranteed to hold at specific program points, allowing for static reasoning about information flow policies that vary at runtime. The use of existentially quantified labels is introduced by Tse and Zdancewic [124], and we follow the same typing discipline for such values. Dependent type systems for IFC has also been explored by Lourenço and Caires [69], Zhang et al. [144] and Gregersen et al. [49].

3.7.5 Verified Runtimes

There has been much work on verifying runtime system components such as garbage collectors [22, 45, 71] and thread schedulers [51] using program logics. We view our work complementary to these efforts. The constructs needed for implementing secure runtime environments, that we identify in this work, may serve as guidelines when applying enforcement techniques different from our type system. Additionally, a program logic may be used to verify the requirements of the instantiation languages used in Zee.

3.8 Conclusion

We have shown that IFC techniques can be used to secure low-level languages with sufficient expressivity to implement runtime environments. To do this, we have designed and implemented Zee, a language with a static IFC type system. We have used Zee to implement two realistic case studies demonstrating the usefulness of the language, and the permissiveness of the type system. We have proven that well-typed programs written in Zee satisfies termination-insensitive timing-sensitive noninterference.

Chapter 4

Programming with Flow-Limited Authorization: Coarser is Better

MATHIAS V. PEDERSEN, AARHUS UNIVERSITY, DENMARK

STEPHEN CHONG, HARVARD UNIVERSITY, CAMBRIDGE, MASSACHUSETTS

In Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P), 2019.

Abstract

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

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

We present Flamio, an instantiation of FLAM in a language with coarse-grained dynamic information-flow control which naturally lends itself to dynamic enforcement techniques. In our implementation of Flamio, the FLAM proof search rules for deriving trust relationships are implemented as regular Flamio

computations: the IFC requirements for FLAM proof rules are a natural fit with coarse-grained information-flow mechanisms. Flamio even supports remote procedure calls, and thus seamlessly supports FLAM’s distributed proof search.

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

4.1 Introduction

Most modern systems require some form of authorization to control access to data. Such authorization mechanisms tend to be complex and hard to get right, even though the correctness of such components is vital for the security of the system and its users [42]. The behavior of such systems is often dynamic, with access control continually changing on a per-user basis. Thus, the authorization mechanisms used in these systems also need to be dynamic and able to securely control changes to user privileges [42] at runtime. Furthermore, the mere existence of a trust relationship between two principals may leak confidential information to an attacker if the trust relationship was established based on the result of a confidential computation. Dually, if attackers can provide trust relationships that influence otherwise high-integrity computations, they may be able to influence access control decisions inappropriately. These issues are especially ubiquitous in distributed systems, where nodes may not agree on the trust relationship among principals.

The Flow-Limited Authorization Model (FLAM) [9] is an expressive security model designed for rigorous reasoning about dynamic changes to authorization policies in a distributed setting, where nodes can forward trust checking requests to other nodes in the system. FLAM also guarantees that no confidential information is leaked to an attacker through the trust checking mechanism and that security principals cannot inappropriately influence the trust relationships. This makes FLAM ideal for highly dynamic security policies involving many principals with intricate trust relationships. However, currently, no programming language that builds on top of FLAM reaps the full benefit of the authorization logic.

In this paper we introduce Flamio, which takes a coarse-grained approach to information-flow control (IFC). Fine-grained IFC (as seen in information-

flow aware languages like FlowCaml [103] and Jif [83]) labels individual values with security labels. By contrast, coarse-grained IFC does not label individual values, but instead labels the *computational context* in which the program is running with a single label. Coarse-grained IFC lends itself naturally to dynamic enforcement techniques as demonstrated by its success in both operating systems [37, 64, 140, 141] and programming languages [26, 27, 116, 117].

We show that FLAM’s rules for proving trust relationships have a straightforward encoding in the coarse-grained IFC setting of Flamio, which is inspired by the work on Labeled IO (LIO) [117]. As evidence of the straightforward encoding, we implement Flamio as a Haskell library [95] that encapsulates FLAM’s proof search for trust relationships in an information-flow aware computational context, which ensures that the proof search itself does not leak confidential information to, and cannot be inappropriately influenced by, attackers. Leveraging FLAM’s decentralized authorization model, Flamio supports distributed proof search of trust relationships, where nodes can forward trust checking to other nodes in the system. To demonstrate the usefulness of distributed proof search of trust relationships in a setting with dynamic security policies we present three case studies involving distributed computations with confidential information and mutually distrusting principals that must cooperate to perform their tasks.

The case studies also demonstrate a novel technique for mitigating the problem of *label creep* during proof search. Label creep refers to the label of the computational context *creeping* up the information-flow lattice as the program executes. Previous work [117] mitigates label creep using the `toLabeled` construct. However, we cannot use this technique to mitigate label creep during search for proofs of trust relationships. We present a technique that gives the programmer fine-grained control over the way proofs of trust are derived, and how the proofs can affect the label on the computational context, providing the programmer with the ability to control label creep during proof search.

We also present a calculus for Flamio, which formally proves that Flamio enforces a noninterference-based [46] security guarantee that attackers cannot leak or corrupt information, despite the incorporation of FLAM as an expressive, dynamic mechanism to state and reason about trust relationships.

This paper makes the following contributions.

- We show how the FLAM principal lattice integrates cleanly into a language with coarse-grained dynamic information-flow control and distributed trust checking.

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

The rest of the paper is structured as follows. Section 4.2 introduces the necessary concepts from FLAM and LIO, and Section 4.3 demonstrates how Flamio combines FLAM and LIO through several examples. Section 4.4 describes the formal calculus for Flamio and how the FLAM judgment for deciding trust relations is modeled in a coarse-grained setting. Section 4.5 defines the attacker model we consider in this work and presents the formal security guarantees offered by Flamio. Section 4.6 discusses the implementation in Haskell and presents three case studies demonstrating the use of Flamio. Section 4.7 presents related work and Section 4.8 concludes.

4.2 Background on FLAM and LIO

Before discussing how FLAM and LIO fit together, we briefly introduce each separately. We first highlight the essential parts of FLAM necessary for this work; more details can be found in Arden et al. [9]. Similarly, we highlight the essential parts of LIO, and additional details can be found in Stefan et al. [117].

4.2.1 The FLAM Principal Lattice

Figure 4.1 describes the syntax of FLAM principals. The grammar is parametric in a set \mathcal{N} of names representing principals like Alice and Bob. Given a principal p , FLAM can talk about the confidentiality and integrity of p using *basis projections* p^{\rightarrow} and p^{\leftarrow} respectively. The principal p^{\rightarrow} represents the authority to learn anything that p can learn, and p^{\leftarrow} represents the authority to modify anything that p can modify. Given principals p and q , FLAM can also represent the authority of *both* p and q as $p \wedge q$ or the authority of *either* p or q as $p \vee q$. This forms a lattice (\mathcal{P}, \succsim) with a partial order \succsim (pronounced *acts for*), that represents trust: if $p \succsim q$ principal p is allowed to act on behalf of q , i.e., q trusts p to act on its behalf. Principals \perp and \top

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

Figure 4.1: Syntax of FLAM

represent the least and most trusted principals respectively, and operations \wedge and \vee are the lattice join and meet operations respectively. The principals in \mathcal{P} are given by the grammar in Figure 4.1.¹ Besides basis projections \rightarrow and \leftarrow , FLAM also defines *ownership projections* $o:p$ representing the same authority as the principal p , but where the owner o controls which principals $o:p$ trusts. For example, the delegation $\text{Acme:Bob} \succcurlyeq \text{Acme}$ implies that Bob can act for Acme, but it is not the case that Alice can act for Acme:Bob , even if Alice acts for Bob. We will use ownership extensively in the example presented in Section 4.6.2.

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

$$p \sqsubseteq q \triangleq q \rightarrow \wedge p \leftarrow \succcurlyeq p \rightarrow \wedge q \leftarrow \quad (4.1)$$

$$p \sqcup q \triangleq (p \wedge q) \rightarrow \wedge (p \vee q) \leftarrow \quad (4.2)$$

$$p \sqcap q \triangleq (p \vee q) \rightarrow \wedge (p \wedge q) \leftarrow \quad (4.3)$$

That is, $p \sqsubseteq q$ (pronounced p flows to q) if q acts for the confidentiality of p , and p acts for the integrity of q . So if $p \sqsubseteq q$, information labeled with principal p can safely be relabeled to principal q , as q is at least as confidential, and has no more integrity, than p . The *join* of p and q , written $p \sqcup q$ is defined as the principal with the authority of both p 's and q 's confidentiality, and the authority of either p 's or q 's integrity. The *meet* of p and q , written $p \sqcap q$ is defined dually as the confidentiality of either p or q , and the integrity of both p and q . This forms a lattice $(\mathcal{P}, \sqsubseteq)$ with the partial order \sqsubseteq , where the bottom element $\perp_{\sqsubseteq} \triangleq \perp \rightarrow \wedge \top \leftarrow$ represents the least confidential and most trusted principal, and the top element $\top_{\sqsubseteq} \triangleq \top \rightarrow \wedge \perp \leftarrow$ represents the most confidential and least trusted principal.²

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

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

Voice of a principal Finally, FLAM defines the *voice* of a principal p , denoted $\nabla(p)$, as the minimum integrity needed to influence the flow of information labeled p . Using the voice operator, FLAM defines a *speaks for* relation [2, 65] between principals as “principal p speaks for principal q if p acts for the voice of q ”. Formally p speaks for q if $p \succcurlyeq \nabla(q)$. Any principal is equivalent to the conjunction of a confidentiality projection and an integrity projection, i.e., $\forall p. \exists q, r. p \equiv q^{\rightarrow} \wedge r^{\leftarrow}$. The voice of a principal p with normal form $q^{\rightarrow} \wedge r^{\leftarrow}$ is then defined as $\nabla(p) = q^{\leftarrow} \wedge r^{\leftarrow}$ [9].

4.2.2 Coarse-Grained Information Flow using LIO

LIO [117] is a Haskell library for dynamic information-flow control. LIO is parametric in the label model and takes a coarse-grained approach to information-flow using a *floating label model*: instead of attaching a label to each value in the program, the *computational context* is protected with a single label called the *current label*. Throughout the execution of the program the current label will “float up” the information-flow lattice as more confidential (or less trustworthy) information is brought into the computational context. The current label restricts what data can be modified, ensuring that non-confidential side effects do not depend on confidential information, and dually that trusted effects do not depend on untrusted information.

The type of LIO computations gives rise to a monad [131] that encapsulates raising the current label. The monadic structure of LIO makes programming with it convenient in Haskell. Specifically, the operation `return e` embeds a pure expression e into the LIO computational context, and the operation $e_1 \gg= e_2$ (pronounced *bind*) chains together monadic LIO operations e_1 and e_2 . Throughout the paper, we use Haskell’s *do notation* in examples, which can be desugared into the calculus presented in Section 4.4.

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

Labeled values In addition to protecting every value in the computational context by a single label ℓ_{cur} , LIO also allows computations to associate explicit labels with particular values. This allows computations to handle data

of different labels in the same context. The type of labeled values of type a is written in Haskell as `Labeled l a`, where l is the type of labels.

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

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

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

Finally, `labelOf` extracts the label of a labeled value. This operation does not return a value in the LIO monad, and thus no information-flow checks are performed when invoking `labelOf`. In other words, the label is protected only by the current label [117]. This fact is important when we define trust checking using strategies in Section 4.4.3.

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

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

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

4.3 Introduction to Flamio by Example

In this section we informally introduce Flamio and the concepts needed to combine LIO and FLAM. Section 4.4 will then formalize the intuitions

presented in this section. We proceed by demonstrating the usefulness of Flamio in the context of a secure, decentralized banking application, which we also discuss in Section 4.6. The intended security policy of the bank is that a user u can transfer money from an account only if the owner of the account trusts u .

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

4.3.1 Remote Procedure Calls

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

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

```

1  login =  $\lambda_{\text{bank}}^{\text{bank}} u p . \text{if } \text{checkCredentials } u p$ 
2          then return  $u$  else return  $\perp$ 

```

is evaluated on the node `bank`, and returns a principal labeled with the integrity of the bank, which represents an access token. The expression `login Alice "password"` evaluates to a value `Alice` labeled with the principal bank^\leftarrow if Alice's password is "password", and \perp labeled with bank^\leftarrow otherwise. As

³We follow previous conventions [9] and omit projections of the \perp principal in the remainder of the paper.

the principal $Alice^{\leftarrow}$ (i.e., the current label of node Alice) does not satisfy $Alice^{\leftarrow} \sqsubseteq bank^{\leftarrow}$, Alice cannot grant herself access because the semantics does not allow her to label values with the principal $bank^{\leftarrow}$. However, Alice can unlabel the returned access token, as it holds that $bank^{\leftarrow} \sqsubseteq Alice^{\rightarrow}$ (i.e., the clearance label of node Alice). So while Alice cannot forge new access tokens, she is free to inspect whether the token grants her access to the bank.

4.3.2 Proof Search

In many practical scenarios, proving trust relationships of the form $p \succcurlyeq q$ requires distributed knowledge spread across multiple nodes. In this section, we will see examples of this, as well as how distributed proof search of trust relationships in Flamio is implemented. The three most important aspects of the proof search are using delegations, managing delegations using *strategies* and forwarding trust checking to other nodes. After discussing these concepts, we apply them in the context of the decentralized banking application. We keep the discussion informal and defer precise formulations of the concepts until Section 4.4.

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

Strategies As delegations are implemented as labeled values, using a delegation in a proof search raises the current label by the label of the delegation, similar to how LIO unlabels labeled values. This makes fine-grained control of how delegations are used important to avoid unnecessary label creep. One approach to such control could be to always unlabel all delegations whose label flows to the clearance label of the node performing the proof search. This would be correct and secure (because all delegations whose label flows to the node's clearance label are used in the proof search), but it may raise the current label unnecessarily (i.e., the proof search may examine more delegations than it needs to). In particular, if a delegation is examined, the current label must be raised, even if the delegation is not ultimately used

in the proof. Thus if a node has a delegation labeled with a very restrictive label, then all proof searches would be tainted by that label. An alternative approach would be a proof search algorithm that unlabels just the right delegations to prove the query. However, as delegations are labeled values, the body of the delegation cannot be inspected without unlabeling the value and raising the current label. So an algorithm has to decide whether to unlabel a delegation by inspecting only the label of the delegation and not its body.

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

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

```

1  ( $\lambda_{Alice^{\rightarrow}}^{Alice}$  - . do h <- unlabel aliceSecret
2                      if h then  $p \succcurlyeq q$  else return 0) ()
```

If `aliceSecret` is labeled with $Alice^{\rightarrow}$, the `unlabel` operation will raise the current label ℓ_{cur} to $\ell_{cur} \sqcup Alice^{\rightarrow}$ tracking the fact that the computation context contains information at level at most $\ell_{cur} \sqcup Alice^{\rightarrow}$. Upon forwarding the

```

1  transfer =  $\lambda_{\perp}^{\text{bank}}$  tok  $u_{\text{from}}$   $u_{\text{to}}$  n . do
2      u <- unlabel tok
3      withStrategy [ $\text{bank}^{\rightarrow} \wedge u_{\text{from}}^{\leftarrow}$ ]
4      (if u  $\succcurlyeq$   $u_{\text{from}}$  then transfer#  $u_{\text{from}}$   $u_{\text{to}}$  n
5      else return ())

```

Figure 4.2: Secure transfer of n dollars from u_{from} to u_{to} using access token tok.

query $p \succcurlyeq q$ to another node m , Alice reveals that `aliceSecret` contained a non-zero value. Checking that Alice’s current label flows to m^{\rightarrow} ensures that the forwarding of the query occurs only if m is allowed to learn this information.

Returning to the secure banking application, consider the implementation of transfer in Figure 4.2, which transfers n dollars from user u_{from} to u_{to} using access token tok.

First, the access token tok acquired by login is unlabeled, revealing the identity of the caller. Then, a check is done to ensure that the caller can act on behalf of the user from which money is being transferred. For this example the bank requires the trust relationship to be observable by the bank, and having integrity of the principal from which the money is transferred, which is reflected in the strategy specifying that only delegations at level $\text{bank}^{\rightarrow} \wedge u_{\text{from}}^{\leftarrow}$ or below (i.e., weaker confidentiality and stronger integrity) are used. Finally, a primitive function `transfer#` performs the actual transfer of money, or if the trust relationship cannot be established, the function returns a unit value. Figure 4.3 demonstrates how Bob can transfer money from Alice if she grants Bob the appropriate trust. First, Alice adds a delegation specifying that Bob can act on behalf of Alice and that this information has the integrity of Alice. Then, Bob performs an RPC to the `login` function, which returns an access token. Finally, he transfers 50 dollars from Alice’s account to his own account. Note that the use of the strategy $[\text{bank}^{\rightarrow} \wedge u_{\text{from}}^{\leftarrow}]$ in `transfer` ensures not only that the bank uses only delegations it is allowed to observe, but also that the bank does not use a delegation provided by an inappropriate principal such as Bob, as Bob cannot add a delegation labeled with the principal $\text{Alice}^{\leftarrow}$ (unless Alice first delegates trust to Bob).

4.4 A Calculus for Flamio

This section introduces a formalization of the Flamio language. We first introduce the syntax and semantics of the language and afterward present


```

1  ( $\lambda_{\perp}^{\text{Alice}} \_ . \text{do assume Bob} \succ \text{Alice @ Alice}^{\leftarrow}$ 
2      ( $\lambda_{\perp}^{\text{Bob}} \_ . \text{do tok} <- \text{login Bob "password"}$ 
3          transfer tok Alice Bob 50) ()) ()

```

Figure 4.3: Alice grants Bob access to perform transfers on her behalf.

the judgment for deriving trust between principals in the language. Finally, this section presents a standard type system for Flamio that guarantees basic type safety.

4.4.1 Syntax

Figure 4.4 shows the syntax of Flamio. The meta-variable v ranges over values, which include boolean literals, the unit value, runtime representations of principals, locations and variables, abstractions $\lambda_{s@p}^n x. e$ (which are parameterized over the node n on which to evaluate the expression e , and the principal p representing the label on the value returned from invoking the abstraction, and s : the type of e), and products. We will omit the type annotation on abstractions in examples. Finally, the language includes lists using `nil` and `::` (pronounced *cons*) to express strategies.

Expressions are ranged over by the meta-variable e and include terms, applications, projections, elimination of booleans and lists, recursive functions, monadic expressions using `return` and `>>=`, allocation, reading and writing of references. Following LIO, the language supports operations for labeling and unlabeling expressions, and the operation `toLabeled` for controlling label creep. The operations `getLabel` and `getClearance` returns the current label and clearance of the computational context respectively, and `labelOf` returns the label of a labeled value. The operation `withScope e` creates a new scope for delegations. The scoping for delegations have dynamic extent [77], i.e., the delegations added during the evaluation of e are visible until evaluation of e finishes. However, unlike traditional dynamic scoping, the delegation is removed once the evaluation of e terminates. This helps ensure safe and correct use of delegations.

Strategies are introduced using the `withStrategy estrat e` operation, which introduces a new strategy e_{strat} for the evaluation of e . Similar to the scoping of delegations, strategies have dynamic extent, and the current strategy can be obtained using `getStrategy`. Finally, delegations can be added, and trust relationships can be queried. The shaded regions describe syntax that is not part of the surface language but rather constructs used during evaluation. We explain these constructs as we describe the evaluation rules for

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

Figure 4.4: The Flamio language

expressions in Section 4.4.2.

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

4.4.2 Semantics

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

Local semantics A structured operational semantics defines the local reduction rules with evaluation contexts [39]. We elide the definition of evaluation contexts, as this is mostly standard for a call-by-name calculus.

We further split the local reduction rules into two categories: pure reduction and monadic reduction. Pure reduction rules $e \rightarrow e'$ reduce expressions independently of the store and of which node is evaluating the expression. They are given in Figure 4.5. Pure reductions include injecting terms into monadic contexts, monadically binding terms, recursive applications, projecting pairs, eliminating booleans and lists and obtaining the label of a labeled value.

$$\begin{array}{l}
 \text{return } v \longrightarrow (v)^{\text{LIO}} \quad (v)^{\text{LIO}} \gg= e \longrightarrow e \ v \quad \text{fix } e \longrightarrow e \ (\text{fix } e) \\
 \pi_i (e_1, e_2) \longrightarrow e_i \quad \text{if } b \text{ then } e_{\text{true}} \text{ else } e_{\text{false}} \longrightarrow e_b \\
 \text{case nil of } e_1 \ e_2 \longrightarrow e_1 \quad \text{case } (e_{\text{hd}} :: e_{\text{tl}}) \text{ of } e_1 \ e_2 \longrightarrow e_2 \ e_{\text{hd}} \ e_{\text{tl}} \\
 \text{labelOf } (e @ p) \longrightarrow p \quad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}
 \end{array}$$

Figure 4.5: Pure reductions for Flamio.

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

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

The monadic small-step relation is defined by the judgment:

$$n; \Sigma \vdash \langle \phi \mid es \rangle \longrightarrow \langle \phi' \mid es' \rangle : \sigma$$

It is read as “the global environment is Σ and node n performs a single reduction and updates its local environment to σ ”. Figure 4.6 and 4.7 show the local monadic reduction rules. Many of the rules verify some trust relationship between principals, written $n; \Sigma \vdash p \succ q : \ell$ and is read as “node n proves that p acts for q using delegations labeled up to ℓ ”. We discuss this judgment in Section 4.4.3.

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

Rules E-NEW, E-READ, E-WRITE, E-LABEL and E-UNLABEL are all equivalent to the ones presented in LIO [117], but now also take into account the possible information-flows arising via deriving trust relationships [9]. Rule E-TO LABELED-1 saves the current label $\Sigma(n).lbl$, and evaluates e using the E-CTX rule. Once e has evaluated to a value E-TO LABELED-2 restores the current label and labels the value with the given label q . This presentation of `toLabeled` is different from the original formulation of LIO [117] and avoids interleaving small-step and big-step operations.

Rules E-ACTS-FOR-TRUE and E-ACTS-FOR-FALSE query the trust relationship between two given principals using the trust judgment, which we explain in Section 4.4.3. The result depends on which delegations are in scope, and on the current strategy. A new scope for delegations is created using E-WITH-SCOPE which evaluates an expression e in a new scope, and once e has reduced to a value, rule E-RESET-SCOPE eliminates the scope. Similarly, rule E-WITH-STRATEGY introduces a new strategy and evaluates an expression in the scope of the new strategy. Once the expression has reduced to a value, rule E-RESET-STRATEGY resets the strategy back to its previous value. Finally, E-ASSUME adds a new delegation. The rule uses the operator ∇ defined in Section 4.2 to ensure that the computational context has sufficient integrity to delegate trust on behalf of q .

Figure 4.8 demonstrates the usefulness of delegation scoping with dynamic extent: On line 2, Alice invokes a function on Bob's node that, on line 5, grants another function (given as an argument) the authority to read Bob's confidential information. In addition, on line 9, Bob enforces the policy that the function is only called once. Due to the use of `withScope` on line 4, the additional authority is given only to the function passed as an argument and only when it is invoked at that point. Any other function cannot read Bob's confidential information.

Global semantics A global configuration is a triple (ns, Σ, S) consisting of a stack of nodes $ns \in \mathcal{N}^*$ representing the RPC call-stack, a global environment Σ , and a mapping S from nodes to local configurations. Figure 4.9 presents the reduction rules for global configurations. Rule G-STEP-LOCAL

$$\begin{array}{c}
\text{E-LIFT-PURE} \\
\frac{es \longrightarrow es'}{n; \Sigma \vdash \langle \phi \mid es \rangle \longrightarrow \langle \phi \mid es' \rangle : \Sigma(n)}
\end{array}
\qquad
\begin{array}{c}
\text{E-CTX} \\
\frac{n; \Sigma \vdash \langle \phi \mid e \rangle \longrightarrow \langle \phi' \mid e' \rangle : \sigma}{n; \Sigma \vdash \langle \phi \mid E[e] \rangle \longrightarrow \langle \phi' \mid E[e'] \rangle : \sigma}
\end{array}$$

$$\begin{array}{c}
\text{E-GET-LABEL} \\
\frac{\ell = \Sigma(n).lbl}{n; \Sigma \vdash \langle \phi \mid \text{getLabel} \rangle \longrightarrow \langle \phi \mid \text{return } \ell \rangle : \Sigma(n)}
\end{array}
\qquad
\begin{array}{c}
\text{E-GET-CLEARANCE} \\
n; \Sigma \vdash \langle \phi \mid \text{getClearance} \rangle \longrightarrow \langle \phi \mid \text{return } n^{\rightarrow} \rangle : \Sigma(n)
\end{array}$$

$$\begin{array}{c}
\text{E-APP} \\
\frac{\begin{array}{l} n; \Sigma \vdash \Sigma(n).lbl \sqsubseteq p \sqsubseteq n^{\rightarrow} : \ell \\ e' = e_1[e_2/x] \gg= \lambda_s^n x. x @ p \\ n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid (\lambda_{s@p}^n x. e_1) e_2 \rangle \longrightarrow \langle \phi \mid e' \rangle : \sigma}
\end{array}
\qquad
\begin{array}{c}
\text{E-NEW} \\
\frac{\begin{array}{l} a \notin \text{dom}(\phi) \quad n; \Sigma \vdash \Sigma(n).lbl \sqsubseteq p \sqsubseteq n^{\rightarrow} : \ell \\ n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell \sqsubseteq n^{\rightarrow} \quad \phi' = \phi [a \mapsto e @ p] \\ \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid \text{new } p \ e \rangle \longrightarrow \langle \phi' \mid \text{return } a \rangle : \sigma}
\end{array}$$

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

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

$$\begin{array}{c}
\text{E-TO LABELED-1} \\
\frac{q = \Sigma(n).lbl \quad e' = \text{toLabeled}_q \ p \ e}{n; \Sigma \vdash \langle \phi \mid \text{toLabeled } p \ e \rangle \longrightarrow \langle \phi \mid e' \rangle : \Sigma(n)}
\end{array}
\qquad
\begin{array}{c}
\text{E-TO LABELED-2} \\
\frac{\sigma = \Sigma(n) [lbl \mapsto p]}{n; \Sigma \vdash \langle \phi \mid \text{toLabeled}_p \ q \ v \rangle \longrightarrow \langle \phi \mid \text{label } q \ v \rangle : \sigma}
\end{array}$$

Figure 4.6: Monadic reductions for Flamio.

$\frac{\text{E-ACTS-FOR-TRUE} \quad \begin{array}{l} n; \Sigma \vdash p \succcurlyeq q : \ell \quad n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid p \succcurlyeq q \rangle \longrightarrow \langle \phi \mid \text{return true} \rangle : \sigma}$	$\frac{\text{E-ACTS-FOR-FALSE} \quad \begin{array}{l} n; \Sigma \vdash p \succcurlyeq q : \text{fail} \quad \ell = n^{\rightarrow} \sqcap \bigsqcup_{s \in \Sigma(n).strat} s \\ \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid p \succcurlyeq q \rangle \longrightarrow \langle \phi \mid \text{return false} \rangle : \sigma}$
$\frac{\text{E-WITH-SCOPE} \quad e' = \text{resetScope}_{\Sigma(n).\Delta}(e)}{n; \Sigma \vdash \langle \phi \mid \text{withScope } e \rangle \longrightarrow \langle \phi \mid e' \rangle : \Sigma(n)}$	$\frac{\text{E-RESET-SCOPE} \quad \sigma = \Sigma(n) [\Delta \mapsto \Delta']}{n; \Sigma \vdash \langle \phi \mid \text{resetScope}_{\Delta'}(v) \rangle \longrightarrow \langle \phi \mid v \rangle : \sigma}$
$\frac{\text{E-WITH-STRATEGY} \quad \begin{array}{l} e' = \text{resetStrategy}_{\Sigma.strat}(e) \quad \sigma = \Sigma(n) [strat \mapsto \bar{p}] \end{array}}{n; \Sigma \vdash \langle \phi \mid \text{withStrategy } \bar{p} \, e \rangle \longrightarrow \langle \phi \mid e' \rangle : \sigma}$	$\frac{\text{E-RESET-STRATEGY} \quad \sigma = \Sigma(n) [strat \mapsto \bar{p}]}{n; \Sigma \vdash \langle \phi \mid \text{resetStrategy}_{\bar{p}}(v) \rangle \longrightarrow \langle \phi \mid v \rangle : \sigma}$
$\frac{\text{E-ASSUME} \quad \begin{array}{l} n; \Sigma \vdash \Sigma(n).lbl \sqsubseteq r : \ell_1 \quad n; \Sigma \vdash \Sigma(n).lbl \succcurlyeq \nabla(q) : \ell_2 \\ n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell_1 \sqcup \ell_2 \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n) [\Delta \mapsto (p, q) @ r :: \Sigma.\Delta], lbl \mapsto \Sigma(n).lbl \sqcup \ell_1 \sqcup \ell_2] \end{array}}{n; \Sigma \vdash \langle \phi \mid \text{assume } (p \succcurlyeq q) @ r \rangle \longrightarrow \langle \phi \mid \text{return } () \rangle : \sigma}$	

Figure 4.7: Monadic reductions for Flamio (cont).

```

1  (λ⊥Alice _ . [...])
2    let g = (λ⊥Bob f . do
3      bref := new Bob← true
4      withScope (do
5        assume Alice→ ≧ Bob→ @ Bob←
6        return (λAlice←Alice x . do
7          b <- !bref
8          bref := false
9          if b then f x else false)))
10     aliceCode
11  in g bobSecret) ()

```

Figure 4.8: Bob grants a function, supplied by Alice, authority to read Bob's confidential information once.

$$\begin{array}{c}
\text{G-STEP-LOCAL} \\
\frac{n; \Sigma \vdash S(n) \longrightarrow s : \sigma}{\langle n \cdot ns, \Sigma, S \rangle \Longrightarrow \langle n \cdot ns, \Sigma [n \mapsto \sigma], S [n \mapsto s] \rangle} \\
\\
\text{G-STEP-APP} \\
\frac{\begin{array}{l} \ell_n = \Sigma(n).lbl \quad \ell_m = \Sigma(m).lbl \quad S(m) = \langle \phi_m \mid es_m \rangle \\ m; \Sigma \vdash \ell_n \sqcup \ell_m \sqsubseteq m^{\rightarrow} \quad S(n) = \langle \phi_n \mid E[(\lambda_{s@p}^m x. e_n) e'_n] \rangle \\ s'_n = \langle \phi_n \mid E[\text{wait}(s)] \rangle \quad \sigma'_m = \Sigma(m) [lbl \mapsto \ell_n \sqcup \ell_m] \\ s'_m = \langle \phi_m \mid (\text{toLabeled}_{\ell_m} p (e_n[e'_n/x])); es_m \rangle \end{array}}{\langle n \cdot ns, \Sigma, S \rangle \Longrightarrow \langle m \cdot n \cdot ns, \Sigma [m \mapsto \sigma'_m], S [n \mapsto s'_n, m \mapsto s'_m] \rangle} \\
\\
\text{G-STEP-RET} \\
\frac{\begin{array}{l} S(n) = \langle \phi_n \mid E[\text{wait}(s)] \rangle \quad S(m) = \langle \phi_m \mid v; es_m \rangle \\ s'_n = \langle \phi_n \mid E[v] \rangle \quad s'_m = \langle \phi_m \mid es_m \rangle \end{array}}{\langle m \cdot n \cdot ns, \Sigma, S \rangle \Longrightarrow \langle n \cdot ns, \Sigma, S [n \mapsto s'_n, m \mapsto s'_m] \rangle}
\end{array}$$

Figure 4.9: Semantics of global steps

lifts a local reduction to a global reduction, and rules G-STEP-APP and G-STEP-RET handle remote procedure calls and returns respectively. When node n sends an RPC to node m , we call n the source node and m the target node. The global reduction rule is written as $\langle ns, \Sigma, S \rangle \Longrightarrow \langle ns', \Sigma', S' \rangle$ and can be read as “the first node in ns updates the environment Σ to Σ' , updates the local configurations S to S' , and modifies the call-stack to ns' ”. We write the reflexive, transitive closure of \Longrightarrow as \Longrightarrow^* .

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

Note that although computation is distributed, the semantics is deterministic: only one expression is reducible at any point. Determinism excludes internal timing leaks and other attacks usually found in concurrent systems [81, 114], while still allowing multiple nodes to share computation.

$$\begin{array}{c}
\text{ACTS-FOR-CONS-1} \quad \text{ACTS-FOR-CONS-2} \quad \text{ACTS-FOR-NIL} \\
\frac{\mathcal{H}; n; \Sigma \vdash_{s \sqcap n \rightarrow} p \succcurlyeq q : \ell}{\mathcal{H}; n; \Sigma \vdash_{s::ss} p \succcurlyeq q : \ell} \quad \frac{\begin{array}{c} \mathcal{H}; n; \Sigma \vdash_{s \sqcap n \rightarrow} p \succcurlyeq q : \text{fail} \\ \mathcal{H}; n; \Sigma \vdash_{ss} p \succcurlyeq q : \ell^? \end{array}}{\mathcal{H}; n; \Sigma \vdash_{s::ss} p \succcurlyeq q : \ell^?} \quad \frac{}{C \vdash_{\text{nil}} p \succcurlyeq q : \text{fail}} \\
\\
\text{ACTS-FOR} \\
\frac{ss = \Sigma(n).\text{strat} \quad \mathcal{H}; n; \Sigma \vdash_{ss} p \succcurlyeq q : \ell}{\mathcal{H}; n; \Sigma \vdash p \succcurlyeq q : \ell}
\end{array}$$

Figure 4.10: Top-level judgment for proving authorization queries in Flamio. The meta-variable C abbreviates $\mathcal{H}; n; \Sigma$.

4.4.3 Deriving Trust Relationship in Flamio

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

Rule ACTS-FOR in Figure 4.10 formalizes the top-level judgment for deriving trust, which iterates through the strategy principals in the current strategy and attempts to prove the trust relationship at each strategy principal in the list. We write $\ell^?$ to mean either a label ℓ , or a failure value fail meaning that the trust relationship could not be established. Rule ACTS-FOR-CONS-1 states that, if we can prove the trust relationship limiting our use of delegations to those bounded above by strategy principal s then the query succeeds. Rule ACTS-FOR-CONS-2 states that, if we cannot prove the trust relationship using delegations bounded above by strategy principal s ($C \vdash_{s \sqcap n \rightarrow} p \succcurlyeq q : \text{fail}$), then the search continues with the tail ss of the strategy. In both rules the strategy principal s is attenuated with the clearance of the node ($n \rightarrow$) to ensure that the node does not attempt to use delegations above its clearance label. Finally, in Rule ACTS-FOR-NIL, if we have tried all strategy principals and have not proved the trust relationship, when the proof search fails.

Figure 4.11 shows how the judgment $\mathcal{H}; n; \Sigma \vdash_s p \succcurlyeq q : \ell$ derives trust relationships between principals using the strategy principal s . The judgment means that node n proves that that q trusts p in the global environment Σ using delegations with labels that are upper bounded by ℓ , and assuming that r acts for s for all $(r, s) \in \mathcal{H}$. We call \mathcal{H} the assumptions of the query, and write $r \succcurlyeq s$ for the assumption (r, s) . We write

$$n; \Sigma \vdash_s p \succcurlyeq q : \ell$$

$$\begin{array}{c}
\text{BOT} \quad \frac{}{C \vdash_s p \succ \perp : \perp^\Xi} \quad \text{TOP} \quad \frac{}{C \vdash_s \top \succ p : \perp^\Xi} \quad \text{REFL} \quad \frac{}{C \vdash_s p \succ p : \perp^\Xi} \quad \text{ASSUMP} \quad \frac{(p \succ q) \in \mathcal{H}}{C \vdash_s p \succ q : \perp^\Xi} \\[10pt]
\text{PROJ} \quad \frac{C \vdash_s p \succ q : \ell}{C \vdash_s p^\pi \succ q^\pi : \ell} \quad \text{PROJR} \quad \frac{}{C \vdash_s p \succ p^\pi : \perp^\Xi} \quad \text{OWN-1} \quad \frac{C \vdash_s o \succ o' : \ell_1 \quad C \vdash_s p \succ p' : \ell_2}{C \vdash_s o:p \succ o':p' : \ell_1 \sqcup \ell_2} \\[10pt]
\text{OWN-2} \quad \frac{C \vdash_s o \succ o' : \ell_1 \quad C \vdash_s p \succ o':p' : \ell_2}{C \vdash_s o:p \succ o':p' : \ell_1 \sqcup \ell_2} \quad \text{CONJ-L} \quad \frac{j \in \{1,2\} \quad C \vdash_s p_j \succ p : \ell}{C \vdash_s p_1 \wedge p_2 \succ p : \ell} \\[10pt]
\text{CONJ-R} \quad \frac{C \vdash_s p \succ p_1 : \ell_1 \quad C \vdash_s p \succ p_2 : \ell_2}{C \vdash_s p \succ p_1 \wedge p_2 : \ell_1 \sqcup \ell_2} \quad \text{DISJ-L} \quad \frac{C \vdash_s p_1 \succ p : \ell_1 \quad C \vdash_s p_2 \succ p : \ell_2}{C \vdash_s p_1 \vee p_2 \succ p : \ell_1 \sqcup \ell_2} \\[10pt]
\text{DISJ-R} \quad \frac{j \in \{1,2\} \quad C \vdash_s p \succ p_j : \ell}{C \vdash_s p \succ p_1 \vee p_2 : \ell} \quad \text{TRANS} \quad \frac{C \vdash_s p \succ q : \ell_1 \quad C \vdash_s q \succ r : \ell_2}{C \vdash_s p \succ r : \ell_1 \sqcup \ell_2} \quad \text{DEL} \quad \frac{p \succ q @ \ell \in \Sigma(n). \Delta \quad \mathcal{H}, p \succ q; n; \Sigma \vdash_s \ell \sqsubseteq s : \ell' \quad \mathcal{H}, p \succ q; n; \Sigma \vdash_s \ell' \sqsubseteq s : \perp^\Xi}{\mathcal{H}; n; \Sigma \vdash_s p \succ q : \ell} \\[10pt]
\text{FWD} \quad \frac{\ell_n = \Sigma(n). \text{lbl} \quad \sigma_m = \Sigma(m) [\text{lbl} \mapsto \ell_n \sqcup \ell_m] \quad \ell_m = \Sigma(m). \text{lbl} \quad \mathcal{H}; n; \Sigma \vdash_s \ell_n \sqcup \ell_m \sqsubseteq m^\rightarrow : \ell_1 \quad \mathcal{H}; m; \Sigma [m \mapsto \sigma_m] \vdash_s p \succ q : \ell_2}{\mathcal{H}; n; \Sigma \vdash_s p \succ q : \ell_1 \sqcup \ell_2}
\end{array}$$

Figure 4.11: Acts for judgment of Flamio. The meta-variable C abbreviates $\mathcal{H}; n; \Sigma$.

to mean $\emptyset; n; \Sigma \vdash_s p \succcurlyeq q : \ell$. That is, the judgment holds with no assumptions. We also write $\mathcal{H}; n; \Sigma \vdash_s p \succcurlyeq q$ to mean $\exists \ell . \mathcal{H}; n; \Sigma \vdash_s p \succcurlyeq q : \ell$. Finally, guided by (4.1) we write

$$\mathcal{H}; n; \Sigma \vdash_s p \sqsubseteq q : \ell$$

to mean $\mathcal{H}; n; \Sigma \vdash_s q^{\rightarrow} \wedge p^{\leftarrow} \succcurlyeq p^{\rightarrow} \wedge q^{\leftarrow} : \ell$ and use similar abbreviations as above.

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

Rule ASSUMP states that any assumptions can be used to derive trust without raising the current label any further. This use of assumptions is an instance of a checked endorsement [11, 29], and is discussed later in this section. Rules OWN-1 and OWN-2 derives trust between ownership projections. First, OWN-1 shows that trust between principals imply trust between owned principals, and OWN-2 states that, if an ownership projection $o':p'$ trusts a principal p and owner o' trusts o then another ownership principal $o:p$ also trusts $o':p'$. Rule FWD expresses how a node n can query another node m for a trust relationship, but only if n allows the information that caused n to contact m to be learned by m (i.e., $\ell_n \sqsubseteq m^{\rightarrow}$, which is implied by the premise $\ell_n \sqcup \ell_m \sqsubseteq m^{\rightarrow}$). Furthermore, when forwarding a query, node m must raise its current label to $\ell_n \sqcup \ell_m$ to propagate the sensitivity of the computational context of node n . Finally, when node n forwards the query to m , the strategy principal used by n is also used by m as otherwise, local reasoning about trust relationship queries would be impossible without knowing the strategies of every node in the system.

Finally, rule DEL expresses how delegations are used to derive trust. First, a delegation that proves the trust relationship must be present ($p \succcurlyeq q @ \ell \in \Sigma(n).\Delta$), then, the label on the delegation must flow to the strategy principal s that is currently bounding how much the current label can be raised ($\mathcal{H}, p \succcurlyeq q; n; \Sigma \vdash \ell \sqsubseteq s : \ell'$). However, the fact that ℓ is bounded by s could also be used to leak information, so ℓ' should also be bounded by s . This checking could potentially continue ad infinitum, so we apply a pragmatic approach and require this check not to use any delegations labeled higher

⁴That the upper bound on the delegation labels is an “input” to the judgment can be seen in the $F\lambda$ calculus [10] where delegation labels appear in the surface syntax of expressions.

than \perp^ε . Section 4.6 motivates this decision, which shows several interesting examples that can all be implemented using this simplified checking mechanism.

When checking if a delegation can be used in the rule DEL, the assumptions \mathcal{H} is extended to include the trust relationship that is being checked. This usage of assumptions is a form of checked endorsement which was also noted to be a useful extension to the Jif programming language [29]. To see the effect of this style of reasoning, consider the query $n; \Sigma \vdash p \sqsubseteq q : \ell$ where $\Sigma(n) = (n^\leftarrow, \Delta, [q])$ and $\Delta = \{(q^\rightarrow \wedge p^\leftarrow \succcurlyeq p^\rightarrow \wedge q^\leftarrow) @ p\}$. This query is equivalent to $n; \Sigma \vdash q^\rightarrow \wedge p^\leftarrow \succcurlyeq p^\rightarrow \wedge q^\leftarrow : \ell$, so applying DEL the goal reduces to proving $p \sqsubseteq q; n; \Sigma \vdash p \sqsubseteq q : \ell'$ for some ℓ' , which (ignoring the assumption) is the exact same query we started out with! However, we now have the assumption $p \sqsubseteq q$, and the goal follows by applying ASSUMP. If no assumption was added when checking that the label on the delegation $p \sqsubseteq q @ p$ flows to the strategy principal q , this trust relationship could not be proven in any finite derivation. While this situation might appear artificial, Section 4.6 presents a use case where this problem arises naturally.

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

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

$$\Delta_1 = \{a \succcurlyeq b @ \perp^\varepsilon\}$$

$$\Delta_2 = \{a \succcurlyeq b @ c, c \sqsubseteq \ell @ \perp^\varepsilon\}$$

$$\Delta_3 = \{a \succcurlyeq b @ c, c \sqsubseteq \ell @ d, d \sqsubseteq \ell @ \perp^\varepsilon\}$$

$$\Delta_4 = \{a \succcurlyeq b @ c, c \sqsubseteq \ell @ d, d \sqsubseteq \ell @ e, e \sqsubseteq \ell @ \perp^\varepsilon\}$$

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

$$a \succcurlyeq b @ c \in \Delta_4 \tag{4.4}$$

$$\{a \succcurlyeq b\}; n; \Sigma_4 \vdash_\ell c \sqsubseteq \ell : d \tag{4.5}$$

$$\{a \succcurlyeq b\}; n; \Sigma_4 \vdash_\ell d \sqsubseteq \ell : \perp^\varepsilon \tag{4.6}$$

Condition (4.4) holds by definition of Δ_4 , and (4.5) holds by applying DEL. But (4.6) does not hold: we can only show $\{a \succcurlyeq b\}; n; \Sigma_4 \vdash d \sqsubseteq \ell : e$ and $\{a \succcurlyeq b\}; n; \Sigma_4 \vdash e \sqsubseteq \ell : \perp^\varepsilon$, but this does not imply (4.6). That is, Flamio cannot prove that the label on the information “the label on the required

delegation flows to the current strategy principal” is $\perp_{\mathbb{E}}$. We have not found a realistic scenario where this presents a problem, and we leave lifting this restriction as future work.

4.4.4 A Type System for Flamio.

Since Flamio controls information-flows via dynamic checks, the type system for Flamio is straightforward. We write $n; \Gamma \vdash e : s$ when expression e can be given type s in a global type environment $\Gamma : \mathcal{N} \rightarrow (\text{Var} \uplus \text{Loc} \rightarrow s)$ on node n . Note that the typing environment maps both variables and locations to types. Figure 4.12 shows excerpts of this judgment. Rule T-ABS states that a function has a function type and that, the typing environment for node m is used when checking the type of the body, where m is the target node. Rule T-LAB states that labeled expressions have labeled types. Rules T-RETURN and T-BIND are standard typing rules for monadic expressions. Rule T-TO-LABELED states that an expression `toLabeled` e_1 e_2 is well-typed when e_1 is a principal, and that the expression has type `Labeled` s if e_2 has type s . We say a location a belongs to node n if $a \in \text{dom}(\Gamma_n)$. Rule T-NEW states that when a reference is allocated on a node n the type of the location returned belongs to n , and T-READ states that a reference can only be read on a node to which the location belongs. Finally, T-WAIT states that the type attached to a waiting expression is the type of the expression.

Given a global typing environment Γ we write $n; \Gamma \vdash \phi$ if, for all a such that $\phi(a) = e @ p$ and $\Gamma_n(a) = \text{Ref}_n s$ it holds that $n; \Gamma \vdash e : s$. We write $n; \Gamma \vdash \langle \phi \mid es \rangle : \tau s$ if $es = e_1 \cdots e_n$ and $n; \Gamma \vdash \phi$ and $n; \Gamma \vdash e_i : s_i$ for $i = 1, \dots, n$ and $\tau s = s_1 \dots s_n$. We lift this definition to global configurations and write $\Gamma \vdash_m \langle n, \Sigma, S \rangle : \tau s$ if for all $n' \in \mathcal{N}$ there exists a type $\tau s'$ such that $n'; \Gamma \vdash S_{n'} : \tau s'$, and furthermore, when $n' = m$ we have $\tau s' = \tau s$.

4.5 Security Guarantees

In this section, we define the attacker model and show the security guarantees given by Flamio. Specifically, we show that Flamio executions satisfy termination-insensitive noninterference (TINI) [12]. Formally, an attacker is some principal \mathcal{A} . Note that this principal might be a conjunction of named principals $n_1 \wedge \cdots \wedge n_k$ representing a set of k colluding principals. In the sections that follow, we denote a \mathcal{N} -indexed set of memories as a function $\Phi : \mathcal{N} \rightarrow (\text{Loc} \rightarrow \nu)$.

$\frac{}{n; \Gamma \vdash p : \text{Principal}}$	$\frac{\Gamma_n(x) = s}{n; \Gamma \vdash x : s}$	$\frac{\Gamma_n(a) = \text{Ref}_n s}{n; \Gamma \vdash a : \text{Ref}_n s}$
$\frac{\begin{array}{l} \text{T-ABS} \\ n; \Gamma, x : s_1 \vdash e : \text{LIO } s_2 \\ s = s_1 \rightarrow \text{LIO (Labeled } s_2) \end{array}}{n; \Gamma \vdash \lambda_{s_2 @ p}^m x. e : s}$	$\frac{\begin{array}{l} \text{T-LAB} \\ n; \Gamma \vdash e_2 : s \\ n; \Gamma \vdash e_1 : \text{Principal} \end{array}}{n; \Gamma \vdash e_2 @ e_1 : \text{Labeled } s}$	
$\frac{\begin{array}{l} \text{T-RETURN} \\ n; \Gamma \vdash e : s \end{array}}{n; \Gamma \vdash \text{return } e : \text{LIO } s}$	$\frac{\begin{array}{l} \text{T-BIND} \\ n; \Gamma \vdash e_1 : \text{LIO } s_1 \\ n; \Gamma \vdash e_2 : s_1 \rightarrow \text{LIO } s_2 \end{array}}{n; \Gamma \vdash e_1 \gg e_2 : \text{LIO } s_2}$	
$\frac{\begin{array}{l} \text{T-TO-LABELED} \\ n; \Gamma \vdash e_1 : \text{Principal} \\ n; \Gamma \vdash e_2 : \text{LIO } s \\ s' = \text{LIO (Labeled } s) \end{array}}{n; \Gamma \vdash \text{toLabeled } e_1 \ e_2 : s'}$	$\frac{\begin{array}{l} \text{T-READ} \\ n; \Gamma \vdash e : \text{Ref}_n s \end{array}}{n; \Gamma \vdash !e : \text{LIO } s}$	$\frac{\begin{array}{l} \text{T-NEW} \\ n; \Gamma \vdash e_1 : \text{Principal} \\ n; \Gamma \vdash e_2 : s \\ s' = \text{LIO (Ref}_n s) \end{array}}{n; \Gamma \vdash \text{new } e_1 \ e_2 : s'}$
$\frac{\begin{array}{l} \text{T-WAIT} \\ s' = \text{LIO (Labeled } s) \end{array}}{n; \Gamma \vdash \text{wait}(s) : s'}$		

Figure 4.12: Typing judgment for Flamio.

4.5.1 Trace Semantics

We express the attacker model in terms of a trace semantics, in which certain operations in the language emit *events* which may or may not be observable by \mathcal{A} . The grammar for events is given in Figure 4.13. A non-empty event (α, Σ, n) contains the type of the event α , the current environment when the event was emitted Σ , and the node n that emitted the event. The types of events include: write events $\text{write}(a, e)$, emitted when a node writes an expression e to reference a ; allocation events $\text{new}(a, e)$, emitted when a node allocates a new reference a , initialized to e ; call events $\text{call}(e, m)$, emitted when a node invokes an RPC e on node m ; and return events $\text{ret}(v, m)$, emitted when a node finishes an RPC, returning value v to the caller node m . In addition, we have release events release , explained below. Finally, the empty event ε is emitted by operations that do not emit some other event. We call a sequence of events a *trace*. We write the concatenation of traces as $t_1 \cdot t_2$ and we write the empty trace as ε .

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

Figure 4.13: The syntax of events.

$$\begin{array}{c}
\text{E-WRITE-EV} \\
\frac{[\dots] \quad ev = (\text{write}(a, e'), \Sigma, n)}{n; \Sigma \vdash \langle \phi \mid a := e' \rangle \xrightarrow{ev} \langle \phi' \mid \text{return } () \rangle : \sigma} \\
\\
\text{G-STEP-RET-EV} \\
\frac{S(n) = \langle \phi_n \mid E[\text{wait}(s)] \rangle \quad S(m) = \langle \phi_m \mid v; es_m \rangle \quad ev = (\text{ret}(v, n), \Sigma, m)}{\langle m \cdot n \cdot ns, \Sigma, S \rangle \xrightarrow{ev} \langle n \cdot ns, \Sigma, S[n \mapsto s'_n, m \mapsto s'_m] \rangle} \\
\\
\text{G-STEP-LOCAL-EV} \\
\frac{n; \Sigma \vdash S(n) \xrightarrow{ev} s : \sigma}{\langle n \cdot ns, \Sigma, S \rangle \xrightarrow{ev} \langle n \cdot ns, \Sigma[n \mapsto \sigma], S[n \mapsto s] \rangle} \\
\\
\text{E-ASSUME-EV} \\
\frac{[\dots] \quad ev = (\text{release}(p, q, r), \Sigma, n)}{n; \Sigma \vdash \langle \phi \mid \text{assume } (p \succcurlyeq q) @ r \rangle \xrightarrow{ev} \langle \phi \mid \text{return } () \rangle : \sigma}
\end{array}$$

Figure 4.14: Augmented semantics emitting events.

Release events Flamio is a very expressive language that permits downgrading, i.e., intentionally relaxing information-flow restrictions on data [83]. To define noninterference, we are concerned only with executions that do not downgrade information to \mathcal{A} (or, more precisely, from p to q where $p \not\sqsubseteq \mathcal{A}$ and $q \sqsubseteq \mathcal{A}$). We expect generalizations of noninterference, like robust declassification [85, 137] and nonmalleable information-flow [28] to hold for Flamio, but in this work we consider only the case where no node downgrades information to \mathcal{A} . To capture this intuition, we introduce the notion of a *bad* release event. Intuitively, when no bad release events are emitted nothing is being downgraded to \mathcal{A} .

Bad release events We call downgrading of confidentiality labels *declassification*, and downgrading of integrity labels *endorsement*. As Flamio permits both declassification and endorsement using delegations, a bad event should capture both cases. We call a release event $ev = (\text{release}(p, q, r), \Sigma, n)$ bad, written $\mathcal{A} \vdash \text{bad}(ev)$, if $n; \Sigma \vdash r \sqsubseteq \mathcal{A}$ and one of the following conditions hold:

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

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

Examples of bad release events The release event

$$ev = (\text{release}(\mathcal{A}, \text{Alice}^\rightarrow, \perp^\mathbb{E}), \Sigma, n)$$

generated by the local step $n; \Sigma \vdash \langle \phi \mid \text{assume } (\mathcal{A} \succcurlyeq \text{Alice}^\rightarrow) @ \perp^\mathbb{E} \rangle \xrightarrow{ev} \langle \phi \mid e \rangle : \sigma$ (i.e., a declassification from Alice^\rightarrow to \mathcal{A}) is bad since $n; \Sigma \vdash \perp^\mathbb{E} \sqsubseteq \mathcal{A}$ (i.e., \mathcal{A} can observe the delegation), $n; \Sigma \vdash \mathcal{A} \sqsubseteq \mathcal{A}^\rightarrow$ (i.e., it is a declassification to a principal that \mathcal{A} can observe), and $n; \Sigma \vdash \text{Alice}^\rightarrow \not\sqsubseteq \mathcal{A}^\rightarrow$ (i.e., it is a declassification from a principal that \mathcal{A} previously could not observe).

The release event $ev = (\text{release}(\mathcal{A}, \text{Bob}^\leftarrow, \perp^\mathbb{E}), \Sigma, n)$ generated by the local step $n; \Sigma \vdash \langle \phi \mid \text{assume } (\mathcal{A} \succcurlyeq \text{Bob}^\leftarrow) @ \perp^\mathbb{E} \rangle \xrightarrow{ev} \langle \phi \mid () \rangle : \sigma$ (i.e., an endorsement from \mathcal{A} to Bob^\leftarrow) is bad since $n; \Sigma \vdash \perp^\mathbb{E} \sqsubseteq \mathcal{A}$ (i.e., \mathcal{A} can observe the delegation), $n; \Sigma \vdash \mathcal{A}^\leftarrow \sqsubseteq \mathcal{A}$ (i.e., it is an endorsement from a principal that \mathcal{A} can modify), and $n; \Sigma \vdash \mathcal{A}^\leftarrow \not\sqsubseteq \text{Bob}^\leftarrow$ (i.e., it is an endorsement to a principal that \mathcal{A} previously could not modify).

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

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

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

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

We augment the semantics from Section 4.4 with events. Figure 4.14 shows an excerpt of the augmented semantics (we use [...] to elide the premises presented in Section 4.4). Except for the emitted event these rules correspond exactly to the rules in Figures 4.6, 4.7 and 4.9. We write $\langle ns, \Sigma, S \rangle \xRightarrow{t}^*$ when there exists a configuration $\langle ns', \Sigma', S' \rangle$ such that $\langle ns, \Sigma, S \rangle \xRightarrow{t}^* \langle ns', \Sigma', S' \rangle$ and $S'(n) = \langle \phi_n \mid \bullet \rangle$ for all n . We also write $\langle ns, \Sigma, S \rangle \xRightarrow{\mathcal{A} \rightsquigarrow t'}^*$ when $\langle ns, \Sigma, S \rangle \xRightarrow{t}^*$ and $t' = t \upharpoonright \mathcal{A}$.

We define an \mathcal{A} -equivalence relation that makes explicit which traces and memories an attacker \mathcal{A} can distinguish. As they both contain expressions, we define an \mathcal{A} -equivalence on expressions, and Figure 4.15 shows an excerpt of this judgment. Intuitively, two expressions e_1 and e_2 are considered \mathcal{A} -equivalent if the current label on each context does not flow to \mathcal{A} , or if the label on each context flows to \mathcal{A} and e_1 and e_2 are equal “up to labeled values with a label that does not flow to \mathcal{A} ”. Figure 4.15 formalizes this intuition: rule EQ-HIGH states that two expressions are \mathcal{A} -equivalent in environments where the current label does not flow to \mathcal{A} , and the remaining rules state that two expressions are \mathcal{A} -equivalent if the current label of each environment flows to \mathcal{A} , and the expressions are equal up to labeled values that \mathcal{A} cannot observe.

For most cases $C \vdash e_1 \simeq_{\mathcal{A}}^{\theta} e_2$ recursively inspects the subexpressions of each expression, but a few cases need special care: to relate dynamically allocated locations we use a partial bijection [17, 104] $\theta : \text{Loc} \rightarrow \text{Loc}$ in EQ-ADDR. The most important rules are EQ-LABELED-1 (that makes explicit the notion that an attacker cannot distinguish two terms labeled with a principal which does not flow to \mathcal{A}), and EQ-LABELED-2 (that states that \mathcal{A} can “look inside” terms labeled with principals that flow to \mathcal{A}). When θ is not important we write $n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq_{\mathcal{A}} e_2$ to mean $n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq_{\mathcal{A}}^{\theta} e_2$ for some bijection θ .

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

Finally, Figure 4.16 shows \mathcal{A} -equivalence on memories. Rule STORE-EQ-EMPTY states that two empty memories are \mathcal{A} -equivalent, and rule STORE-EQ-LOW states that extending two memories with \mathcal{A} -equivalent expressions preserves \mathcal{A} -equivalence. Lastly, rules STORE-EQ-HIGH-1 and STORE-EQ-HIGH-2 states that both memories can be extended with terms labeled with a principal that does not flow to \mathcal{A} without the attacker being able to distinguish the memories. We extend the notion of \mathcal{A} -equivalence on memo-

$$\begin{array}{c}
\text{EQ-HIGH} \\
\frac{i = 1, 2 \quad n; \Sigma_i \vdash \Sigma_i(n).lbl \not\sqsubseteq \mathcal{A}}{n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq_{\mathcal{A}}^{\theta} e_2} \\
\\
\text{EQ-ADDR} \\
\frac{\theta(a_1) = a_2 \quad n; \Sigma_i \vdash \Sigma_i(n).lbl \sqsubseteq \mathcal{A} \quad i = 1, 2}{n; \Sigma_1; \Sigma_2 \vdash a_1 \simeq_{\mathcal{A}}^{\theta} a_2} \\
\\
\text{EQ-LABELED-1} \\
\frac{n; \Sigma_i \vdash p_i \not\sqsubseteq \mathcal{A} \quad n; \Sigma_i \vdash \Sigma_i(n).lbl \sqsubseteq \mathcal{A} \quad i = 1, 2}{n; \Sigma_1; \Sigma_2 \vdash e_1 @ p_1 \simeq_{\mathcal{A}}^{\theta} e_2 @ p_2} \\
\\
\text{EQ-LABELED-2} \\
\frac{n; \Sigma_i \vdash q \sqsubseteq \mathcal{A} \quad n; \Sigma_i \vdash \Sigma_i(n).lbl \sqsubseteq \mathcal{A} \quad i = 1, 2 \quad n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq_{\mathcal{A}}^{\theta} e_2}{n; \Sigma_1; \Sigma_2 \vdash e_1 @ q \simeq_{\mathcal{A}}^{\theta} e_2 @ q}
\end{array}$$

Figure 4.15: \mathcal{A} -equivalence for terms and expressions.

$$\begin{array}{c}
\text{STORE-EQ-EMPTY} \\
\frac{}{C \vdash \emptyset \simeq_{\mathcal{A}}^{\theta} \emptyset} \\
\\
\text{STORE-EQ-HIGH-1} \\
\frac{C_1 \vdash q \not\sqsubseteq \mathcal{A} \quad C \vdash \phi_1 \simeq_{\mathcal{A}}^{\theta} \phi_2}{C \vdash \phi_1 [a \mapsto e @ q] \simeq_{\mathcal{A}}^{\theta} \phi_2} \\
\\
\text{STORE-EQ-HIGH-2} \\
\frac{C_2 \vdash q \not\sqsubseteq \mathcal{A} \quad C \vdash \phi_1 \simeq_{\mathcal{A}}^{\theta} \phi_2}{C \vdash \phi_1 \simeq_{\mathcal{A}}^{\theta} \phi_2 [a \mapsto e @ q]} \\
\\
\text{STORE-EQ-LOW} \\
\frac{\theta(a_1) = a_2 \quad C \vdash e_1 \simeq_{\mathcal{A}}^{\theta} e_2 \quad C_i \vdash q \sqsubseteq \mathcal{A} \quad i = 1, 2 \quad C \vdash \phi_1 \simeq_{\mathcal{A}}^{\theta} \phi_2 \quad \phi'_i = \phi_i [a_i \mapsto (e_i @ q)]}{C \vdash \phi'_1 \simeq_{\mathcal{A}}^{\theta} \phi'_2}
\end{array}$$

Figure 4.16: \mathcal{A} -equivalence for memories. The meta-variable C abbreviates $n; \Sigma_1; \Sigma_2$, and $C_i = n; \Sigma_i$.

ries to \mathbb{N} -indexed sets of memories and write $m; \Sigma_1; \Sigma_2 \vdash \Phi \simeq_{\mathcal{A}}^{\theta} \Psi$ when $\forall n \in \mathbb{N}. m; \Sigma_1; \Sigma_2 \vdash \Phi(n) \simeq_{\mathcal{A}}^{\theta} \Psi(n)$.

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

Attacker knowledge We present our noninterference result using the notion of attacker knowledge [12, 13]. The attacker knowledge is the set of

initial memories that could lead to a given observable trace, and a larger knowledge set corresponds to more uncertainty about the initial memory. Formally, attacker knowledge, given a trace t produced by expression e , is the set $k_{\mathcal{A}}^n(e, t)$.

$$k_{\mathcal{A}}^n(e, t) = \left\{ \Psi \mid \langle n, \emptyset, S \rangle \xRightarrow{\mathcal{A} \rightsquigarrow t'}^* \wedge t \simeq_{\mathcal{A}} t' \right\}$$

where $S(m) = \langle \Psi(n) \mid \llbracket e \rrbracket_n(m) \rangle$ and

$$\llbracket e \rrbracket_n(m) = \begin{cases} e & \text{if } n = m \\ \bullet & \text{otherwise} \end{cases}$$

ensures that expression e is initially evaluated on node n , and the remaining nodes start with an empty list of expressions. As is standard for termination-insensitive noninterference (TINI), the policy [46] is defined as all \mathcal{A} -equivalent terminating memories, which we denote by $k_{\mathcal{A}}^{\downarrow n}(\Phi, e)$.

$$k_{\mathcal{A}}^{\downarrow n}(\Phi, e) = \{ \Psi \mid \Phi \simeq_{\mathcal{A}} \Psi \wedge \langle n, \emptyset, S \rangle \xRightarrow{*} \}$$

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

Theorem 4.5.1 (Noninterference). *Let e satisfy $n; \Gamma \vdash e : s$. If $\langle n, \emptyset, S \rangle \xRightarrow{\mathcal{A} \rightsquigarrow t'}^*$ for $S(m) = \langle \Phi(m) \mid \llbracket e \rrbracket_n(m) \rangle$ such that $\mathcal{A} \vdash \text{good}(t)$ then $k_{\mathcal{A}}^n(e, t) \supseteq k_{\mathcal{A}}^{\downarrow n}(\Phi, e)$.*

The technical report [94] contains a complete proof of Theorem 4.5.1. The theorem shows that we have securely integrated FLAM into an LIO-like setting with a floating label, where proofs of trust relationships do not inappropriately reveal confidential information, nor are they inappropriately affected by untrusted information.

4.6 Implementation and Case Studies

We have implemented Flamio as a monadic library in Haskell [95]. The code is approximately 2,100 lines of code in total, and uses FLAM's efficient query resolution algorithm for authorization queries [9]. Proof search for trust relationships is implemented as computations in the Flamio monad, ensuring that delegations are not used inappropriately. The case studies

demonstrate how application-specific search strategies are used to prevent label creep during proof search.

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

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

First, 1) does not lead to loss of expressivity: as shown by Cooper and Wadler [32], a program in a calculus similar to that of Section 4.4 can be translated by performing defunctionalization to a Haskell program (which can then use the Flamio implementation).

Second, 2) significantly reduces network communication but means that the implementation is unsound if the trust relationship between principals changes during query resolution. Orthogonal work on query isolation [67] can provide transactional behavior for distributed systems like Flamio.

Using this implementation, we have constructed three use cases for Flamio consisting of roughly 500 lines of code. The first use case is a distributed bank, which was already presented in Section 4.3. The banking example shows how users can perform remote procedure calls to handle transactions across accounts between different users. A user u can authorize user u' to transfer money on behalf of u by adding a delegation $u' \succcurlyeq u @ \text{bank}^{\rightarrow} \wedge u^{\leftarrow}$ locally on u 's node. This delegation is read as “ u trusts u' , and this information is confidential to bank and has the integrity of u ”. When bank wishes to prove the trust relationship between u and u' to authorize a transfer of money, a proof search is issued, and using the FWD rule, is forwarded to node u . Node u then proves the trust relationship using the local delegation $u' \succcurlyeq u @ \text{bank}^{\rightarrow} \wedge u^{\leftarrow}$.

In addition to the banking example, we construct a secure social jukebox service where people schedule music during social gatherings. The third use

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

case is a secure database containing confidential information about government agencies. The third example also demonstrates how the *ASSUMP* rule prevents infinite derivation trees in authorization queries, and how such queries can show up in practical use cases.

4.6.1 Secure Social Jukebox Service

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

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

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

Function *get* returns a pair (s, lss) containing the current song being played s , and a list of labeled strings lss such that $s' @ p \in lss$ represents that p voted for s' . So if Alice wants to vote for “Taylor Swift - Shake It Off”, she appends a labeled string “Taylor Swift - Shake It Off”@Alice using *put*. By labeling her vote with the principal Alice, she knows that only principals p such that $\text{Alice}^{\rightarrow}$ flows to p^{\rightarrow} can learn her vote (i.e., by *E-UNLABEL* it must be the case that Alice flows to p^{\rightarrow}). Furthermore, since the integrity of the label on the vote is $\text{Alice}^{\leftarrow}$, she knows that any vote of the form $s @ \text{Alice}$ for some song s must be placed by a principal p such that $\text{Alice}^{\leftarrow}$ trusts p^{\leftarrow} (i.e., by *E-LABEL* it must be the case that p ’s current label flows to Alice).

Figure 4.17 shows an example of Alice voting for “Shake it Off” by Taylor Swift. First, Alice adds a delegation that allows J to read Alice’s labeled vote. She then labels her vote and inserts it into the list of labeled songs. When J then unlabels the labeled song title, a proof search will be issued checking $J^{\leftarrow} \sqcup \text{Alice}$ flows to J^{\rightarrow} , which is equivalent to checking that $\text{Alice}^{\rightarrow}$ trusts $J^{\rightarrow} \wedge (J \vee \text{Alice})^{\leftarrow}$. This trust relationship holds by the delegation that Alice placed in Figure 4.17.

4.6.2 Government Agency Records

As a final example demonstrating the usefulness of Flamio, we show how confidential delegations can be used to keep government agency records.

```

1  ( $\lambda_{\text{Alice}^{\leftarrow}}^{\text{Alice}^{\rightarrow}} \_ \_ . \text{do}$ 
2    assume  $J^{\rightarrow} \succcurlyeq \text{Alice}^{\rightarrow} @ (\perp^{\rightarrow} \wedge \text{Alice}^{\leftarrow})$ 
3     $\text{ls} \leftarrow \text{label Alice "Taylor Swift - Shake It Off"}$ 
4     $(\text{curSong}, \text{votedSongs}) \leftarrow \text{get}$ 
5     $\text{put} (\text{curSong}, \text{ls} :: \text{votedSongs})$ ) ()

```

Figure 4.17: Alice places a secret vote for Taylor Swift.

Suppose the CIA hires a subset of \mathcal{N} as agents. This information should be visible only to other CIA agents, not to the general public. Furthermore, only the CIA should be able to hire agents.

We implement an enforcement mechanism for this security policy as follows: when $\text{CIA} \in \mathcal{N}$ hires an agent $n \in \mathcal{N}$, two delegations are created: first, trust is delegated from CIA to $\text{CIA}:n$. By the properties of ownership projections [9], this delegation can be read as “CIA trusts n , but CIA controls which principals can act for n ”.⁶ To satisfy the security policy, this delegation should be visible only to other agents, and CIA should be able to trust that any such delegation could have been placed only by the CIA. This directly translates into the label $(\text{CIA}:\text{AgentDB})^{\rightarrow} \wedge \text{CIA}^{\leftarrow}$ on the delegation.

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

Figure 4.18 shows how Alice can verify that Bob is also a secret agent. On line 4, Alice and Bob are both hired as secret agents using the function `hire` defined on lines 1-3.

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

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

⁶The use of ownership principals prevent *delegation loopholes* [9].

```

1  let hire =  $\lambda_{CIA}^{CIA} n . \mathbf{do}$ 
2    assume  $CIA:n \succcurlyeq CIA @ (CIA:AgentDB)^{\rightarrow} \wedge CIA^{\leftarrow}$ 
3    assume  $n^{\rightarrow} \succcurlyeq (CIA:AgentDB)^{\rightarrow} @ (CIA:AgentDB)^{\rightarrow} \wedge CIA^{\leftarrow}$ 
4    in ( $\lambda_{CIA}^{CIA} - . \mathbf{do}$  hire Alice; hire Bob) ()
5    [...]
6    ( $\lambda_{Alice}^{Alice} - . \mathbf{withStrategy}$  [Alice $^{\rightarrow} \wedge CIA^{\leftarrow}$ ] (do
7      isAgent <- CIA:Bob  $\succcurlyeq CIA$ 
8      if isAgent then [...] else [...])) ()

```

Figure 4.18: Alice verifies that Bob is a secret agent for the CIA.

trusts $Alice^{\rightarrow}$ under the assumption that $(CIA:AgentDB)^{\rightarrow}$ trusts $Alice^{\rightarrow}$, and so the trust relationship is established using **ASSUMP**.

Notice that without the **ASSUMP** rule, we would not be able to prove that $(CIA:AgentDB)^{\rightarrow}$ trusts $Alice^{\rightarrow}$ (or, equivalently, that $CIA:AgentDB$ can learn information labeled as Alice): we would keep applying **DEL** without a way of terminating the derivation. But, intuitively, this relationship should hold as we have a delegation of trust from $(CIA:AgentDB)^{\rightarrow}$ to $Alice^{\rightarrow}$. Rule **ASSUMP** allows the proof search to assume the delegation when proving that it is secure to use the delegation, effectively expressing that “it is secure to use the delegation because the delegation says so”: a form of checked endorsement [11, 29].

4.7 Related Work

4.7.1 FLAM

The FLAM technical report [10] presents a security-typed language $F\lambda$ in which policies are FLAM principals. Much like Flamio, $F\lambda$ can delegate trust during evaluation and allows querying of trust relationships. However, the decision of whether or not to allow downgrading (i.e., adding new trust relationships) must be performed statically using a relatively simple type system. By contrast, in Flamio all decisions about whether to allow downgrading are done during evaluation, meaning that Flamio can potentially allow more downgrading and remain secure. As $F\lambda$ is a language with fine-grained IFC the programmer is also burdened with more label annotations than would be expected in similar Flamio programs.

FLAC [8] is a calculus for flow-limited authorization that allows static reasoning about mechanisms such as commitment schemes or bearer credentials that require dynamic authorization. FLAC builds a sophisticated type system on top of FLAM that provides noninterference and robust declassifi-

cation guarantees. Although FLAC offers many high-level features to build practical authorization mechanisms, it uses a limited subset of FLAM, e.g., it does not have distributed trust checking (corresponding to FLAM’s Fwd rule).

Hyperflow [43] is a new processor architecture for nonmalleable timing-sensitive IFC that uses FLAM principals encoded as bit vectors as the label model. This encoding offers efficient computation of joins, meets, and projections. Hyperflow extends the RISC-V processor with IFC instructions and limits how information flows through registers and memory pages. The hardware is programmed in a new hardware-description language, ChiselFlow, embedded in Scala. Much like Flamio, each process in a Hyperflow processor contains a current label and a clearance label, but unlike our model, Hyperflow requires the programmer to raise the current label explicitly. This decision avoids the possible side-channel caused by raising the current label depending on sensitive information (a channel that we close in Flamio), at the cost of putting the burden of raising the label on the programmer.

4.7.2 LIO and Coarse-Grained Information Flow

Coarse-grained IFC has traditionally been applied mostly in operating system security [64, 140] by associating a single label with a process. LIO [117] implements coarse-grained IFC as a monadic Haskell library using a current label that can float up to the computation’s clearance label, similar to Flamio.

Recent work [104, 128] on the expressiveness of fine-grained versus coarse-grained IFC shows that they have the same expressive power. Rajani and Garg [104] encourage the use of coarse-grained IFC as it places less annotation burden on the programmer. Based on our experience with Flamio, we agree with this observation.

4.7.3 RPCs and Distributed Computation

Our semantics for remote procedure calls is inspired by the *Location-aware Simple Abstract Machine* (LSAM) [89]. A global LSAM configuration is a set of local LSAM configurations indexed by names, and remote procedure calls are performed by suspending the computation and transferring control to another local LSAM configuration. Whereas LSAM considers a first-order language (useful as a target language for compiling other languages for distributed computation), our semantics handles higher-order functions in the style of the source language in [32], and LSAM would be a natural compilation target for our work.

Much work has been devoted to information-flow programming languages for distributed systems. Both SIF [30] and Swift [29] are based on Jif [83], and target web applications by tracking confidentiality and integrity of data sent between a server and a client. Fabric [67] extends Jif with remote procedure calls and transactions, and enforces security by using a combination of static and dynamic enforcement mechanisms. Fabric and Flamio share many features: Both have trust orderings on principals that can be queried and modified at runtime, and remote nodes communicate via RPC. On the other hand, Fabric and Flamio differ in many ways: Fabric is a language with fine-grained IFC, while Flamio is a coarse-grained system build as a library directly on top of LIO, which in itself is a library in Haskell.

Fabric prevents information leakage from *read channels* (i.e., if node n accesses data on node m depending on information confidential to n , node m learns about n 's confidential information) using *access labels*, but unlike Flamio does not protect against read channels arising from authorization queries [9].

4.8 Conclusion

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

4.9 Acknowledgements

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

Chapter 5

Conclusion and Future Work

This dissertation demonstrates how secure runtime systems for modern programming languages can be implemented, and shown to satisfy strong guarantees about confidentiality and integrity.

To support this conclusion, two artifacts have been presented:

- Pedersen and Askarov [92] presented a formal semantics for a modern programming language with automatic memory management, and a mechanized proof of timing-sensitive security guarantees.
- Pedersen and Askarov [93] presented a programming language for *implementing* modern programming language runtime systems, and a type system for which any well-typed program satisfies a strong notion of timing-sensitive security. The language can implement secure variants of common garbage collection algorithms, as well as secure variants of cooperative thread schedulers.

Many questions are still open. One interesting question is regarding the process of compiling a high-level language to Zee, the language presented in Chapter 3. In particular, what are the semantic security guarantees one would like from such a compilation process, and how does it relate to existing notions of secure compilation [4]?

The dissertation also shows how a complex label model, the Flow-Limited Authorization Model, can be used to reason about confidentiality and integrity during program execution. The experience obtained from implementing the language presented in [96], as well as developing the case studies, supports the conclusions made by Rajani and Garg [104] (and again by [128]):

[...] there seems to be some merit to preferring coarse-grained IFC type systems over fine-grained ones in general.

A number of interesting research directions is also possible for Flamio, the language presented in Chapter 4. The restrictions put on the DEL rule (i.e., the rule for proving trust relationships using delegations) can possibly be relaxed using the concept of label chains, as recently proposed by Kozyri et al. [63]. A label chain $\Omega = \ell_1, \dots$ is a (possibly infinite) sequence of labels such that ℓ_{n+1} represents the sensitivity of the label ℓ_n . If Flamio is extended such that label chains could be used to label a delegation $p \succcurlyeq q$, the delegation rule DEL can be generalized to remove the seemingly arbitrary restriction that sensitivity of the knowledge that *the sensitivity of the information required to prove that the label on the delegation flows to the current strategy principal* must be fully public and fully trusted information.

Part III

Appendix

Appendix A

Static Enforcement of Security in Runtime Systems

A.1 Syntax of Zee

The meta-variable c ranges over commands, and e ranges over expressions. Figure A.1 shows the syntax of commands and expressions. The commands in the gray box refers to commands not present in the surface syntax, but only used during evaluation. The figure also shows the syntax of security labels, ranged over by the meta-variable k , security- and base types, ranged over by the meta-variables s and t , respectively. Finally, the figure also shows the syntax of patterns, ranged over by the meta-variable pat for runtime type analysis.

As Zee allows for type variables to appear at the source level (i.e., from universal or existential quantification), the syntax of security types include a type variable α . This type variable includes a security label k to properly capture the notion of raising a security type to the label of a security label. The syntax of base types include a construct `sizeof s` , representing the size of a value of type s at runtime.

Figure A.2 shows the syntax of function definitions and complete programs in Zee. A function specifies a list of label variables, type variables, and regular variables. Similar to previous work on type systems for IFC, a function also declares a lower bound on the program counter label (i.e., the pc label) and, novel to our work, an upper bound on the frame label (i.e., the fr label). A program \mathcal{P} is a list of function definitions \bar{f} followed by a command c .

Finally, the set of values are ranged over by the meta-variable v . The syntax of *values* is defined in Figure A.3. Values include numbers n , addresses with a version number ν , and pairs consisting of a security level, or type, and a

$$\begin{aligned}
c &::= \text{skip} \mid \text{let } x : s := e \text{ in } c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \mid c; c \\
&\mid *e := e \mid x := *e \mid \text{at } k \text{ with bound } e \text{ do } c \mid \text{if } (k \sqsubseteq k) \text{ then } c \text{ else } c \\
&\mid \text{match } \alpha \text{ with } \overline{pat} \Rightarrow c \mid x := fp \mid f(\overline{k})\langle \overline{s} \rangle(\overline{e}) \\
&\mid \text{let } (\alpha : \mathbf{type} \ k, x : s) := e \text{ in } c \mid \text{let } (\kappa : \mathbf{level} \ k, x : s) := e \text{ in } c \\
&\mid \text{delay } n \mid \text{unscope}(x) \mid \text{epilogue} \\
e &::= n \mid x \mid e \oplus e \mid \text{null} \mid \text{unroll } e \mid \text{roll } e \mid \text{pack } (s, e) \text{ as } \exists \alpha : \mathbf{type} \ k. s \\
&\mid \text{pack } (k, e) \text{ as } \exists \kappa : \mathbf{level} \ k. s \mid \text{sizeof } s \mid \&x \\
k &::= \ell \mid \kappa \mid k \sqcup k \mid k \sqcap k \\
s &::= t \ k \mid \alpha^k \mid \overline{s} \\
t &::= \mathbf{int} \mid k \mapsto s \mid s @ s \mid \exists \alpha : \mathbf{type} \ k. s \mid \exists \kappa : \mathbf{level} \ k. s \mid \mu \alpha : \mathbf{type} \ k. s \\
&\mid \mathbf{size}[s] \\
pat &::= \mathbf{int} \ \kappa \mid (pat @ pat) \ \kappa \mid (\kappa \mapsto pat) \ \kappa \mid \overline{pat} \mid \alpha
\end{aligned}$$

Figure A.1: The syntax of Zee. We write α to mean α^\perp .

$$\begin{aligned}
\mathcal{F} &::= f(\overline{\kappa : \mathbf{level} \ k_1})\langle \overline{\alpha : \mathbf{type} \ k_2} \rangle(\overline{x : s}) \xrightarrow{fr}_{pc} c \\
\mathcal{P} &::= \overline{\mathcal{F}}; c
\end{aligned}$$

Figure A.2: Syntax of function definitions and programs in Zee.

$$\begin{aligned}
v &::= n \mid a_v \mid (\ell, v) \mid (\tau, v) \\
\tau &::= \pi \ell \mid \overline{\tau} \mid \not\downarrow \\
\pi &::= \mathbf{int} \mid \ell \mapsto \tau \mid \tau @ \tau \mid \exists \alpha : \mathbf{type}. s \mid \exists \kappa : \mathbf{level}. s \mid \mu \alpha : \mathbf{type}. s
\end{aligned}$$

Figure A.3: Values in Zee.

value. This represents values of existential types. Runtime representations of security types include a nonsense type value $\not\downarrow$ representing the value of a type of a variable that is not yet in scope. Note that the security labels on \exists and μ types have been erased: they are not needed to guarantee our security policy.

A.2 Semantics of Zee

Figures A.4 and A.5 defines the small-step evaluation judgment. The judgment is written $\langle c, M, P, q \rangle_v \rightarrow \langle c', M', P', q' \rangle_v$, and expresses that command c steps to c' in a single step, and updates the stack (M, P) to (M', P') consuming $q' - q$ execution units.

As the first component, M , of a stack (M, P) is a mapping from *addresses* to values, there must be translation from variable names to stack locations. In practice, this is usually done by a compiler and we abstract away from the exact translation technique by assuming a translation function δ that maps variable names to a natural numbers representing the position in the stack frame. We assume that δ is injective. That is, if $\delta(x) = \delta(y)$ then $x = y$. This assumption rules out optimizations such as coalescing register allocation that stores different variables, with non-overlapping live ranges, at the same stack location.

Relation $\tau \lesssim pat$ specifies that the type value τ *matches* the pattern pat , and is defined in Figure A.6. This relation is needed to define the semantics of pattern matching. We write $\tau_{n_1 \dots n_2}$ for the type value $\tau_{n_1}, \tau_{n_1+1}, \dots, \tau_{n_2}$, where $\tau = \tau_1, \dots, \tau_n$.

When a type value τ matches a pattern pat , the semantics assigns the free variables in pat with values obtained by scrutinizing τ . The interpretation of pat , written $\llbracket pat \rrbracket$, is a function that receives the store, the frame, and the matched type, and returns an updated store and an updated frame. The frame is updated to keep the type information in the frame up-to-date with the local variables in pat .

The semantics needs to compute the size of a runtime representation of a time, which is computed using the function $|\cdot| : \tau \rightarrow \mathbb{N}_\perp$, that optionally returns an undefined value \perp when invoked on nonsense types $\not\downarrow$.

Evaluation of an expression e is given by the big-step judgment $\langle e, m, p \rangle \Downarrow v$, expressing that e evaluates to v when giving the stack (m, p) .

Similarly, Figure A.10 defines the evaluation judgment for security types and security labels.

Finally, Figure A.11 defines the evaluation judgment for evaluation of base types.

A.3 A Type System for Zee

We now describe the type system of Zee. Throughout this section we use the following abbreviation to define the type of the frame pointer:

$$T_{st}(pc, fr, k) = (\mu \alpha : \mathbf{type} \ k. (\exists \beta : \mathbf{type} \ fr. (\exists \gamma : \mathbf{type} \ fr. (\beta \cdot \alpha @ \gamma) pc) \perp) \perp) \perp$$

Figure A.13 describes the typing relation, written $\Gamma, \Pi, \phi, pc, fr \vdash c$, stating that command c is well-typed under typing environments Γ and Π assuming the flow relations in ϕ and with a program counter- and frame label pc and

$$\begin{array}{c}
\text{S-IF-T} \quad \frac{\langle e, M, P \rangle \Downarrow n \quad n \neq 0}{\langle \text{if } e \text{ then } c_1 \text{ else } c_2, M, P, q \rangle_{\mathbf{v}} \rightarrow \langle c_1, M, P, q+1 \rangle_{\mathbf{v}}} \quad \text{S-IF-F} \quad \frac{\langle e, M, P \rangle \Downarrow 0}{\langle \text{if } e \text{ then } c_1 \text{ else } c_2, M, P, q \rangle_{\mathbf{v}} \rightarrow \langle c_2, M, P, q+1 \rangle_{\mathbf{v}}} \quad \text{S-WHILE-F} \quad \frac{\langle e, M, P \rangle \Downarrow 0}{\langle \text{while } e \text{ do } c, M, P, q \rangle_{\mathbf{v}} \rightarrow \langle \text{stop}, M, P, q+1 \rangle_{\mathbf{v}}} \\
\\
\text{S-WHILE-T} \quad \frac{\langle e, M, P \rangle \Downarrow n \quad n \neq 0}{\langle \text{while } e \text{ do } c, M, P, q \rangle_{\mathbf{v}} \rightarrow \langle c; \text{while } e \text{ do } c, M, P, q+1 \rangle_{\mathbf{v}}} \quad \text{S-SKIP} \quad \frac{}{\langle \text{skip}, M, P, q \rangle_{\mathbf{v}} \rightarrow \langle \text{stop}, M, P, q+1 \rangle_{\mathbf{v}}} \\
\\
\text{S-ASGN} \quad \frac{\langle e, m \cdot M, P \rangle \Downarrow v \quad m' = m[\delta(x) + \text{fp}(m) \mapsto v]}{\langle x := e, m \cdot M, P, q \rangle_{\mathbf{v}} \rightarrow \langle \text{stop}, m' \cdot M, P, q+1 \rangle_{\mathbf{v}}} \\
\\
\text{S-WRITE} \quad \frac{\begin{array}{c} m_i = (m_{\mathbb{I}}, v_i) \in M \quad a \in \mathbb{I} \quad v_i \leq \gamma \\ \langle e_1, M, P \rangle \Downarrow a_{\gamma} \quad \langle e_2, M, P \rangle \Downarrow v \end{array}}{\langle *e_1 := e_2, M, P, q \rangle_{\mathbf{v}} \rightarrow \langle \text{stop}, M[a \mapsto v], P, q+1 \rangle_{\mathbf{v}}} \\
\\
\text{S-READ} \quad \frac{\begin{array}{c} M = m \cdot M' \quad m_i = (m_{\mathbb{I}}, v_i) \in M \quad a \in \mathbb{I} \quad v_i \leq \gamma \\ \langle e, M, P \rangle \Downarrow a_{\gamma} \quad m' = m[\delta(x) + \text{fp}(m) \mapsto M(n)] \end{array}}{\langle x := *e, m \cdot M, P, q \rangle_{\mathbf{v}} \rightarrow \langle \text{stop}, m' \cdot M', P, q+1 \rangle_{\mathbf{v}}} \quad \text{S-AT} \quad \frac{\langle e, m, P \rangle \Downarrow n}{\langle \text{at } k \text{ with bound } e \text{ do } c, m, P, q \rangle_{\mathbf{v}} \rightarrow \langle c; \text{delay } n, m, P, q+1 \rangle_{\mathbf{v}}} \\
\\
\text{S-DELAY} \quad \frac{n \leq q}{\langle \text{delay } n, m, P, q \rangle_{\mathbf{v}} \rightarrow \langle \text{delay } n, m, P, n+1 \rangle_{\mathbf{v}}} \quad \text{S-FP} \quad \frac{\begin{array}{c} v = (\text{cod}(p_{\text{arg}}), (\text{cod}(p_{\text{local}}), \text{fp}(m)_{\mathbf{v}})) \\ m = (m_{\mathbb{I}}, v) \quad m' = m[\delta(x) + \text{fp}(m) \mapsto v] \end{array}}{\langle x := \text{fp}, m \cdot M, p \cdot P, q \rangle_{\mathbf{v}} \rightarrow \langle \text{stop}, m' \cdot M, p \cdot P, q+1 \rangle_{\mathbf{v}}} \\
\\
\text{S-LET} \quad \frac{\begin{array}{c} M = m \cdot M' \quad \langle s, p \rangle \Downarrow_{\text{type}} \tau \quad \langle e, m, p \rangle \Downarrow v \\ m' = m[\delta(x) + \text{fp}(m) \mapsto v] \quad p' = p[p_{\text{local}} \mapsto p_{\text{local}}[x \mapsto \tau]] \end{array}}{\langle \text{let } x : s := e \text{ in } c, M, p \cdot P, q \rangle_{\mathbf{v}} \rightarrow \langle c; \text{unscope}(x), m' \cdot M', p' \cdot P, q+1 \rangle_{\mathbf{v}}} \\
\\
\text{S-UNSCOPE} \quad \frac{p' = p[\text{local} \mapsto p.\text{local}[x \mapsto \zeta]]}{\langle \text{unscope}(x), M, p \cdot P, q \rangle_{\mathbf{v}} \rightarrow \langle \text{stop}, M, p' \cdot P, q+1 \rangle_{\mathbf{v}}}
\end{array}$$

Figure A.4: Small-step relation for commands.

S-UNPACK-LEV

$$\begin{array}{c}
P = p \cdot P' \quad \langle s, p' \rangle \Downarrow_{\text{type}} \tau \quad \langle e, m, p \rangle \Downarrow (\ell_1, v_2) \\
p' = p[p_{\text{var}} \mapsto p_{\text{var}}[\kappa \mapsto \ell_1], p_{\text{local}} \mapsto p_{\text{local}}[x \mapsto \tau]] \\
M = m \cdot M' \quad m' = m[\delta(x) + \text{fp}(m) \mapsto v_2] \\
\hline
\langle \text{let } (\kappa : \mathbf{level} \ k, x : s) := e \text{ in } c, M, P, q \rangle_v \\
\rightarrow \langle c; \text{unscope}(x), m' \cdot M', p' \cdot P', q + 1 \rangle_v
\end{array}$$

S-UNPACK-TY

$$\begin{array}{c}
P = p \cdot P' \quad \langle s, p' \rangle \Downarrow_{\text{type}} \tau \quad \langle e, m, p \rangle \Downarrow (\tau_1, v_2) \\
p' = p[p_{\text{var}} \mapsto p_{\text{var}}[\alpha \mapsto \tau_1], p_{\text{local}} \mapsto p_{\text{local}}[x \mapsto \tau]] \\
M = m \cdot M' \quad m' = m[\delta(x) + \text{fp}(m) \mapsto v_2] \\
\hline
\langle \text{let } (\alpha : \mathbf{type} \ k, x : s) := e \text{ in } c, M, P, q \rangle_v \\
\rightarrow \langle c; \text{unscope}(x), m' \cdot M', p' \cdot P', q + 1 \rangle_v
\end{array}$$

S-EPILOGUE

$$\begin{array}{c}
\langle \text{epilogue}, (\mathbb{I}_1, |m_1|, v_1) \cdot (\mathbb{I}_2, |m_2|, v_2) \cdot M, p \cdot P, q \rangle_v \\
\rightarrow \langle \text{stop}, (\mathbb{I}_2, |m_2|, \max(v_1, v_2) + 1) \cdot M, P, q + 1 \rangle_v
\end{array}$$

S-MATCH

$$\begin{array}{c}
\text{argmin}_{i=1, \dots, n} (\tau \lesssim \text{pat}_i) = j \quad \langle \alpha, p \rangle \Downarrow_{\text{type}} \tau \quad \llbracket \text{pat}_j \rrbracket(p, \tau) = p' \\
\hline
\langle \text{match } \alpha \text{ with } (\text{pat}_i \Rightarrow c_i)_{i=1, \dots, n}, M, p \cdot P, q \rangle_v \rightarrow \langle c_j, M, p' \cdot P, q + 1 \rangle_v
\end{array}$$

S-CALL

$$\begin{array}{c}
\mathbb{F}(f) = \langle \kappa_1, \dots, \kappa_n \rangle \langle \alpha_1, \dots, \alpha_m \rangle (x_1 : s'_1, \dots, x_r : s'_r) = c \\
\langle k_i, P \rangle \Downarrow_{\text{lab}} \ell_i \quad \langle s_i, P \rangle \Downarrow_{\text{type}} \tau_i \quad \langle e_i, M, P \rangle \Downarrow v_i \\
\langle s'_i, P' \rangle \Downarrow_{\text{type}} \tau'_i \quad M = m \cdot M' \quad m' = (\mathbb{I}', |m'|, v) \\
P = (p_{\text{var}}, p_{\text{args}}, p_{\text{local}}) \cdot P' \quad p' = (p'_{\text{var}}, p'_{\text{arg}}, p'_{\text{local}}) \\
p'_{\text{var}} = \{\kappa_i \mapsto \ell_i \mid i = 1, \dots, n\} \cup \{\alpha_i \mapsto \tau_i \mid i = 1, \dots, m\} \\
p'_{\text{arg}} = \{x_i \mapsto \tau'_i \mid i = 1, \dots, r\} \quad p'_{\text{local}} = \{x \mapsto \perp \mid x \in c\} \\
\mathbb{I}' = \{\delta(x_i) + \text{sp}(m) \mid i = 1, \dots, r\} \\
\quad \cup \{\text{sp}(m)\} \cup \{\delta(z) + \text{sp}(m) \mid z \in c\} \\
|m'| = \{\text{sp}(m) \mapsto (\text{cod}(p_{\text{arg}}), (\text{cod}(p_{\text{local}}), \text{fp}(m)_v))\} \\
\quad \cup \{\delta(x_i) + \text{sp}(m) \mapsto v_i \mid i = 1, \dots, r\} \\
\hline
\langle f \langle k_1, \dots, k_n \rangle \langle s_1, \dots, s_m \rangle (e_1, \dots, e_r), M, P, q \rangle_v \\
\rightarrow \langle c; \text{epilogue}, m' \cdot M, p' \cdot P, q + 1 \rangle_{v+1}
\end{array}$$

S-SEQ-CONT

$$\begin{array}{c}
c'_1 \neq \text{stop} \\
\langle c_1, m, P, q \rangle_v \rightarrow \langle c'_1, m', P', q' \rangle_{v'} \\
\hline
\langle c_1; c_2, m, P, q \rangle_v \rightarrow \langle c'_1; c_2, m', P', q' \rangle_{v'}
\end{array}$$

S-SEQ-STOP

$$\begin{array}{c}
\langle c_1, m, P, q \rangle_v \rightarrow \langle \text{stop}, m', P', q' \rangle_{v'} \\
\hline
\langle c_1; c_2, m, P, q \rangle_v \rightarrow \langle c_2, m', P', q' \rangle_{v'}
\end{array}$$

S-INST

$$\begin{array}{c}
\langle c, m, P, h, q \rangle_v \rightarrow \langle c', m', P', h', q' \rangle_{v'} \\
\hline
\langle c, m, P, h, q \rangle_v \rightarrow \langle c', m', P', h', q' \rangle_{v'}
\end{array}$$

Figure A.5: Small-step relation for commands (cont).

$$\begin{array}{c}
\frac{}{\mathbf{int} \ell \lesssim \mathbf{int} \kappa} \qquad \frac{\tau \lesssim pat}{(\ell_1 \mapsto \tau) \ell_2 \lesssim (\kappa_1 \mapsto pat) \kappa_2} \qquad \frac{\tau_i \lesssim pat_i \ i = 1, 2}{(\tau_1 @ \tau_2) \ell \lesssim (pat_1 @ pat_2) \kappa} \\
\\
\frac{}{\tau \lesssim \alpha} \qquad \frac{\begin{array}{c} |\bar{\tau}| = n \quad |\overline{pat}| = m \quad m \leq n \\ \forall i \in \{1, \dots, m-1\} . \tau_i \lesssim pat_i \quad \tau_{m \dots n} \lesssim pat_m \end{array}}{\bar{\tau} \lesssim \overline{pat}}
\end{array}$$

Figure A.6: Semantics for pattern matching.

$$\begin{array}{c}
\llbracket \mathbf{int} \kappa \rrbracket(p, \mathbf{int} \ell) = p[\kappa \mapsto \ell] \qquad \frac{\llbracket pat_1 \rrbracket(p, \tau_1) = p' \quad \llbracket pat_2 \rrbracket(p', \tau_2) = p''}{\llbracket (pat_1 @ pat_2) \kappa \rrbracket(p, (\tau_1 @ \tau_2) \ell) = p''[\kappa \mapsto \ell]} \\
\\
\frac{\begin{array}{c} \llbracket pat \rrbracket(p, \tau) = p' \\ p'' = p'[\kappa_1 \mapsto \ell_1, \kappa_2 \mapsto \ell_2] \end{array}}{\llbracket (\kappa_1 \mapsto pat) \kappa_2 \rrbracket(p, (\ell_1 \mapsto \tau) \ell_2) = p''} \qquad \llbracket \alpha \rrbracket(p, \tau) = p[\alpha \mapsto \tau] \\
\\
\frac{\begin{array}{c} |\bar{\tau}| = n \quad |\overline{pat}| = m \quad m \leq n \quad p_0 = p \\ \forall i \in \{1, \dots, m-1\} . \llbracket pat_i \rrbracket(p_{i-1}, \tau_i) = p_i \\ \llbracket pat_m \rrbracket(p_{m-1}, \tau_{m \dots n}) = p' \end{array}}{\llbracket \overline{pat} \rrbracket(p, \bar{\tau}) = p'}
\end{array}$$

Figure A.7: Semantics for closing the free variables in a pattern.

$$\begin{array}{c}
|\pi \ell| = 1 \\
|\tau_1, \dots, \tau_n| = \sum_{i=1}^n |\tau_i| \\
|\bot| = \bot
\end{array}$$

where $\bot + n = n + \bot = \bot$ for all $n \in \mathbb{N}$.

Figure A.8: Computing the size of a runtime representation of a type.

$$\begin{array}{c}
\text{E-NUM} \\
\frac{}{\langle n, m, p \rangle \Downarrow n} \\
\\
\text{E-VAR} \\
\frac{m(\delta(x) + \text{fp}(m)) = v}{\langle x, m, p \rangle \Downarrow v} \\
\\
\text{E-BINOP} \\
\frac{\langle e_i, m, p \rangle \Downarrow v_i \quad v_1 \oplus v_2 = v}{\langle e_1 \oplus e_2, m, p \rangle \Downarrow v} \\
\\
\text{E-NULL} \\
\frac{v = \text{VERSION}(m)}{\langle \text{null}, m, p \rangle \Downarrow 0_v} \\
\\
\text{E-SIZEOF} \\
\frac{\langle s, p \rangle \Downarrow_{\text{type}} \tau}{\langle \text{sizeof } s, m, p \rangle \Downarrow |\tau|} \\
\\
\text{E-PACK-TY} \\
\frac{\langle s, p \rangle \Downarrow_{\text{type}} \tau \quad \langle e, m, p \rangle \Downarrow v}{\langle \text{pack } (s, e) \text{ as } _, m, p \rangle \Downarrow (\tau, v)} \\
\\
\text{E-PACK-LEV} \\
\frac{\langle k, p \rangle \Downarrow_{\text{lab}} \ell \quad \langle e, m, p \rangle \Downarrow v}{\langle \text{pack } (k, e) \text{ as } _, m, p \rangle \Downarrow (\ell, v)} \\
\\
\text{E-UNROLL} \\
\frac{\langle e, m, p \rangle \Downarrow v}{\langle \text{unroll } e, m, p \rangle \Downarrow v} \\
\\
\text{E-ROLL} \\
\frac{\langle e, m, p \rangle \Downarrow v}{\langle \text{roll } e, m, p \rangle \Downarrow v} \\
\\
\text{E-ADDR OF} \\
\frac{v = \text{VERSION}(m)}{\langle \&x, m, p \rangle \Downarrow (\delta(x) + \text{fp}(m))_v} \\
\\
\text{E-SIZEOF} \\
\frac{\langle s, p \rangle \Downarrow_{\text{type}} \tau}{\langle \text{sizeof } s, m, p \rangle \Downarrow |\tau|}
\end{array}$$

Figure A.9: Semantics for expression evaluation.

$$\begin{array}{c}
\text{E-SECTY-SECTY} \\
\frac{\langle t, p \rangle \Downarrow \pi \quad \langle k, p \rangle \Downarrow_{\text{lab}} \ell}{\langle tk, p \rangle \Downarrow_{\text{type}} \pi \ell} \\
\\
\text{E-SECTY-PROD} \\
\frac{\langle s_i, p \rangle \Downarrow_{\text{type}} \tau_i \quad i = 1, \dots, n}{\langle \bar{s}, p \rangle \Downarrow_{\text{type}} \bar{\tau}} \\
\\
\text{E-SECTY-VAR} \\
\frac{p(\alpha) = \tau}{\langle \alpha, p \rangle \Downarrow_{\text{type}} \tau} \\
\\
\text{E-LEV-VAR} \\
\frac{p(\kappa) = \ell}{\langle \kappa, p \rangle \Downarrow_{\text{lab}} \ell} \\
\\
\text{E-LEV-JOIN} \\
\frac{\langle k_i, p \rangle \Downarrow_{\text{lab}} \ell_i}{\langle k_1 \sqcup k_2, p \rangle \Downarrow_{\text{lab}} \ell_1 \sqcup \ell_2} \\
\\
\text{E-LEV-MEET} \\
\frac{\langle k_i, p \rangle \Downarrow_{\text{lab}} \ell_i}{\langle k_1 \sqcap k_2, p \rangle \Downarrow_{\text{lab}} \ell_1 \sqcap \ell_2} \\
\\
\text{E-LEV-LIT} \\
\langle \ell, p \rangle \Downarrow_{\text{lab}} \ell
\end{array}$$

Figure A.10: Semantics for security type evaluation.

$$\begin{array}{c}
\text{E-TY-INT} \\
\langle \text{int}, p \rangle \Downarrow \text{int} \\
\\
\text{E-TY-PTR} \\
\frac{\langle k, p \rangle \Downarrow_{\text{lab}} \ell \quad \langle s, p \rangle \Downarrow_{\text{type}} \tau}{\langle k \mapsto s, p \rangle \Downarrow \ell \mapsto \tau} \\
\\
\text{E-TY-SPTR} \\
\frac{\langle s_i, p \rangle \Downarrow_{\text{type}} \tau_i \quad i = 1, 2}{\langle s_1 @ s_2, p \rangle \Downarrow \tau_1 @ \tau_2} \\
\\
\text{E-TY-EX-TY} \\
\langle \exists \alpha : \mathbf{type} \ k. s, p \rangle \Downarrow \exists \alpha : \mathbf{type}. s \\
\\
\text{E-TY-EX-LEV} \\
\langle \exists \kappa : \mathbf{level} \ k. s, p \rangle \Downarrow \exists \kappa : \mathbf{level}. s \\
\\
\text{E-TY-REC} \\
\langle \mu \alpha : \mathbf{type} \ k. s, p \rangle \Downarrow \mu \alpha : \mathbf{type}. s \\
\\
\text{E-TY-SIZEOF} \\
\frac{\langle s, p \rangle \Downarrow_{\text{type}} \tau}{\langle \text{size}[s], p \rangle \Downarrow \text{int}}
\end{array}$$

Figure A.11: Semantics for base type evaluation.

$$\begin{aligned} (t\ k_1)^{k_2} &= t\ (k_1 \sqcup k_2) \\ (s_1, \dots, s_n)^k &= s_1^k, \dots, s_n^k \\ (\alpha^{k_1})^{k_2} &= \alpha^{k_1 \sqcup k_2} \end{aligned}$$

Figure A.12: Raising the sensitivity of a security type by a label.

fr, respectively. Figure A.12 defines an operation on security types for raising the sensitivity of the security type with a label. This is most often used to capture the requirement that one security type s' has to be more sensitive than another security type s , but also be more sensitive than the program counter label or frame label, i.e., $s^{pc} <: s'$.

Relation $\Pi \vdash pat \rightsquigarrow_k \Pi' : s$ defined in Figure A.14 is used to extend the typing environment when type checking commands executed after match command. The relation specifies that a scrutinee can safely be assigned type s if the runtime value matches the pattern pat assuming environment Π is updated to Π' . Finally, the label k represents an upper bound on the information that influences the type of s .

Relation $s_1 \llbracket \oplus \rrbracket s_2 \rightarrow s$, given in Figure A.15, computes the type s of the result of evaluating a binary expression \oplus on two expressions of type s_1 and s_2 respectively.

Figure A.16 defines the typing relation for expressions. Relation $\Gamma, \Pi, \phi \vdash e : s$ specifies that expression e has type s in the typing environment Γ and assuming the constraints ϕ .

Relations $\Pi, \phi \vdash_{\text{type}} s : k$, $\Pi, \phi \vdash_{\text{type}} t : k$, and $\Pi; \phi \vdash_{\text{lab}} k : k'$, defined in Figure A.17, defines well-formedness rules for security types, base types and security labels.

A.4 Semantic Well-Formedness

Relation $P \vdash k_1 \sqsubseteq k_2$ specifies that $\ell_1 \sqsubseteq \ell_2$ holds, where k_i evaluates to ℓ_i in stack frame P .

$$\begin{array}{c} \ell_1 \sqsubseteq \ell_2 \\ \hline \langle k_i, P \rangle \Downarrow_{\text{lab}} \ell_i \quad i = 1, 2 \\ \hline P \models k_1 \sqsubseteq k_2 \end{array} \quad \begin{array}{c} \ell_1 \not\sqsubseteq \ell_2 \\ \hline \langle k_i, P \rangle \Downarrow_{\text{lab}} \ell_i \quad i = 1, 2 \\ \hline P \models k_1 \not\sqsubseteq k_2 \end{array} \quad \begin{array}{c} P \models \phi_i \quad i = 1, 2 \\ \hline P \models \phi_1 \wedge \phi_2 \end{array}$$

$$\begin{array}{c}
\text{T-LET} \\
\frac{\Gamma, \Pi, \phi \vdash e : r \quad \Pi, \phi \vdash_{\text{type}} s : k \quad \phi \vdash r^{pc} <: s \quad fr' = fr \sqcup k \quad \Gamma[x \mapsto s], \Pi, \phi, pc, fr' \vdash c}{\Gamma, \Pi, \phi, pc, fr \vdash \text{let } x : s := e \text{ in } c}
\\
\\
\text{T-AT} \\
\frac{\Pi; \phi \vdash_{\text{lab}} k : pc \quad \Gamma, \Pi, \phi \vdash e : \mathbf{int} pc \quad \phi \vdash pc \sqsubseteq k \quad \Gamma, \Pi, \phi, k, fr \vdash c}{\Gamma, \Pi, \phi, pc, fr \vdash \text{at } k \text{ with bound } e \text{ do } c}
\\
\\
\begin{array}{cc}
\text{T-IF} & \text{T-FP} \\
\frac{\Gamma, \Pi, \phi \vdash e : \mathbf{int} pc \quad \Gamma, \Pi, \phi, pc, fr \vdash c_i \quad i = 1, 2}{\Gamma, \Pi, \phi, pc, fr \vdash \text{if } e \text{ then } c_1 \text{ else } c_2} & \frac{\Pi; \phi \vdash_{\text{lab}} fr : k \quad \phi \vdash T_{\text{st}}(pc, fr, k)^{pc} <: \Gamma(x)}{\Gamma, \Pi, \phi, pc, fr \vdash x := \text{fp}}
\end{array}
\\
\\
\text{T-MATCH} \\
\frac{\Pi(\alpha) = \mathbf{type} k \quad \phi \vdash k \sqsubseteq pc \quad \Pi \vdash pat_i \leadsto_k \Pi_i : s_i \quad \Gamma[s_i/\alpha], \Pi_i[s_i/\alpha], \phi, pc, fr \vdash c_i[s_i/\alpha]}{\Gamma, \Pi, \phi, pc, fr \vdash \text{match } \alpha \text{ with } \overline{pat} \Rightarrow \overline{c}}
\\
\\
\text{T-UNPACK-TY} \\
\frac{\Gamma, \Pi, \phi \vdash e : (\exists \alpha : \mathbf{type} k_1. r) pc \quad \phi \vdash r^{pc} <: s \quad \Gamma' = \Gamma[x \mapsto s] \quad \Pi' = \Pi[\alpha \mapsto \mathbf{type}_{k_1}] \quad \Pi', \phi \vdash_{\text{type}} r : k_2 \quad fr' = fr \sqcup k_1 \sqcup k_2 \quad \Gamma', \Pi', \phi, pc, fr' \vdash c}{\Gamma, \Pi, \phi, pc, fr \vdash \text{let } (\alpha : \mathbf{type} k_1, x : s) := e \text{ in } c}
\\
\\
\text{T-UNPACK-LEV} \\
\frac{\Gamma, \Pi, \phi \vdash e : (\exists \kappa : \mathbf{level} k_1. r) pc \quad \phi \vdash r^{pc} <: s \quad \Gamma' = \Gamma[x \mapsto s] \quad \Pi' = \Pi[\kappa \mapsto \mathbf{level}_{k_1}] \quad \Pi', \phi \vdash_{\text{type}} r : k_2 \quad fr' = fr \sqcup k_1 \sqcup k_2 \quad \Gamma', \Pi', \phi, pc, fr' \vdash c}{\Gamma, \Pi, \phi, pc, fr \vdash \text{let } (\kappa : \mathbf{level} k_1, x : s) := e \text{ in } c}
\\
\\
\begin{array}{cc}
\text{T-FlowsTo} & \text{T-INST-C} \\
\frac{\Pi; \phi \vdash_{\text{lab}} k_i : pc \quad \Gamma, \Pi, \phi \wedge k_1 \sqsubseteq k_2, pc, fr \vdash c_1 \quad \Gamma, \Pi, \phi \wedge k_1 \not\sqsubseteq k_2, pc, fr \vdash c_2}{\Gamma, \Pi, \phi, pc, fr \vdash \text{if } (k_1 \sqsubseteq k_2) \text{ then } c_1 \text{ else } c_2} & \frac{\Gamma, \Pi, \phi, pc, fr \vdash c}{\Gamma, \Pi, \phi, pc, fr \vdash c}
\end{array}
\\
\\
\begin{array}{cc}
\text{T-WHILE} & \text{T-SEQ} \\
\frac{\Gamma, \Pi, \phi \vdash e : \mathbf{int} pc \quad \Gamma, \Pi, \phi, pc, fr \vdash c}{\Gamma, \Pi, \phi, pc, fr \vdash \text{while } e \text{ do } c} & \frac{\Gamma, \Pi, \phi, pc, fr \vdash c_i \quad i = 1, 2}{\Gamma, \Pi, \phi, pc, fr \vdash c_1; c_2}
\end{array}
\\
\\
\text{T-CALL} \\
\frac{\mathbb{F}(f) = \langle \kappa_1 : k_1^1, \dots, \kappa_n : k_n^1 \rangle \langle \alpha_1 : k_1^2, \dots, \alpha_m : k_m^2 \rangle (x_1 : s_1, \dots, x_r : s_r) \xrightarrow[k_{pc}]{k_{fr}} 1 \quad \Pi; \phi \vdash_{\text{lab}} k_i : k_i^1[k_{i-1}/\kappa_{i-1}, \dots, k_1/\kappa_1] \quad \Pi, \phi \vdash_{\text{type}} s_i : k_i^2[k_n/\kappa_n, \dots, k_1/\kappa_1][s_{i-1}/\alpha_{i-1}, \dots, s_1/\alpha_1] \quad \Gamma, \Pi, \phi \vdash e_i : s_i[k_n/\kappa_n, \dots, k_1/\kappa_1][s_m/\alpha_m, \dots, s_1/\alpha_1] \quad \phi \vdash k_{pc}[k_n/\kappa_n, \dots, k_1/\kappa_1] = pc \quad \phi \vdash fr \sqsubseteq k_{fr}[k_n/\kappa_n, \dots, k_1/\kappa_1]}{\Gamma, \Pi, \phi, pc, fr \vdash f\langle k_1, \dots, k_n \rangle \langle s_1, \dots, s_m \rangle (e_1, \dots, e_r)}
\end{array}$$

Figure A.13: Typing relation for commands.

$$\begin{array}{c}
\Pi \vdash \mathbf{int} \ \kappa \rightsquigarrow_{\kappa} \Pi[\kappa \mapsto \mathbf{level} \ k] : \mathbf{int} \ \kappa \qquad \Pi \vdash \alpha \rightsquigarrow_{\kappa} \Pi[\alpha \mapsto \mathbf{type}_{\kappa}] : \alpha \\
\\
\frac{\Pi \vdash pat_1 \rightsquigarrow_{\kappa} \Pi_1 : s_1 \quad \Pi_1 \vdash pat_2 \rightsquigarrow_{\kappa} \Pi_2 : s_2}{\Pi \vdash (pat_1 @ pat_2) \ \kappa \rightsquigarrow_{\kappa} \Pi'[\kappa \mapsto \mathbf{level}_{\kappa}] : (s_1 @ s_2) \ \kappa} \\
\\
\frac{\Pi \vdash pat \rightsquigarrow_{\kappa} \Pi' : s}{\Pi \vdash (\kappa_1 \mapsto pat) \ \kappa_2 \rightsquigarrow_{\kappa} \Pi'[\kappa_1 \mapsto \mathbf{level}_{\kappa}, \kappa_2 \mapsto \mathbf{level} \ k] : (\kappa_1 \mapsto s) \ \kappa_2} \\
\\
\frac{\begin{array}{c} \Pi_0 = \Pi \\ \Pi_{i-1} \vdash pat_i \rightsquigarrow_{\kappa} \Pi_i : s_i \quad i = 1, \dots, n \end{array}}{\Pi \vdash p_1, \dots, p_n \rightsquigarrow_{\kappa} \Pi_n : s_1, \dots, s_n}
\end{array}$$

Figure A.14: Generating typing environments.

$$\begin{array}{l}
\mathbf{int} \ k_1 \llbracket \oplus \rrbracket \mathbf{int} \ k_2 \rightarrow \mathbf{int} \ (k_1 \sqcup k_2) \\
(s_1 @ s \cdot s_2) \ k_1 \llbracket + \rrbracket \mathbf{size}[s] \ k_2 \rightarrow (s_1 \cdot s @ s_2) \ (k_1 \sqcup k_2) \\
(s_1 \cdot s @ s_2) \ k_1 \llbracket - \rrbracket \mathbf{size}[s] \ k_2 \rightarrow (s_1 @ s \cdot s_2) \ (k_1 \sqcup k_2) \\
(k \mapsto s) \ k_1 \llbracket + \rrbracket \mathbf{int} \ k_2 \rightarrow (k \mapsto s) \ (k_1 \sqcup k_2) \\
(k \mapsto s) \ k_1 \llbracket - \rrbracket \mathbf{int} \ k_2 \rightarrow (k \mapsto s) \ (k_1 \sqcup k_2)
\end{array}$$

Figure A.15: Relation specifying the type of the result of a binary operation when given the type of the two operands.

Figure A.18 describes well-formedness for configurations, which uses well-formedness of values. This relation is in defined in Figure A.19.

A.5 Attacker Model

In this section, we define the attacker model we consider.

We define a semantics augmented with events, given by the following grammar:

$$\begin{array}{l}
ev ::= \varepsilon \mid \mathbf{rd}(x \leftarrow v, q) \\
\mid \mathbf{unp}(\ell, x : \tau \leftarrow v, q) \mid \mathbf{let}(x : \tau \leftarrow v, q) \mid \textcolor{red}{ev}
\end{array}$$

We extend the small-step operational semantics, defined in Figure A.4, with events, and write the augmented semantics as $\langle c, m, P, h, q \rangle_{\mathbf{v}} \xrightarrow{ev} \langle c', m', P', h', q' \rangle_{\mathbf{v}'}$.

We write $p_1, p_2 \vdash v_1 =_{\mathcal{A}}^z v_2 : \tau_1 \times \tau_2$, when an attacker at level \mathcal{A} cannot distinguish values v_1 and v_2 . This judgment is defined in Figure A.20.

$\frac{}{\Gamma, \Pi, \phi \vdash n : \mathbf{int} \perp}$	$\frac{}{\Gamma, \Pi, \phi \vdash x : \Gamma(x)}$	$\frac{\text{T-PACK-TY} \quad \begin{array}{l} \Pi, \phi \vdash_{\text{type}} s_2 \quad \Gamma, \Pi, \phi \vdash e : s_2[s_1/x] \\ \Pi, \phi \vdash_{\text{type}} s_1 : k_1 \quad t = \exists x : \mathbf{type} \, k_1. s_2 \end{array}}{\Gamma, \Pi, \phi \vdash \text{pack}(s_1, e) \text{ as } t : t \perp}$
$\frac{\text{T-SIZEOF} \quad \Pi, \phi \vdash_{\text{type}} s : k}{\Gamma, \Pi, \phi \vdash \text{sizeof } s : \mathbf{size}[s] \, k}$	$\frac{\text{T-BINOP} \quad \begin{array}{l} \Gamma, \Pi, \phi \vdash e_i : s_i \\ s_1 \llbracket \oplus \rrbracket s_2 \rightarrow s \end{array}}{\Gamma, \Pi, \phi \vdash e_1 \oplus e_2 : s}$	$\frac{\text{T-UNROLL} \quad \Gamma, \Pi, \phi \vdash e : (\mu \alpha : \mathbf{type} \, k_1. s) \, k_2}{\Gamma, \Pi, \phi \vdash \text{unroll } e : s[(\mu \alpha : \mathbf{type} \, k_1. s) \, k_2 / \alpha] \, k_2}$
$\frac{\text{T-ROLL} \quad \Gamma, \Pi, \phi \vdash e : s[(\mu \alpha : \mathbf{type} \, k_1. s) \, k_2 / \alpha] \, k_3}{\Gamma, \Pi, \phi \vdash \text{roll } e : (\mu \alpha : \mathbf{type} \, k_1. s) \, k_2 \sqcup k_3}$	$\frac{\text{T-NUL-HEAP} \quad \begin{array}{l} \Pi, \phi \vdash_{\text{type}} s \\ \Pi; \phi \vdash_{\text{lab}} k \end{array}}{\Gamma, \Pi, \phi \vdash \text{null} : (k \mapsto s) \perp}$	$\frac{\text{T-NUL-STACK} \quad \Pi, \phi \vdash_{\text{type}} s_i \quad i = 1, 2}{\Gamma, \Pi, \phi \vdash \text{null} : (s_1 @ s_2) \perp}$
$\frac{\text{T-PACK-LEV} \quad \begin{array}{l} \Pi, \phi \vdash_{\text{type}} s \quad \Gamma, \Pi, \phi \vdash e : s[k/\kappa] \\ \Pi; \phi \vdash_{\text{lab}} k : k' \quad t = \exists \kappa : \mathbf{level} \, k'. s \end{array}}{\Gamma, \Pi, \phi \vdash \text{pack}(k, e) \text{ as } t : t \perp}$	$\frac{\text{T-ADDR-OF} \quad \Gamma(x) = s}{\Gamma, \Pi, \phi \vdash \&x : (@ s) \perp}$	$\frac{\text{T-SUB} \quad \Gamma, \Pi, \phi \vdash e : s_1 \quad \phi \vdash s_1 <: s_2}{\Gamma, \Pi, \phi \vdash e : s_2}$

Figure A.16: Typing relation for expressions.

As both security types and security labels have a runtime representation, we need to specify which security type values and levels an attacker considers equivalent. Figure A.21 formalizes these notions.

Finally, Figure A.22 defines \mathcal{A} -equivalence for events. This relation is written $\Gamma \mid p_1, p_2 \vdash ev =_{\mathcal{A}} ev$. Figure A.23 defines \mathcal{A} -observability of events, specifying when an attacker \mathcal{A} can observe the event ev . This relation uses observability on security type values, which we define as

$$\pi \ell \sqsubseteq \mathcal{A} \triangleq \ell \sqsubseteq \mathcal{A}$$

Note that neither a product security type, nor a nonsense security type, is ever observable. This is due to the fact that no variable will ever be typed with these constructors.

Given a trace t we write $\lfloor t \rfloor_{\mathcal{A}}$ for the trace containing only \mathcal{A} -observable events. It is defined inductively as follows:

$$\begin{aligned} \lfloor \varepsilon \rfloor_{\mathcal{A}} &= \varepsilon \\ \lfloor (ev, \Gamma, p) \cdot t \rfloor_{\mathcal{A}} &= \begin{cases} (ev, \Gamma, p) \cdot \lfloor t \rfloor_{\mathcal{A}} & \Gamma, p \vdash ev \sqsubseteq \mathcal{A} \\ \lfloor t \rfloor_{\mathcal{A}} & \text{otherwise} \end{cases} \end{aligned}$$

$\frac{\text{T-SEC-TYPE} \quad \begin{array}{c} \Pi, \phi \vdash_{\text{type}} t : k_1 \\ \Pi; \phi \vdash_{\text{lab}} k : k_2 \end{array}}{\Pi, \phi \vdash_{\text{type}} t k : k_1 \sqcup k_2}$	$\frac{\text{T-SEC-VAR} \quad \begin{array}{c} \Pi(\alpha) = \mathbf{type} k_1 \quad \Pi; \phi \vdash_{\text{lab}} k : k_2 \end{array}}{\Pi, \phi \vdash_{\text{type}} \alpha^k : k_1 \sqcup k_2}$	$\frac{\text{T-SEC-PROD} \quad \Pi, \phi \vdash_{\text{type}} s_i : k_i \quad i = 1, \dots, n}{\Pi, \phi \vdash_{\text{type}} s_1, \dots, s_n : \sqcup_{i=1, \dots, n} k_i}$	
$\frac{\text{T-SEC-SUB} \quad \begin{array}{c} \Pi, \phi \vdash_{\text{type}} s : k_1 \quad \phi \vdash k_1 \sqsubseteq k_2 \end{array}}{\Pi, \phi \vdash_{\text{type}} s : k_2}$	$\frac{\text{T-BASE-INT} \quad \Pi, \phi \vdash_{\text{type}} \mathbf{int} : \perp}{\Pi, \phi \vdash_{\text{type}} \mathbf{int} : \perp}$	$\frac{\text{T-BASE-SPTR} \quad \Pi, \phi \vdash_{\text{type}} s_i : k_i}{\Pi, \phi \vdash_{\text{type}} s_1 @ s_2 : k_1 \sqcup k_2}$	
$\frac{\text{T-BASE-PTR} \quad \begin{array}{c} \Pi; \phi \vdash_{\text{lab}} k : k_1 \quad \Pi, \phi \vdash_{\text{type}} s : k_2 \end{array}}{\Pi, \phi \vdash_{\text{type}} k \mapsto s : k_1 \sqcup k_2}$	$\frac{\text{T-BASE-EX-TY} \quad \begin{array}{c} \Pi; \phi \vdash_{\text{lab}} k : k_1 \\ \Pi[\alpha \mapsto \mathbf{type} k], \phi \vdash_{\text{type}} s : k_2 \end{array}}{\Pi, \phi \vdash_{\text{type}} \exists \alpha : \mathbf{type} k. s : k_1 \sqcup k_2}$		
$\frac{\text{T-BASE-EX-LEV} \quad \begin{array}{c} \Pi; \phi \vdash_{\text{lab}} k : k_1 \quad \kappa \notin \text{FV}(k_2) \\ \Pi[\kappa \mapsto \mathbf{level} k], \phi \vdash_{\text{type}} s : k_2 \end{array}}{\Pi, \phi \vdash_{\text{type}} \exists \kappa : \mathbf{type} k. s : k_1 \sqcup k_2}$	$\frac{\text{T-BASE-MU} \quad \begin{array}{c} \phi \vdash k_1 \sqsubseteq k' \\ \Pi[\alpha \mapsto \mathbf{type} k']; \phi \vdash_{\text{lab}} s : k' \\ \Pi; \phi \vdash_{\text{lab}} k_1 : k' \end{array}}{\Pi, \phi \vdash_{\text{type}} (\mu \alpha : \mathbf{type} k. s) : k'}$	$\frac{\text{T-BASE-SIZE} \quad \Pi, \phi \vdash_{\text{type}} s : k}{\Pi, \phi \vdash_{\text{type}} \mathbf{size}[s] : k}$	
$\frac{\text{T-BASE-SUB} \quad \begin{array}{c} \Pi, \phi \vdash_{\text{type}} s : k_1 \quad \phi \vdash k_1 \sqsubseteq k_2 \end{array}}{\Pi, \phi \vdash_{\text{type}} s : k_2}$	$\frac{\text{T-LAB-LIT} \quad \Pi; \phi \vdash_{\text{lab}} \ell : \perp}{\Pi; \phi \vdash_{\text{lab}} \ell : \perp}$	$\frac{\text{T-LAB-VAR} \quad \Pi(\kappa) = \mathbf{level} k}{\Pi; \phi \vdash_{\text{lab}} \kappa : k}$	$\frac{\text{T-LAB-JOIN} \quad \Pi; \phi \vdash_{\text{lab}} k_i : k'_i \quad i = 1, 2}{\Pi; \phi \vdash_{\text{lab}} k_1 \sqcup k_2 : k'_1 \sqcup k'_2}$
$\frac{\text{T-LAB-MEET} \quad \begin{array}{c} \Pi; \phi \vdash_{\text{lab}} k_i : k'_i \quad i = 1, 2 \end{array}}{\Pi; \phi \vdash_{\text{lab}} k_1 \sqcap k_2 : k'_1 \sqcap k'_2}$	$\frac{\text{T-LAB-SUB} \quad \begin{array}{c} \phi \vdash k_1 \sqsubseteq k_2 \quad \Pi; \phi \vdash_{\text{lab}} k : k_1 \end{array}}{\Pi; \phi \vdash_{\text{lab}} k : k_2}$		

Figure A.17: Well-formedness relation for security types, base types and security labels.

$$\begin{aligned}
& \Gamma, \Pi, \phi \models \langle c, M, P, h, q \rangle_v \triangleq \Gamma, \Pi, \phi \models (M, P, h) \\
& \frac{}{\Gamma, \Pi, \phi \models (\varepsilon, \varepsilon, h)} \quad \frac{\Gamma, \Pi, \phi \models (m, p, h) \quad \Gamma, \Pi, \phi \models (M, P, h)}{\Gamma, \Pi, \phi \models (m \cdot M, p \cdot P, h)} \\
& \Gamma, \Pi, \phi \models (m, p, h) \triangleq \\
& (\forall x. \Gamma(x) = s \wedge m(\delta(x) + \text{fp}(m)) = v \wedge \langle s, p \rangle \Downarrow_{\text{type}} \tau \implies \Gamma, \Pi, \phi \models^{M, P, h} v : \tau) \quad \wedge \\
& (\forall \alpha. \Pi(\alpha) = \mathbf{type} \, k \wedge \alpha \in \text{dom}(p) \implies \exists \tau. p(\alpha) = \tau) \quad \wedge \\
& (\forall \kappa. \Pi(\kappa) = \mathbf{level} \, k \wedge \kappa \in \text{dom}(p) \implies \exists \ell. p(\kappa) = \ell) \\
& \Gamma, \Pi, \phi \models^{M, P, h} v : \tau \triangleq \forall z. \Gamma, \Pi, \phi \models_z^{M, P, h} v : \tau
\end{aligned}$$

Figure A.18: Well-formedness of configurations.

$$\begin{aligned}
& \overline{\Gamma, \Pi, \phi \models_0^{M, P, h} v : \tau} \quad \overline{\Gamma, \Pi, \phi \models_z^{M, P, h} n : \mathbf{int} \, \ell} \quad \overline{\Gamma, \Pi, \phi \models_z^{M, P, h} n : \mathbf{size}[\tau] \, \ell} \\
& \frac{M = M_1 \cdot m \cdot M_2 \quad m(a) = v \implies \Gamma, \Pi, \phi \models_z^{M, P, h} v : \tau_2 \quad \Gamma, \Pi, \phi \models_z^{M, P, h} (a - |\tau_1|) : \tau_1}{\Gamma, \Pi, \phi \models_z^{M, P, h} a_v : (\tau_1 @ \tau_2) \, \ell} \\
& \frac{\Gamma, \Pi, \phi \models_z^{P, h} a_v : (\ell_p \mapsto \tau) \, \ell}{\Gamma, \Pi, \phi \models_z^{M, P, h} a_v : (\ell_p \mapsto \tau) \, \ell} \quad \frac{\langle s[\ell/\kappa], P \rangle \Downarrow_{\text{type}} \tau \quad \Gamma, \Pi, \phi \models_z^{M, P, h} v : \tau^{\ell'}}{\Gamma, \Pi, \phi \models^{M, P, h} (\ell, v) : (\exists \kappa : \mathbf{level}. s) \, \ell'[z]} \\
& \frac{\langle s[\tau/\alpha], P \rangle \Downarrow_{\text{type}} \tau' \quad \Gamma, \Pi, \phi \models_z^{M, P, h} v : \tau'^{\ell}}{\Gamma, \Pi, \phi \models_z^{M, P, h} (\tau, v) : (\exists \alpha : \mathbf{type}. s) \, \ell} \\
& \frac{\langle s[(\mu \alpha : \mathbf{type}. s) \ell/\alpha], P \rangle \Downarrow_{\text{type}} \tau \quad \Gamma, \Pi, \phi \models_{z-1}^{M, P, h} v : \tau}{\Gamma, \Pi, \phi \models_z^{M, P, h} v : (\mu \alpha : \mathbf{type}. s) \, \ell}
\end{aligned}$$

Figure A.19: Well-formedness of values.

EQ-VAL-0	EQ-INT-LOW	EQ-INT-HIGH
$\frac{}{p_1, p_2 \vdash v_1 =_{\mathcal{A}}^0 v_2 : \tau_1 \times \tau_2}$	$\frac{\ell \sqsubseteq \mathcal{A}}{p_1, p_2 \vdash n =_{\mathcal{A}}^z n : \mathbf{int} \ell \times \mathbf{int} \ell}$	$\frac{\ell_i \not\sqsubseteq \mathcal{A} \quad i = 1, 2}{p_1, p_2 \vdash n_1 =_{\mathcal{A}}^z n_2 : \mathbf{int} \ell_1 \times \mathbf{int} \ell_2}$
EQ-SPTR-LOW	EQ-SPTR-HIGH	
$\frac{\ell \sqsubseteq \mathcal{A}}{p_1, p_2 \vdash a_v =_{\mathcal{A}}^z a_v : (\tau_1 @ \tau_2) \ell \times (\tau_1 @ \tau_2) \ell}$	$\frac{\ell_i \not\sqsubseteq \mathcal{A}}{p_1, p_2 \vdash a_v =_{\mathcal{A}}^z a_v : (\tau_1^1 @ \tau_2^1) \ell_1 \times (\tau_1^2 @ \tau_2^2) \ell_2}$	
EQ-PTR-LOW	EQ-PTR-HIGH	
$\frac{\ell' \sqsubseteq \mathcal{A}}{p_1, p_2 \vdash a_v =_{\mathcal{A}}^z a_v : (\ell \mapsto \tau) \ell' \times (\ell \mapsto \tau) \ell'}$	$\frac{\ell'_i \not\sqsubseteq \mathcal{A}}{p_1, p_2 \vdash a_v =_{\mathcal{A}}^z a_v : (\ell_1 \mapsto \tau_1) \ell'_1 \times (\ell_2 \mapsto \tau_2) \ell'_2}$	
EQ-EX-TY-LOW	EQ-EX-TY-HIGH	
$\frac{\langle s'_i[\tau_i/\alpha], p_i \rangle \Downarrow_{\text{type}} \tau''_i \quad p_1, p_2 \vdash v_1 =_{\mathcal{A}}^z v_2 : \tau''_1 \times \tau''_2 \quad \ell \sqsubseteq \mathcal{A} \quad \tau'_i = (\exists \alpha : \mathbf{type} \ell'_i. s_i) \ell \quad i = 1, 2 \quad p_1, p_2 \vdash \tau_1 =_{\mathcal{A}}^z \tau_2 : \ell'_1 \times \ell'_2}{p_1, p_2 \vdash (\tau_1, v_1) =_{\mathcal{A}}^z (\tau_2, v_2) : \tau'_1 \times \tau'_2}$	$\frac{\ell_i \not\sqsubseteq \mathcal{A} \quad i = 1, 2 \quad \tau'_i = (\exists \alpha : \mathbf{type} \ell'_i. s_i) \ell_i \quad i = 1, 2 \quad p_1, p_2 \vdash \tau_1 =_{\mathcal{A}} \tau_2 : \ell'_1 \times \ell'_2}{p_1, p_2 \vdash (\tau_1, v_1) =_{\mathcal{A}}^z (\tau_2, v_2) : \tau'_1 \times \tau'_2}$	
EQ-EX-LEV-LOW	EQ-EX-LEV-HIGH	
$\frac{\langle s'_i[\ell_i/\kappa], p_i \rangle \Downarrow_{\text{type}} \tau''_i \quad p_1, p_2 \vdash v_1 =_{\mathcal{A}}^z v_2 : \tau''_1 \times \tau''_2 \quad \ell \sqsubseteq \mathcal{A} \quad \tau'_i = (\exists \kappa : \mathbf{level} \ell'_i. s_i) \ell \quad i = 1, 2 \quad \ell_1 =_{\mathcal{A}}^z \ell_2 : \ell'_1 \times \ell'_2}{p_1, p_2 \vdash (\ell_1, v_1) =_{\mathcal{A}}^z (\ell_2, v_2) : \tau'_1 \times \tau'_2}$	$\frac{\ell''_i \not\sqsubseteq \mathcal{A} \quad i = 1, 2 \quad \tau'_i = (\exists \kappa : \mathbf{level} \ell'_i. s_i) \ell''_i \quad i = 1, 2 \quad \ell_1 =_{\mathcal{A}} \ell_2 : \ell'_1 \times \ell'_2}{p_1, p_2 \vdash (\ell_1, v_1) =_{\mathcal{A}}^z (\ell_2, v_2) : \tau'_1 \times \tau'_2}$	
EQ-REC		
$\frac{\ell_1 =_{\mathcal{A}} \ell_2 : \ell_1 \times \ell_2 \quad p_1, p_2 \vdash \ell'_1 =_{\mathcal{A}} \ell'_2 : \ell'_1 \times \ell'_2 \quad \langle s_i[(\mu \alpha : \mathbf{type} \ell_i. s_i) \ell'/\alpha], p_i \rangle \Downarrow_{\text{type}} \tau_i \quad p_1, p_2 \vdash v_1 =_{\mathcal{A}}^{z-1} v_2 : \tau_1 \times \tau_2}{p_1, p_2 \vdash v_1 =_{\mathcal{A}}^z v_2 : (\mu \alpha : \mathbf{type} \ell_1. s_1) \ell'_1 \times (\mu \alpha : \mathbf{type} \ell_2. s_2) \ell'_2}$		

Figure A.20: \mathcal{A} -equivalence for values.

$\frac{\ell' \sqsubseteq \mathcal{A}}{p_1, p_2 \vdash \pi \ell =_{\mathcal{A}} \pi \ell : \ell' \times \ell'}$	$\frac{\ell \sqsubseteq \mathcal{A}}{p_1, p_2 \vdash \bar{\tau} =_{\mathcal{A}} \bar{\tau} : \ell \times \ell}$	$\frac{\ell \sqsubseteq \mathcal{A}}{p_1, p_2 \vdash \underline{\ell} =_{\mathcal{A}} \underline{\ell} : \ell \times \ell}$
$\frac{\ell_i \not\sqsubseteq \mathcal{A} \quad i = 1, 2}{p_1, p_2 \vdash \tau_1 =_{\mathcal{A}} \tau_2 : \ell_1 \times \ell_2}$	$\frac{\ell' \sqsubseteq \mathcal{A}}{\ell =_{\mathcal{A}} \ell : \ell' \times \ell'}$	$\frac{\ell'_i \not\sqsubseteq \mathcal{A} \quad i = 1, 2}{\ell_1 =_{\mathcal{A}} \ell_2 : \ell'_1 \times \ell'_2}$

Figure A.21: \mathcal{A} -equivalence for security types and levels.

$$\begin{array}{c}
\frac{}{\Gamma \mid p_1, p_2 \vdash \varepsilon =_{\mathcal{A}} \varepsilon} \quad \frac{\langle \Gamma(x), p_i \rangle \Downarrow_{\text{type}} \tau_i \quad i = 1, 2 \quad p_1, p_2 \vdash v_1 =_{\mathcal{A}} v_2 : \tau_1 \times \tau_2}{\Gamma \mid p_1, p_2 \vdash \text{asgn}(x \leftarrow v_1, q) =_{\mathcal{A}} \text{asgn}(x \leftarrow v_2, q)} \\
\\
\frac{\langle \Gamma(x), p_i \rangle \Downarrow_{\text{type}} \tau_i \quad i = 1, 2 \quad p_1, p_2 \vdash v_1 =_{\mathcal{A}} v_2 : \tau_1 \times \tau_2}{\Gamma \mid p_1, p_2 \vdash \text{rd}(x \leftarrow v_1, q) =_{\mathcal{A}} \text{rd}(x \leftarrow v_2, q)} \\
\\
\frac{\ell_1 =_{\mathcal{A}} \ell_2 : \ell_1 \times \ell_2 \quad p_1, p_2 \vdash v_1 =_{\mathcal{A}} v_2 : \tau_1 \times \tau_2}{\Gamma \mid p_1, p_2 \vdash \text{unp}(\ell_1, y : \tau_1 \leftarrow v_1, q) =_{\mathcal{A}} \text{unp}(\ell_2, y : \tau_2 \leftarrow v_2, q)} \\
\\
\frac{p_1, p_2 \vdash v_1 =_{\mathcal{A}} v_2 : \tau_1 \times \tau_2}{\Gamma \mid p_1, p_2 \vdash \text{let}(x : \tau_1 \leftarrow v_1, q) =_{\mathcal{A}} \text{let}(x : \tau_2 \leftarrow v_2, q)} \quad \frac{\Gamma \mid p_1, p_2 \vdash \textcolor{red}{ev}_1 =_{\mathcal{A}} \textcolor{red}{ev}_2}{\Gamma \mid p_1, p_2 \vdash \textcolor{red}{ev}_1 =_{\mathcal{A}} \textcolor{red}{ev}_2}
\end{array}$$

Figure A.22: \mathcal{A} -equivalence for events.

$$\begin{array}{ccc}
\text{Ev-OBS-RD} & \text{Ev-OBS-UNP-1} & \text{Ev-OBS-UNP-2} \\
\frac{\langle \Gamma(x), P \rangle \Downarrow_{\text{type}} \tau \quad \tau \sqsubseteq \mathcal{A}}{\Gamma, p \vdash \text{rd}(x \leftarrow v, q) \sqsubseteq \mathcal{A}} & \frac{\ell \sqsubseteq \mathcal{A}}{\Gamma, p \vdash \text{unp}(\ell, y : \tau \leftarrow v, q) \sqsubseteq \mathcal{A}} & \frac{\tau \sqsubseteq \mathcal{A}}{\Gamma, p \vdash \text{unp}(\ell, y : \tau \leftarrow v, q) \sqsubseteq \mathcal{A}} \\
\\
\text{Ev-OBS-DECL} & & \text{Ev-OBS-INST} \\
\frac{\tau \sqsubseteq \mathcal{A}}{\Gamma, p \vdash \text{let}(x : \tau \leftarrow v, q) \sqsubseteq \mathcal{A}} & & \frac{\Gamma, P \vdash \textcolor{red}{ev} \sqsubseteq \mathcal{A}}{\Gamma, P \vdash \textcolor{red}{ev} \sqsubseteq \mathcal{A}}
\end{array}$$

Figure A.23: \mathcal{A} -observability for events.

Given two traces t_1, t_2 we say they are \mathcal{A} -equivalent, written $t_1 =_{\mathcal{A}} t_2$, when $(\lfloor t_1 \rfloor_{\mathcal{A}})_i =_{\mathcal{A}} (\lfloor t_2 \rfloor_{\mathcal{A}})_i$ for $i = 1, \dots, n$ where $n = |\lfloor t_1 \rfloor_{\mathcal{A}}| = |\lfloor t_2 \rfloor_{\mathcal{A}}|$.

We write $\langle c, m, P, h, q \rangle_{\mathcal{V}} \xrightarrow{\mathcal{A} \rightsquigarrow t}^* \langle c', m', P', q' \rangle_{\mathcal{V}}$, when $\langle c, m, P, h, q \rangle_{\mathcal{V}} \xrightarrow{t'}^* \langle c', m', P', q' \rangle_{\mathcal{V}}$, and $t =_{\mathcal{A}} t'$.

Finally, we define a notion of \mathcal{A} -equivalent stack frames:

$$\frac{\forall x, i. \langle \Gamma(x), p_i \rangle \Downarrow_{\text{type}}^{\ell_i} \tau_i \wedge m_i(\delta(x) + \text{fp}(m_i)) = v_i \implies p_1, p_2 \vdash v_1 =_{\mathcal{A}} v_2 : \tau_1^{\ell_1} \times \tau_2^{\ell_2}}{\Gamma \vdash (p_1, m_1) =_{\mathcal{A}} (p_2, m_2)}$$

$$\frac{pc \sqsubseteq \mathcal{A} \quad \Gamma \vdash (p_1, m_1) =_{\mathcal{A}} (p_2, m_2) \quad \Gamma \vdash (P_1, M_1)_{\overline{pc_1}} =_{\mathcal{A}} (P_2, M_2)_{\overline{pc_2}}}{\Gamma \vdash (p_1 \cdot P_1, m_1 \cdot M_1)_{pc \cdot \overline{pc_1}} =_{\mathcal{A}} (p_2 \cdot P_2, m_2 \cdot M_2)_{pc \cdot \overline{pc_2}}}$$

$$\frac{pc' \not\sqsubseteq \mathcal{A} \quad \Gamma \vdash (P_1, M_1)_{\overline{pc_1}} =_{\mathcal{A}} (P_2, M_2)_{\overline{pc_2}}}{\Gamma \vdash (P_1, M_1)_{\overline{pc_1}} =_{\mathcal{A}} (p_2 \cdot P_2, m_2 \cdot M_2)_{pc' \cdot \overline{pc_2}}}$$

$$\frac{pc' \not\sqsubseteq \mathcal{A} \quad \Gamma \vdash (P_1, M_1)_{\overline{pc_1}} =_{\mathcal{A}} (P_2, M_2)_{\overline{pc_2}}}{\Gamma \vdash (p_1 \cdot P_1, m_1 \cdot M_1)_{pc' \cdot \overline{pc_1}} =_{\mathcal{A}} (P_2, M_2)_{\overline{pc_2}}} \quad \Gamma \vdash (\varepsilon, \varepsilon)_{\varepsilon} =_{\mathcal{A}} (\varepsilon, \varepsilon)_{\varepsilon}$$

A.6 Required Properties of the Instantiation Languages

Obviously, not all languages are valid instantiation languages. For instance, a *bad* instantiation language could leak sensitive information through a direct/explicit flow (i.e., an assignment $\text{low} := h$). To filter out these bad instantiation languages we require the following three properties about the semantics of the instantiation language.

First, the execution must be safe with respect to the well-formedness definitions (i.e., Definition A.18).

Property A.6.1 (Single-run reduction safety). *If*

$$\Gamma, \Pi, \phi, pc, fr \vdash \langle c, M, h, P, q \rangle_{\nu} \rightarrow \langle c', M', h', P', q' \rangle_{\nu'} : \Gamma', \Pi', \phi', pc', fr'$$

it holds that $\Gamma \subseteq \Gamma', \Pi \subseteq \Pi', \nu \leq \nu'$, and $\phi' \implies \phi$. Finally it holds that $\Gamma', \Pi', \phi' \models \langle c', M', h', P', q' \rangle_{\nu'}, P' \models \phi'$ and $\Gamma', \Pi', \phi', pc', fr' \vdash c'$.

Second, the execution semantics of the instantiation language must satisfy a single-step noninterference property. This property guarantees that \mathcal{A} -equivalence of memories and traces and preserved during execution.

Property A.6.2 (Single-step noninterference). *If $\Gamma \vdash (P_1, M_1) =_{\mathcal{A}} (P_2, M_2)$ and $\phi \vdash pc \sqsubseteq \mathcal{A}$ and*

$$\Gamma, \Pi, \phi, pc, fr \vdash \langle c, M_i, h_i, P_i, q \rangle_{\nu} \xrightarrow{\widetilde{ev}_i} \langle c'_i, M'_i, h'_i, P'_i, q' \rangle_{\nu'_i} : \Gamma'_i, \Pi'_i, \phi'_i, pc'_i, fr'_i$$

for $i = 1, 2$ then $\Gamma'_1 = \Gamma'_2, \Pi'_1 = \Pi'_2, \Gamma'_1 \vdash (P'_1, M'_1) =_{\mathcal{A}} (P'_2, M'_2), q'_1 = q'_2$ and $\Gamma'_1 \mid P'_1, P'_2 \vdash \widetilde{ev}_1 =_{\mathcal{A}} \widetilde{ev}_2$.

Finally, the semantics must satisfy a confinement [66] property.

Property A.6.3 (Confinement). *If $\phi \vdash pc \not\sqsubseteq \mathcal{A}$ and*

$$\Gamma, \Pi, \phi, pc, fr \vdash \langle c, M, h, P, q \rangle_{\vee} \xrightarrow{\tilde{ev}} \langle c', M', h', P', q' \rangle_{\vee}, : \Gamma', \Pi', \phi', pc', fr'$$

then $\Gamma' \vdash (P, M) =_{\mathcal{A}} (P', M')$ and $\phi', P' \vdash ev \not\sqsubseteq \mathcal{A}$.

A.7 Noninterference for Zee

We write $\langle c, m, P, h, q \rangle_{\vee} \rightarrow^*$ when there exists a configuration C with command stop such that $\langle c, m, P, h, q \rangle_{\vee} \rightarrow^* C$, and we use similar notation for the event semantics.

Given a trace t , the attacker's knowledge $k_{\mathcal{A}}(c, t)$ is the set of stacks such that the execution of c produces an \mathcal{A} -equivalent trace:

$$k_{\mathcal{A}}(c, t) = \left\{ (M, P) \mid \langle c, M, P \rangle \xrightarrow{\mathcal{A} \rightsquigarrow t}^* \right\}$$

Note that a larger knowledge set corresponds to an attacker obtaining less information, and smaller sets correspond to a more precise knowledge. To define termination-insensitive noninterference, we define $\mathbb{M}_{\mathcal{A}, \Gamma}(c, M_1, P_1)$ as the set containing \mathcal{A} -equivalent stacks that generates a terminating execution of c .

$$\mathbb{M}_{\mathcal{A}, \Gamma}(c, M_1, P_1) = \left\{ (M_2, P_2) \mid \begin{array}{l} \Gamma \vdash (P_1, M_1) =_{\mathcal{A}} (P_2, M_2) \\ \wedge \langle c, M_2, P_2 \rangle \rightarrow^* \end{array} \right\}$$

Using attacker knowledge and the set of terminating memories, we can define the noninterference policy as follows.

Definition A.7.1 (Termination-insensitive noninterference). *A program c satisfies termination-insensitive interference wrt. typing environment Γ if, for all M and P it holds that $\langle c, M, P \rangle \xrightarrow{\mathcal{A} \rightsquigarrow t}^*$ implies $k_{\mathcal{A}}(c, t) \supseteq \mathbb{M}_{\mathcal{A}, \Gamma}(c, M, P)$.*

The soundness of our type-based enforcement mechanism is now formalized by the following theorem:

Theorem A.7.1 (Soundness). *If $\Gamma \vdash c$ then c satisfies Definition A.7.1*

To prove this, we define a semantics that *bridges* over high events as follows.

$$\begin{array}{c}
 \text{BRIDGE-STOP} \\
 \frac{\tilde{e}\tilde{v} \not\sqsubseteq A \quad \text{stop}(C')}{\Gamma, \Pi, \phi, pc, fr \vdash C \xrightarrow{\tilde{e}\tilde{v}} C' : \Gamma', \Pi', \phi', pc', fr'} \\
 \frac{\Gamma, \Pi, \phi, pc, fr \vdash C \xrightarrow{\tilde{e}\tilde{v}} C' : \Gamma', \Pi', \phi', pc', fr'}{\Gamma, \Pi, \phi, pc, fr \vdash C \curvearrow_0^{\tilde{e}\tilde{v}} C' : \Gamma', \Pi', \phi', pc', fr'} \\
 \\
 \text{BRIDGE-LOW} \\
 \frac{\tilde{e}\tilde{v} \sqsubseteq A}{\Gamma, \Pi, \phi, pc, fr \vdash C \xrightarrow{\tilde{e}\tilde{v}} C' : \Gamma', \Pi', \phi', pc', fr'} \\
 \frac{\Gamma, \Pi, \phi, pc, fr \vdash C \xrightarrow{\tilde{e}\tilde{v}} C' : \Gamma', \Pi', \phi', pc', fr'}{\Gamma, \Pi, \phi, pc, fr \vdash C \curvearrow_0^{\tilde{e}\tilde{v}} C' : \Gamma', \Pi', \phi', pc', fr'} \\
 \\
 \text{BRIDGE-TRANS} \\
 \frac{\tilde{e}\tilde{v} \not\sqsubseteq A \quad \Gamma, \Pi, \phi, pc, fr \vdash C \xrightarrow{\tilde{e}\tilde{v}} C' : \Gamma', \Pi', \phi', pc', fr' \quad \Gamma', \Pi', \phi', pc', fr' \vdash C' \curvearrow_n^{\tilde{e}\tilde{v}'} C'' : \Gamma'', \Pi'', \phi'', pc'', fr''}{\Gamma, \Pi, \phi, pc, fr \vdash C \curvearrow_{1+n}^{\tilde{e}\tilde{v}'} C'' : \Gamma'', \Pi'', \phi'', pc'', fr''}
 \end{array}$$

We write ℓs for a list of levels $\bar{\ell}$.

Lemma A.7.2 (Confinement). *If $\Gamma, \Pi, \phi, pc, fr \vdash c$ and $\ell_{pc} \not\sqsubseteq A$ and*

$$\Gamma, \Pi, \Gamma', \Pi' \vdash \langle c, M, h, P, q \mid \ell_{pc} \cdot \ell s_{pc} \rangle_{\vee} \curvearrow_n^{\tilde{e}\tilde{v}} \langle c', M', h', P', q' \mid \ell s' \rangle_{\vee},$$

then $\Gamma \vdash (P, M) =_{\mathcal{A}} (P', M')$ and $\tilde{e}\tilde{v} \not\sqsubseteq A$.

Proof. Induction in n , using Property A.6.3 the case of $c = \mathbf{c}$. □

We write pcs for a list of labels (i.e., $pcs = \overline{pc}$). The soundness theorem is now proven by proving a stronger theorem:

Theorem A.7.3. *If*

1. $\Gamma, \Pi, \phi, pc, fr \vdash c$
2. $\phi \vdash pc \sqsubseteq A$
3. $\Gamma \vdash (P_1, M_1)_{pcs_1} =_{\mathcal{A}} (P_2, M_2)_{pcs_2}$
4. $\Gamma, \Pi, \phi, pc, fr \vdash \langle c, M_1, P_1, h_1, q \mid pcs_1 \rangle_{\vee} \curvearrow_{n_1}^{\tilde{e}\tilde{v}_1} \langle c'_1, M'_1, P'_1, h'_1, q'_1 \mid pc1'_1 \rangle_{\vee} :$
 $\Gamma'_1, \Pi'_1, \phi'_1, pc'_1, fr'_1$
5. $\Gamma, \Pi, \phi, pc, fr \vdash \langle c, M_2, P_2, h_2, q_2 \mid pcs_2 \rangle_{\vee} \curvearrow_{n_2}^{\tilde{e}\tilde{v}_2} \langle c'_2, M'_2, P'_2, h'_2, q'_2 \mid pcs'_2 \rangle_{\vee} :$
 $\Gamma'_2, \Pi'_2, \phi'_2, pc'_2, fr'_2$

then

1. $\Gamma'_1 = \Gamma'_2$ and $\Pi'_1 = \Pi'_2$ (call these Γ' and Π')
2. $\widetilde{ev}_1 =_{\mathcal{A}} \widetilde{ev}_2$
3. $\Gamma' \vdash (P'_1, M'_1)_{pcs'_1} =_{\mathcal{A}} (P'_2, M'_2)_{pcs'_2}$
4. $q'_1 = q'_2$
5. $c'_1 = c'_2$

Proof. Induction in n_1 . For $n = 0$, assumption 4 must be either an instance of BRIDGE-STOP, or BRIDGE-LOW. Proceed by induction in c . For $n > 0$, assumption 4 must be an instance of BRIDGE-TRANS. Proceed by induction in c . For the case of $c = \text{at } k \text{ with bound } e \text{ do } c'$ we apply Lemma A.7.2 when $\langle k, P \rangle \Downarrow_{\text{lab}} \ell$ and $\ell \not\sqsubseteq \mathcal{A}$. For $c = \text{c}$ the result follows by case analysis on Hypothesis 5 and Property A.6.2. \square

Finally, we can relate the traditional semantics with the bridge step semantics using the following Lemma.

Lemma A.7.4 (Adequacy of bridge steps). *If $C \xrightarrow{t}^n C''$ then either $k = 0$ or there exists C' , ev and t' such that $t = ev \cdot t'$ and $C \curvearrowright_k^{ev} C' \xrightarrow{t'}^{n-k-1} C''$.*

Proof. Induction in n . \square

Theorem A.7.1 now follows by picking $pc = \perp$ in Theorem A.7.3 and repeated applications of Lemma A.7.4.

Appendix B

Programming with Flow-Limited Authorization: Coarser is Better

B.1 Syntax

The syntax of FLAM principals is shown in Figure B.1. The meta-variable p range over FLAM principals, which include the bottom principal, the top principal, named principals, the conjunction and disjunction of two principals, basic projections of principals for obtaining the integrity or confidentiality part of a principal, and owned principals.

The syntax of Flamio is shown in Figure B.2. The meta-variable v ranges over values, which include boolean literals, the unit value, principals, addresses, variable names, abstractions living on the machine belonging to node n , pairs, lists, and two special constructs used during evaluation.

The meta-variable e ranges over expressions, which include standard constructs such as values, applications, projections, conditionals, elimination of lists, fixpoints, monadic operations, allocation of, reading from, and writing to references. The language embeds LIO like constructs such as labeling, unlabeling, mitigation of label creep, operations for obtaining the current floating label, the clearance, and the current strategy, obtaining the label of an expression, evaluating an expression in a new delegation scope, dynam-

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

Figure B.1: Syntax of FLAM

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

Figure B.2: The Flamio language

ically checking authority, evaluating an expression with a new strategy, and modifying the trust relationship between principals.

The meta-variable τ ranges over types and include the type of booleans, the unit type, functions types, lists types, products, the type of principals, and the type of labeled values, the monadic type that evaluates to a value of type τ , and finally the type of references on node n pointing to values of type τ .

Figure B.3 describes the syntax of evaluation contexts, ranged over by the meta-variable E . The syntax corresponds to a call-by-name evaluation semantics, similar to that of LIO.

$$\begin{aligned}
 E ::= & [\cdot] \mid E e \mid \pi_i E \mid \text{if } E \text{ then } e \text{ else } e \mid \text{case } E \text{ of } e e \mid \text{return } E \mid E \gg e \\
 & \mid \text{new } E e \mid !E \mid E := e \mid \text{label } E e \mid \text{unlabel } E \mid \text{toLabeled } E e \mid \text{labelOf } E \\
 & \mid \text{withStrategy } E e \mid \text{assume } (p \succcurlyeq E) @ q \mid \text{assume } (E \succcurlyeq e) @ q \mid \text{assume } (e \succcurlyeq e) @ E \\
 & \mid E \succcurlyeq e \mid p \succcurlyeq E \mid E; \bar{e}
 \end{aligned}$$

Figure B.3: Evaluation context for Flamio.

B.2 Local Semantics

The semantics is split into several layers: a reduction relation $es \rightarrow es'$ for pure expressions is given in Figure B.4. This forms the bottom layer of the semantics.

$$\begin{array}{l}
\text{return } v \rightarrow (v)^{\text{LIO}} \quad (v)^{\text{LIO}} \gg= e \rightarrow e \ v \quad \text{fix } e \rightarrow e \ (\text{fix } e) \\
\pi_i (e_1, e_2) \rightarrow e_i \quad \text{if } b \text{ then } e_{\text{true}} \text{ else } e_{\text{false}} \rightarrow e_b \quad \text{case nil of } e_1 \ e_2 \rightarrow e_1 \\
\text{case } (e_{\text{hd}} :: e_{\text{tl}}) \text{ of } e_1 \ e_2 \rightarrow e_2 \ e_{\text{hd}} \ e_{\text{tl}} \quad \text{labelOf } (e @ p) \rightarrow p \quad \frac{e \rightarrow e'}{E[e] \rightarrow E[e']}
\end{array}$$

Figure B.4: Pure reductions for Flamio.

$$\begin{array}{l}
\text{E-LIFT-PURE} \quad \frac{es \rightarrow es'}{n; \Sigma \vdash \langle \phi \mid es \rangle \rightarrow \langle \phi \mid e's \rangle : \Sigma(n)} \\
\text{E-CTX} \quad \frac{n; \Sigma \vdash \langle \phi \mid e \rangle \rightarrow \langle \phi' \mid e' \rangle : \sigma}{n; \Sigma \vdash \langle \phi \mid E[e] \rangle \rightarrow \langle \phi' \mid E[e'] \rangle : \sigma} \\
\text{E-GET-LABEL} \quad \frac{\ell = \Sigma(n).lbl}{n; \Sigma \vdash \langle \phi \mid \text{getLabel} \rangle \rightarrow \langle \phi \mid \text{return } \ell \rangle : \Sigma(n)} \\
\text{E-GET-CLEARANCE} \quad \frac{}{n; \Sigma \vdash \langle \phi \mid \text{getClearance} \rangle \rightarrow \langle \phi \mid \text{return } n^{\rightarrow} \rangle : \Sigma(n)} \\
\text{E-APP} \quad \frac{\begin{array}{l} n; \Sigma \vdash \Sigma(n).lbl \sqsubseteq p \sqsubseteq n^{\rightarrow} : \ell \\ e' = e_1[e_2/x] \gg= \lambda_{\tau}^n x. x @ p \\ n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid (\lambda_{\tau @ p}^n x. e_1) \ e_2 \rangle \rightarrow \langle \phi \mid e' \rangle : \sigma} \\
\text{E-NEW} \quad \frac{\begin{array}{l} a \notin \text{dom}(\phi) \quad n; \Sigma \vdash \Sigma(n).lbl \sqsubseteq p \sqsubseteq n^{\rightarrow} : \ell \\ n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell \sqsubseteq n^{\rightarrow} \quad \phi' = \phi [a \mapsto e @ p] \\ \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid \text{new } p \ e \rangle \rightarrow \langle \phi' \mid \text{return } a \rangle : \sigma} \\
\text{E-READ} \quad \frac{\begin{array}{l} \phi(a) = e @ p \quad n; \Sigma \vdash \Sigma(n).lbl \sqcup p \sqsubseteq n^{\rightarrow} : \ell \\ n; \Sigma \vdash \Sigma(n).lbl \sqcup p \sqcup \ell \sqsubseteq n^{\rightarrow} \\ \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup p \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid ! a \rangle \rightarrow \langle \phi \mid \text{return } e \rangle : \sigma} \\
\text{E-WRITE} \quad \frac{\begin{array}{l} \phi(a) = e @ p \quad n; \Sigma \vdash \Sigma(n).lbl \sqsubseteq p \sqsubseteq n^{\rightarrow} : \ell \\ n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell \sqsubseteq n^{\rightarrow} \quad \phi' = \phi [a \mapsto e' @ p] \\ \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup \ell] \end{array}}{n; \Sigma \vdash \langle \phi \mid a := e' \rangle \rightarrow \langle \phi' \mid \text{return } () \rangle : \sigma}
\end{array}$$

Figure B.5: Monadic reductions for Flamio.

At the next layer in the semantics is the monadic local reduction steps $n; \Sigma \vdash \langle \phi \mid es \rangle \rightarrow \langle \phi' \mid es' \rangle : \sigma$ performed by a node n . The semantics of monadic local reduction steps is given in Figure B.5, B.6 and B.7.

$$\begin{array}{c}
\text{E-LABEL} \\
\frac{n; \Sigma \vdash \Sigma(n).lbl \sqsubseteq p \sqsubseteq n^{\rightarrow} : \ell \quad n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell \sqsubseteq n^{\rightarrow} \quad \sigma = \Sigma(n) [\Sigma(n).lbl \mapsto \Sigma(n).lbl \sqcup \ell]}{n; \Sigma \vdash \langle \phi \mid \text{label } p \ e \rangle \rightarrow \langle \phi \mid \text{return } (e @ p) \rangle : \sigma} \\
\\
\text{E-UNLABEL} \\
\frac{n; \Sigma \vdash \Sigma(n).lbl \sqcup p \sqsubseteq n^{\rightarrow} : \ell \quad n; \Sigma \vdash \Sigma(n).lbl \sqcup p \sqcup \ell \sqsubseteq n^{\rightarrow} \quad \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup p \sqcup \ell]}{n; \Sigma \vdash \langle \phi \mid \text{unlabel } (e @ p) \rangle \rightarrow \langle \phi \mid \text{return } e \rangle : \sigma} \\
\\
\text{E-TOLABELLED-1} \\
\frac{q = \Sigma(n).lbl \quad e' = \text{toLabeled}_q \ p \ e}{n; \Sigma \vdash \langle \phi \mid \text{toLabeled } p \ e \rangle \rightarrow \langle \phi \mid e' \rangle : \Sigma(n)} \\
\\
\text{E-TOLABELLED-2} \\
\frac{\sigma = \Sigma(n) [lbl \mapsto p]}{n; \Sigma \vdash \langle \phi \mid \text{toLabeled}_p \ q \ v \rangle \rightarrow \langle \phi \mid \text{label } q \ v \rangle : \sigma}
\end{array}$$

Figure B.6: Monadic reductions for Flamio (cont).

$$\begin{array}{c}
\text{E-ACTS-FOR-TRUE} \\
\frac{n; \Sigma \vdash p \succcurlyeq q : \ell \quad n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell \sqsubseteq n^{\rightarrow} \quad \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup \ell]}{n; \Sigma \vdash \langle \phi \mid p \succcurlyeq q \rangle \rightarrow \langle \phi \mid \text{return true} \rangle : \sigma} \\
\\
\text{E-ACTS-FOR-FALSE} \\
\frac{n; \Sigma \vdash p \succcurlyeq q : \text{fail} \quad \ell = n^{\rightarrow} \sqcap \bigsqcup_{s \in \Sigma(n).strat} s \quad \sigma = \Sigma(n) [lbl \mapsto \Sigma(n).lbl \sqcup \ell]}{n; \Sigma \vdash \langle \phi \mid p \succcurlyeq q \rangle \rightarrow \langle \phi \mid \text{return false} \rangle : \sigma} \\
\\
\text{E-WITH-SCOPE} \\
\frac{e' = \text{resetScope}_{\Sigma(n).\Delta}(e)}{n; \Sigma \vdash \langle \phi \mid \text{withScope } e \rangle \rightarrow \langle \phi \mid e' \rangle : \Sigma(n)} \\
\\
\text{E-RESET-SCOPE} \\
\frac{\sigma = \Sigma(n) [\Delta \mapsto \Delta']}{n; \Sigma \vdash \langle \phi \mid \text{resetScope}_{\Delta'}(v) \rangle \rightarrow \langle \phi \mid v \rangle : \sigma} \\
\\
\text{E-WITH-STRATEGY} \\
\frac{e' = \text{resetStrategy}_{\Sigma.strat}(e) \quad \sigma = \Sigma(n) [strat \mapsto \vec{p}]}{n; \Sigma \vdash \langle \phi \mid \text{withStrategy } \vec{p} \ e \rangle \rightarrow \langle \phi \mid e' \rangle : \sigma} \\
\\
\text{E-RESET-STRATEGY} \\
\frac{\sigma = \Sigma(n) [strat \mapsto \vec{p}]}{n; \Sigma \vdash \langle \phi \mid \text{resetStrategy}_{\vec{p}}(v) \rangle \rightarrow \langle \phi \mid v \rangle : \sigma} \\
\\
\text{E-ASSUME} \\
\frac{n; \Sigma \vdash \Sigma(n).lbl \sqsubseteq r : \ell_1 \quad n; \Sigma \vdash \Sigma(n).lbl \succcurlyeq \nabla(q) : \ell_2 \quad n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell_1 \sqcup \ell_2 \sqsubseteq n^{\rightarrow} \quad \sigma = \Sigma(n) [\Delta \mapsto (p, q) @ r :: \Sigma.\Delta], lbl \mapsto \Sigma(n).lbl \sqcup \ell_1 \sqcup \ell_2]}{n; \Sigma \vdash \langle \phi \mid \text{assume } (p \succcurlyeq q) @ r \rangle \rightarrow \langle \phi \mid \text{return } () \rangle : \sigma}
\end{array}$$

Figure B.7: Monadic reductions for Flamio (cont).

$$\begin{array}{c}
\text{ACTS-FOR-CONS-1} \\
\frac{\mathcal{H}; n; \Sigma \vdash_{s \sqcap n \rightarrow} p \succcurlyeq q : \ell}{\mathcal{H}; n; \Sigma \vdash_{s::ss} p \succcurlyeq q : \ell}
\end{array}
\quad
\begin{array}{c}
\text{ACTS-FOR-CONS-2} \\
\frac{\mathcal{H}; n; \Sigma \vdash_{s \sqcap n \rightarrow} p \succcurlyeq q : \text{fail} \quad \mathcal{H}; n; \Sigma \vdash_{ss} p \succcurlyeq q : \ell^?}{\mathcal{H}; n; \Sigma \vdash_{s::ss} p \succcurlyeq q : \ell^?}
\end{array}
\quad
\begin{array}{c}
\text{ACTS-FOR-NIL} \\
\frac{}{C \vdash_{\text{nil}} p \succcurlyeq q : \text{fail}}
\end{array}$$

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

Figure B.8: Top-level judgment for proving authorization queries in Flamio. The meta-variable C abbreviates $\mathcal{H}; n; \Sigma$.

B.3 Semantics of Authorization

Like the small-step semantics for evaluating expressions, the semantics of resolving authorization queries is split into layers. The top layer using ACTS-FOR to try each principal in the strategy when proving a trust relationship between two principals p and q .

The judgment for deriving trust given a specific strategy principal s is written as $\mathcal{H}; n; \Sigma \vdash_s p \succcurlyeq q : \ell$, and is given in Figure B.9. To facilitate inductive proofs, the judgment is indexed by a natural number $k \in \mathbb{N}$ representing the depth of a derivation.

As FLAM unifies principals and labels, we can define the flow relation in terms of the trust relation.

$$\frac{\mathcal{H}; n; \Sigma \vdash_s^k q^{\rightarrow} \wedge q^{\leftarrow} \succcurlyeq p^{\rightarrow} \wedge q^{\leftarrow} : \ell}{\mathcal{H}; n; \Sigma \vdash_s^k p \sqsubseteq q : \ell}$$

Figure B.10: Information-flow judgment in Flamio.

The trust judgment satisfies the following property, which intuitively shows that any derivation must have a proof of depth at least 1.

Lemma B.3.1. *If $\mathcal{H}; n; \Sigma \vdash_s^k p \succcurlyeq q : \ell$ then $k > 0$.*

Proof. Induction in k . □

Lemma B.3.2. *If $\mathcal{H}; n; \Sigma \vdash p \succcurlyeq q : \ell$ then $\Sigma; n \vdash \ell \sqsubseteq \bigsqcup_{s \in \Sigma(n).\text{strat}} s$.*

BOT $\frac{k > 0}{C \vdash_s^k p \succcurlyeq \perp : \perp^\Xi}$	TOP $\frac{k > 0}{C \vdash_s^k \top \succcurlyeq p : \perp^\Xi}$	REFL $\frac{k > 0}{C \vdash_s^k p \succcurlyeq p : \perp^\Xi}$	ASSUMP $\frac{k > 0 \quad (p \succcurlyeq q) \in \mathcal{H}}{C \vdash_s^k p \succcurlyeq q : \perp^\Xi}$
PROJ $\frac{C \vdash_s^k p \succcurlyeq q : \ell}{C \vdash_s^{k+1} p^\pi \succcurlyeq q^\pi : \ell}$	PROJR $\frac{k > 0}{C \vdash_s^k p \succcurlyeq p^\pi : \perp^\Xi}$	OWN-1 $\frac{C \vdash_s^k o \succcurlyeq o' : \ell_1 \quad C \vdash_s^k p \succcurlyeq p' : \ell_2}{C \vdash_s^{k+1} o:p \succcurlyeq o':p' : \ell_1 \sqcup \ell_2}$	
OWN-2 $\frac{C \vdash_s^k o \succcurlyeq o' : \ell_1 \quad C \vdash_s^k p \succcurlyeq o':p' : \ell_2}{C \vdash_s^{k+1} o:p \succcurlyeq o':p' : \ell_1 \sqcup \ell_2}$	CONJ-L $\frac{j \in \{1,2\} \quad C \vdash_s^k p_j \succcurlyeq p : \ell}{C \vdash_s^{k+1} p_1 \wedge p_2 \succcurlyeq p : \ell}$	CONJ-R $\frac{C \vdash_s^k p \succcurlyeq p_1 : \ell_1 \quad C \vdash_s^k p \succcurlyeq p_2 : \ell_2}{C \vdash_s^{k+1} p \succcurlyeq p_1 \wedge p_2 : \ell_1 \sqcup \ell_2}$	
DISJ-L $\frac{C \vdash_s^k p_1 \succcurlyeq p : \ell_1 \quad C \vdash_s^k p_2 \succcurlyeq p : \ell_2}{C \vdash_s^{k+1} p_1 \vee p_2 \succcurlyeq p : \ell_1 \sqcup \ell_2}$	DISJ-R $\frac{j \in \{1,2\} \quad C \vdash_s^k p \succcurlyeq p_j : \ell}{C \vdash_s^{k+1} p \succcurlyeq p_1 \vee p_2 : \ell}$	TRANS $\frac{C \vdash_s^k p \succcurlyeq q : \ell_1 \quad C \vdash_s^k q \succcurlyeq r : \ell_2}{C \vdash_s^{k+1} p \succcurlyeq r : \ell_1 \sqcup \ell_2}$	
DEL $\frac{p \succcurlyeq q @ \ell \in \Sigma(n). \Delta \quad \mathcal{H}, p \succcurlyeq q; n; \Sigma \vdash_s^k \ell \sqsubseteq s : \ell' \quad \mathcal{H}, p \succcurlyeq q; n; \Sigma \vdash_s^k \ell' \sqsubseteq s : \perp^\Xi}{\mathcal{H}; n; \Sigma \vdash_s^{k+1} p \succcurlyeq q : \ell}$	FWD $\frac{\ell_n = \Sigma(n). \text{lbl} \quad \sigma_m = \Sigma(m) [\text{lbl} \mapsto \ell_n \sqcup \ell_m] \quad \ell_m = \Sigma(m). \text{lbl} \quad \mathcal{H}; n; \Sigma \vdash_s^k \ell_n \sqcup \ell_m \sqsubseteq m^\rightarrow : \ell_1 \quad \mathcal{H}; m; \Sigma [m \mapsto \sigma_m] \vdash_s^k p \succcurlyeq q : \ell_2}{\mathcal{H}; n; \Sigma \vdash_s^{k+1} p \succcurlyeq q : \ell_1 \sqcup \ell_2}$		

Figure B.9: Acts for judgment of Flamio. The meta-variable C abbreviates $\mathcal{H}; n; \Sigma$.

Proof. Induction in $\Sigma(n). \text{strat}$, followed by induction the depth of the derivation, using Lemma B.3.1 in the $k = 0$ case. \square

B.4 Global Semantics

The final and topmost layer of the evaluation semantics is a global small-step semantics, written $\langle ns, \Sigma, S \rangle \Rightarrow \langle ns', \Sigma', S' \rangle$, and is described by the rules in Figure B.11. Rule G-STEP-LOCAL lifts a local monadic reduction step to the global semantics, and G-STEP-APP and G-STEP-RET implements remote procedure calls and returns, respectively.

$$\begin{array}{c}
\text{G-STEP-LOCAL} \\
\frac{n; \Sigma \vdash S(n) \rightarrow s : \sigma}{\langle n \cdot ns, \Sigma, S \rangle \Rightarrow \langle n \cdot ns, \Sigma [n \mapsto \sigma], S [n \mapsto s] \rangle} \\
\\
\text{G-STEP-APP} \\
\frac{\begin{array}{l} \ell_{n_1} = \Sigma(n_1).lbl \quad \ell_{n_2} = \Sigma(n_2).lbl \quad S(n_2) = \langle \phi_{n_2} \mid es_{n_2} \rangle \\ n_2; \Sigma \vdash \ell_{n_1} \sqcup \ell_{n_2} \sqsubseteq n_2^{\rightarrow} \quad S(n_1) = \langle \phi_{n_1} \mid E[(\lambda_{\tau@p}^{n_2} x. e_{n_1}) e'_{n_1}] \rangle \\ s'_{n_1} = \langle \phi_{n_1} \mid E[\text{wait}(\tau)] \rangle \quad \sigma'_{n_2} = \Sigma(n_2) [lbl \mapsto \ell_{n_1} \sqcup \ell_{n_2}] \\ s'_{n_2} = \langle \phi_{n_2} \mid (\text{toLabeled}_{\ell_{n_2}} p (e_{n_1}[e'_{n_1}/x])) ; es_{n_2} \rangle \end{array}}{\langle n_1 \cdot ns, \Sigma, S \rangle \Rightarrow \langle n_2 \cdot n_1 \cdot ns, \Sigma [n_2 \mapsto \sigma'_{n_2}], S [n_1 \mapsto s'_{n_1}, n_2 \mapsto s'_{n_2}] \rangle} \\
\\
\text{G-STEP-RET} \\
\frac{\begin{array}{l} S(n_1) = \langle \phi_{n_1} \mid E[\text{wait}(\tau)] \rangle \quad S(n_2) = \langle \phi_{n_2} \mid v ; es_{n_2} \rangle \\ s'_{n_1} = \langle \phi_{n_1} \mid E[v] \rangle \quad s'_{n_2} = \langle \phi_{n_2} \mid es_{n_2} \rangle \end{array}}{\langle n_2 \cdot n_1 \cdot ns, \Sigma, S \rangle \Rightarrow \langle n_1 \cdot ns, \Sigma, S [n_1 \mapsto s'_{n_1}, n_2 \mapsto s'_{n_2}] \rangle}
\end{array}$$

Figure B.11: Semantics of global steps

B.5 Type System

As Flamio uses dynamic information-flow control only, the type system is essentially a standard simply-typed lambda calculus type system extended with references. We write $n; \Gamma \vdash e : \tau$ when expression e can be given type τ in a global type environment $\Gamma : \mathcal{N} \rightarrow (\text{Var} \uplus \text{Loc} \rightarrow \tau)$ on node n . Note that the typing environment maps both variables and locations to types. Figure B.12 and B.13 shows the rules of this judgment.

As expected, the type system satisfies a standard statement about preservation. To define preservation we need a notion of a well-formed store.

Definition B.5.1. *A store ϕ is well-formed wrt. n and Γ if, for all a such that $\phi(a) = e @ p$ and $\Gamma(a) = \text{Ref}_n \tau$ it holds that $n; \Gamma \vdash e : \tau$.*

We write $n; \Gamma \vdash \phi$ when ϕ is well-formed wrt. n and Γ .

In order to track the allocations we augment the semantics and keep track of the changes to Γ during the execution. We write

$$n; \Sigma \vdash \langle \phi \mid es \rangle \xrightarrow[\Gamma]{\Gamma'} \langle \phi' \mid es' \rangle : \sigma$$

when the execution of es modified typing environment Γ to Γ' . Only the

T-PRINCIPAL $\frac{}{n; \Gamma \vdash p : \text{Principal}}$	T-BOOL $\frac{b \in \{\text{true}, \text{false}\}}{n; \Gamma \vdash b : \text{Bool}}$	T-UNIT $n; \Gamma \vdash () : \text{Unit}$	T-PROD $\frac{n; \Gamma \vdash e_i : \tau_i \quad i = 1, 2}{n; \Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)}$	T-VAR $\frac{\Gamma_n(x) = \tau}{n; \Gamma \vdash x : \tau}$
T-REF $\frac{\Gamma_n(a) = \text{Ref}_n \tau}{n; \Gamma \vdash a : \text{Ref}_n \tau}$	T-CONS $\frac{n; \Gamma \vdash e_1 : \tau \quad n; \Gamma \vdash e_2 : [\tau]}{n; \Gamma \vdash e_1 :: e_2 : [\tau]}$		T-NIL $n; \Gamma \vdash \text{nil} : [\tau]$	T-LABELED $\frac{n; \Gamma \vdash e : \tau}{n; \Gamma \vdash e @ p : \text{Labeled } \tau}$
T-LIO $\frac{n; \Gamma \vdash e : \tau}{n; \Gamma \vdash (e)^{\text{LIO}} : \text{LIO } \tau}$	T-ABS $\frac{m; \Gamma, x : \tau_1 \vdash e : \text{LIO } \tau_2 \quad \tau = \tau_1 \rightarrow \text{LIO } (\text{Labeled } \tau_2)}{n; \Gamma \vdash \lambda_{\tau_2 @ p}^m x. e : \tau}$		T-APP $\frac{n; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad n; \Gamma \vdash e_2 : \tau_1}{n; \Gamma \vdash e_1 e_2 : \tau_2}$	T-PROJ $\frac{n; \Gamma \vdash e : (\tau_1, \tau_2)}{n; \Gamma \vdash \pi_i e : \tau_i}$
T-IF $\frac{n; \Gamma \vdash e : \text{Bool} \quad n; \Gamma \vdash e_i : \tau}{n; \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$	T-CASE $\frac{n; \Gamma \vdash e : [\tau] \quad n; \Gamma \vdash e_{\text{nil}} : \tau' \quad n; \Gamma \vdash e_{\text{cons}} : \tau \rightarrow [\tau] \rightarrow \tau'}{n; \Gamma \vdash \text{case } e_1 \text{ of } e_{\text{nil}} e_{\text{cons}} : \tau'}$		T-FIX $\frac{n; \Gamma \vdash e : \tau \rightarrow \text{LIO } (\text{Labeled } \tau)}{n; \Gamma \vdash \text{fix } e : \text{LIO } (\text{Labeled } \tau)}$	
T-LAB $\frac{n; \Gamma \vdash e_2 : \tau \quad n; \Gamma \vdash e_1 : \text{Principal}}{n; \Gamma \vdash e_2 @ e_1 : \text{Labeled } \tau}$	T-UNLABEL $\frac{n; \Gamma \vdash e : \text{Labeled } \tau}{n; \Gamma \vdash \text{unlabel } e : \text{LIO } \tau}$	T-RETURN $\frac{n; \Gamma \vdash e : \tau}{n; \Gamma \vdash \text{return } e : \text{LIO } \tau}$		
T-BIND $\frac{n; \Gamma \vdash e_1 : \text{LIO } \tau_1 \quad n; \Gamma \vdash e_2 : \tau_1 \rightarrow \text{LIO } \tau_2}{n; \Gamma \vdash e_1 \gg e_2 : \text{LIO } \tau_2}$	T-TO LABELED $\frac{n; \Gamma \vdash e_1 : \text{Principal} \quad n; \Gamma \vdash e_2 : \text{LIO } \tau \quad \tau' = \text{LIO } (\text{Labeled } \tau)}{n; \Gamma \vdash \text{toLabeled } e_1 e_2 : \tau'}$		T-READ $\frac{n; \Gamma \vdash e : \text{Ref}_n \tau}{n; \Gamma \vdash !e : \text{LIO } \tau}$	
T-WRITE $\frac{n; \Gamma \vdash e_1 : \text{Ref}_n \tau \quad n; \Gamma \vdash e_2 : \tau}{n; \Gamma \vdash e_1 := e_2 : \text{LIO } ()}$	T-NEW $\frac{n; \Gamma \vdash e_1 : \text{Principal} \quad n; \Gamma \vdash e_2 : \tau \quad \tau' = \text{LIO } (\text{Ref}_n \tau)}{n; \Gamma \vdash \text{new } e_1 e_2 : \tau'}$		T-WAIT $\frac{\tau' = \text{LIO } (\text{Labeled } \tau)}{n; \Gamma \vdash \text{wait}(\tau) : \tau'}$	
T-GET LABEL $n; \Gamma \vdash \text{getLabel} : \text{LIO Principal}$		T-GET CLEARANCE $n; \Gamma \vdash \text{getClearance} : \text{LIO Principal}$		
T-GET STRATEGY $n; \Gamma \vdash \text{getStrategy} : \text{LIO [Principal]}$	T-LABEL OF $\frac{n; \Gamma \vdash e : \text{Labeled } \tau}{n; \Gamma \vdash \text{labelOf } e : \text{Principal}}$		T-WITH SCOPE $\frac{n; \Gamma \vdash e : \tau}{n; \Gamma \vdash \text{withScope } e : \tau}$	

Figure B.12: Typing judgment for Flamio.

$\frac{\text{T-ACTSFOR}}{n; \Gamma \vdash e_i : \text{Principal} \quad i = 1, 2} \\ n; \Gamma \vdash e_1 \succcurlyeq e_2 : \text{LIO Bool}$	$\frac{\text{T-WITHSTRATEGY} \quad n; \Gamma \vdash e_1 : [\text{Principal}] \quad n; \Gamma \vdash e_2 : \tau}{n; \Gamma \vdash \text{withStrategy } e_1 \ e_2 : \tau}$	$\frac{\text{T-RESETSCOPE} \quad n; \Gamma \vdash e : \tau}{n; \Gamma \vdash \text{resetScope}_\Delta(e) : \tau}$
$\frac{\text{T-ASSUME} \quad n; \Gamma \vdash e_i : \text{Principal} \quad i = 1, 2, 3}{n; \Gamma \vdash \text{assume } (e_1 \succcurlyeq e_2) @ e_3 : \text{LIO Unit}}$	$\frac{\text{T-TOLEBELED-2} \quad n; \Gamma \vdash e : \text{LIO } \tau \quad \tau' = \text{LIO (Labeled } \tau)}{n; \Gamma \vdash \text{toLabeled}_p \ q \ e : \tau'}$	$\frac{\text{T-RESETSTRATEGY} \quad n; \Gamma \vdash e : \tau}{n; \Gamma \vdash \text{resetStrategy}_{\bar{p}}(e) : \tau}$

Figure B.13: Typing judgment for Flamio (cont).

semantics of allocation is changed:

$$\frac{\begin{array}{l} \text{E-NEW} \\ a \notin \text{dom}(\phi) \quad n; \Sigma \vdash \Sigma(n).lbl \sqsubseteq p \sqsubseteq n^\rightarrow : \ell \\ n; \Sigma \vdash \Sigma(n).lbl \sqcup \ell \sqsubseteq n^\rightarrow \quad \phi' = \phi[a \mapsto e @ p] \\ \sigma = \Sigma(n)[lbl \mapsto \Sigma(n).lbl \sqcup \ell] \quad n; \Gamma \vdash e : \tau \end{array}}{n; \Sigma \vdash \langle \phi \mid \text{new } p \ e \rangle \xrightarrow[\Gamma]{\Gamma[a \mapsto \tau]} \langle \phi' \mid \text{return } a \rangle : \sigma}$$

We lift the typing relation to sequences of expressions and write $n; \Gamma \vdash es : \tau$ when $n; \Gamma \vdash es_i : \tau_{s_i}$ for $i = 1, \dots, |es|$ where $|es| = |\tau_s|$.

An easy induction proof shows that if $n; \Sigma \vdash \langle \phi \mid es \rangle \xrightarrow[\Gamma]{\Gamma'} \langle \phi' \mid es' \rangle : \sigma$ then $n; \Sigma \vdash \langle \phi \mid es \rangle \rightarrow \langle \phi' \mid es' \rangle : \sigma$ and similarly if $n; \Gamma \vdash es : \tau_s$ and $n; \Sigma \vdash \langle \phi \mid es \rangle \rightarrow \langle \phi' \mid es' \rangle : \sigma$ then there exists a Γ' such that $n; \Sigma \vdash \langle \phi \mid es \rangle \xrightarrow[\Gamma]{\Gamma'} \langle \phi' \mid es' \rangle : \sigma$. Using these facts we prove preservation of the local monad reduction relation.

Lemma B.5.1 (Preservation for local reduction). *If $n; \Gamma \vdash \phi$ and $n; \Gamma \vdash es : \tau$ and $n; \Sigma \vdash \langle \phi \mid es \rangle \rightarrow \langle \phi' \mid es' \rangle : \sigma$ then there exists $\Gamma' \supseteq \Gamma$ such that $n; \Gamma' \vdash es' : \tau$ and $n; \Gamma' \vdash \phi'$.*

Proof. Induction in $n; \Sigma \vdash \langle \phi \mid es \rangle \rightarrow \langle \phi' \mid es' \rangle : \sigma$. □

And we lift the result to the global semantics:

Lemma B.5.2 (Preservation for global reduction). *Write $S(n) = \langle \phi_n \mid es_n \rangle$ and assume $n; \Gamma \vdash \phi_n$ and $n; \Gamma \vdash es_n : \tau_{s_n}$ for all $n \in \mathbb{N}$ and $(ns, \Sigma, S) \xrightarrow[\Gamma]{\Gamma'}$*

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

Figure B.14: The syntax of events.

$\langle ns', \Sigma', S' \rangle$. Then there exists $\Gamma' \supseteq \Gamma$ such that for all $n \in \mathcal{N}$ it holds that $n; \Gamma' \vdash \phi'_n$ and $n; \Gamma' \vdash es' : \tau s'_n$ where $S'(n) = \langle \phi'_n \mid es'_n \rangle$, where the following relations between types τs_n and $\tau s'_n$ holds:

1. If $\langle ns, \Sigma, S \rangle \xrightarrow[\Gamma]{\Gamma'} \langle ns', \Sigma', S' \rangle$ is an instance of G-STEP-LOCAL then $\tau s_n = \tau s'_n$.
2. If $\langle ns, \Sigma, S \rangle \xrightarrow[\Gamma]{\Gamma'} \langle ns', \Sigma', S' \rangle$ is an instance of G-STEP-APP then $\tau s_n \leq \tau s'_n$.
3. Otherwise, if $\langle ns, \Sigma, S \rangle \xrightarrow[\Gamma]{\Gamma'} \langle ns', \Sigma', S' \rangle$ is an instance of G-STEP-RET then $\tau s'_n \leq \tau s_n$,

where $\tau s_1 \leq \tau s_2$ states that τs_1 is a suffix of τs_2 .

Proof. Induction in $\langle ns, \Sigma, S \rangle \xrightarrow[\Gamma]{\Gamma'} \langle ns', \Sigma', S' \rangle$, using Lemma B.5.1 in the case of G-STEP-LOCAL. \square

B.6 Attacker Model

We augment the semantics with *events* corresponding to side effects produced during the execution of the program, as well as network communication. The syntax of events is shown in Figure B.14.

The augmented semantics is shown in Figure B.15.

The remaining rules simply emit the empty event ε . We write $\langle ns, \Sigma, S \rangle \xRightarrow{t}^*$ when there exists a configuration $\langle ns', \Sigma', S' \rangle$ such that $\langle ns, \Sigma, S \rangle \xRightarrow{t}^* \langle ns', \Sigma', S' \rangle$ and $S'(n) = \langle \phi_n \mid \bullet \rangle$ for all $n \in \mathcal{N} \setminus \{n'\}$ and $S'(n') = \langle \phi_{n'} \mid v \rangle$. That is, when the execution terminates with a value v .

We also write $\langle ns, \Sigma, S \rangle \xRightarrow{A \rightsquigarrow t'}^*$ when $\langle ns, \Sigma, S \rangle \xRightarrow{t}^*$ and $t' = t \upharpoonright A$.

$$\begin{array}{c}
\text{E-WRITE-EV} \\
\frac{[\dots] \quad ev = (\text{write}(a, e), \Sigma, n)}{n; \Sigma \vdash \langle \phi \mid a := e \rangle \xrightarrow[\Gamma]{\Gamma, ev} \langle \phi' \mid \text{return } () \rangle : \sigma} \\
\\
\text{E-NEW-EV} \\
\frac{[\dots] \quad a \notin \text{dom}(\phi) \quad ev = (\text{new}(a, e), \Sigma, n)}{n; \Sigma \vdash \langle \phi \mid \text{new } p \ e \rangle \xrightarrow[\Gamma]{\Gamma, ev} \langle \phi' \mid \text{return } a \rangle : \sigma} \\
\\
\text{G-STEP-RET-EV} \qquad \text{G-STEP-LOCAL-EV} \\
\frac{S(n) = \langle \phi_n \mid E[\text{wait}(\tau)] \rangle \quad S(m) = \langle \phi_m \mid v; \overline{e_m} \rangle \quad ev = (\text{ret}(v, n), \Sigma, m)}{(\llbracket m \cdot n \cdot ns, \Sigma, S \rrbracket \xrightarrow[\Gamma]{\Gamma, ev} \llbracket n \cdot ns, \Sigma, S[n \mapsto s'_n, m \mapsto s'_m] \rrbracket)} \quad \frac{n; \Sigma \vdash S(n) \xrightarrow{ev} s : \sigma}{(\llbracket n \cdot ns, \Sigma, S \rrbracket \xrightarrow[\Gamma]{\Gamma, ev} \llbracket n \cdot ns, \Sigma[n \mapsto \sigma], S[n \mapsto s] \rrbracket)} \\
\\
\text{E-ASSUME-EV} \\
\frac{[\dots] \quad ev = (\text{release}(p, q, r), \Sigma, n)}{n; \Sigma \vdash \langle \phi \mid E[\text{assume } (p \succcurlyeq q) @ r] \rangle \xrightarrow[\Gamma]{\Gamma, ev} \langle \phi \mid E[\text{return } ()] \rangle : \sigma}
\end{array}$$

Figure B.15: Augmented semantics with events.

This definition uses the *restriction* of trace t wrt. an attacker principal \mathcal{A} :

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

Two events ev_1 and ev_2 are \mathcal{A} -equivalent if they cannot be distinguished by the attacker \mathcal{A} . This is written $ev_1 \simeq_{\mathcal{A}}^{\theta} ev_2$, and the rules for this judgment is shown in Figure B.16. The partial bijection $\theta : \mathbb{A} \rightarrow \mathbb{A}$ is used to relate nondeterministically chosen addresses when allocating references.

As can be seen in many of the rules, this requires a \mathcal{A} -equivalence on expressions. An excerpt of the rules of the relation is defined in Figure B.17, and the remaining rules applies the equivalence homomorphically on the syntax of expressions. Finally, \mathcal{A} -equivalence on traces, written $t_1 \simeq_{\mathcal{A}}^{\theta} t_2$, is defined pointwise.

B.7 Noninterference for Flamio

We define attacker's knowledge as the possible memories that could produce a given trace t , when starting the evaluation on node n .

$$k_{\mathcal{A}}^n(e, t) = \left\{ \Psi \mid (\llbracket n, \emptyset, S \rrbracket \xrightarrow[\ast]{\mathcal{A} \rightsquigarrow t'} \wedge t \simeq_{\mathcal{A}} t' \right\}$$

$$\begin{array}{c}
\frac{\theta(a_1) = a_2 \quad n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq_{\mathcal{A}}^{\theta} e_2}{n; \Sigma_1; \Sigma_2 \vdash \text{write}(a_1, e_1) \simeq_{\mathcal{A}}^{\theta} \text{write}(a_2, e_2)} \quad \frac{\theta(a_1) = a_2 \quad n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq_{\mathcal{A}}^{\theta} e_2}{n; \Sigma_1; \Sigma_2 \vdash \text{new}(a_1, e_1) \simeq_{\mathcal{A}}^{\theta} \text{new}(a_2, e_2)} \\
\\
\frac{n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq_{\mathcal{A}}^{\theta} e_2}{n; \Sigma_1; \Sigma_2 \vdash \text{call}(e_1, n') \simeq_{\mathcal{A}}^{\theta} \text{call}(e_2, n')} \quad \frac{n; \Sigma_1; \Sigma_2 \vdash v_1 \simeq_{\mathcal{A}}^{\theta} v_2}{n; \Sigma_1; \Sigma_2 \vdash \text{ret}(v_1, n') \simeq_{\mathcal{A}}^{\theta} \text{ret}(v_2, n')} \\
\\
\frac{n; \Sigma_1; \Sigma_2 \vdash \text{release}(p, q, r) \simeq_{\mathcal{A}}^{\theta} \text{release}(p, q, r) \quad \begin{array}{c} ev_i = (\alpha_i, \Sigma_i, n) \quad n; \Sigma_i \vdash \Sigma_1(n).lbl \sqsubseteq \mathcal{A} \quad i = 1, 2 \\ \Sigma_1(n).lbl = \Sigma_2(n).lbl \quad n; \Sigma_1; \Sigma_2 \vdash \alpha_1 \simeq_{\mathcal{A}}^{\theta} \alpha_2 \end{array}}{ev_1 \simeq_{\mathcal{A}}^{\theta} ev_2} \\
\\
\frac{\begin{array}{c} ev_i = (\alpha_i, \Sigma_i, n) \\ n; \Sigma_i \vdash \Sigma_i(n).lbl \not\sqsubseteq \mathcal{A} \quad i = 1, 2 \end{array}}{ev_1 \simeq_{\mathcal{A}}^{\theta} ev_2}
\end{array}$$

Figure B.16: \mathcal{A} -equivalence for events.

$$\begin{array}{c}
\frac{n; \Sigma_i \vdash p \sqsubseteq \mathcal{A} \quad n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq_{\mathcal{A}}^{\theta} e_2}{n; \Sigma_1; \Sigma_2 \vdash e_1 @ p \simeq_{\mathcal{A}}^{\theta} e_2 @ p} \quad \frac{n; \Sigma_i \vdash p_i \not\sqsubseteq \mathcal{A}}{n; \Sigma_1; \Sigma_2 \vdash e_1 @ p_1 \simeq_{\mathcal{A}}^{\theta} e_2 @ p_2} \quad n; \Sigma_1; \Sigma_2 \vdash p \simeq_{\mathcal{A}}^{\theta} p \\
\\
n; \Sigma_1; \Sigma_2 \vdash x \simeq_{\mathcal{A}}^{\theta} x \quad \frac{\theta(a_1) = a_2}{n; \Sigma_1; \Sigma_2 \vdash a_1 \simeq_{\mathcal{A}}^{\theta} a_2}
\end{array}$$

Figure B.17: Excerpt of the \mathcal{A} -equivalence for expressions.

where $S(m) = \langle \Psi(n) \mid \llbracket e \rrbracket_n(m) \rangle$ and

$$\llbracket e \rrbracket_n(m) = \begin{cases} e & \text{if } n = m \\ \bullet & \text{otherwise} \end{cases}$$

Using the attacker's knowledge we can formulate a number of security policies [12]. In particular, we can define termination-insensitive noninterference [46] by using the following policy (i.e., a lower bound on the attacker's knowledge):

$$k_{\mathcal{A}}^{\downarrow n}(\Phi, e) = \{ \Psi \mid \Phi \simeq_{\mathcal{A}} \Psi \wedge \langle n, \emptyset, S \rangle \Rightarrow^* \}$$

where $S(m) = \langle \Psi(n) \mid \llbracket e \rrbracket_n(m) \rangle$. This definition uses \mathcal{A} -equivalence on global memories, which is defined as the pointwise extension of memory

$$\begin{array}{c}
\text{STORE-EQ-EMPTY} \\
\hline
n; \Sigma_1; \Sigma_2 \vdash \emptyset \simeq_{\mathcal{A}}^{\emptyset} \emptyset
\end{array}
\qquad
\begin{array}{c}
\text{STORE-EQ-LOW} \\
\theta(a_1) = a_2 \quad n; \Sigma_1; \Sigma_2 \vdash e_1 \simeq_{\mathcal{A}}^{\emptyset} e_2 \\
n; \Sigma_i \vdash q \sqsubseteq \mathcal{A} \quad i = 1, 2 \quad n; \Sigma_1; \Sigma_2 \vdash \phi_1 \simeq_{\mathcal{A}}^{\emptyset} \phi_2 \\
\hline
n; \Sigma_1; \Sigma_2 \vdash \phi_1 [a_1 \mapsto (e_1 @ q)] \simeq_{\mathcal{A}}^{\emptyset} \phi_2 [a_2 \mapsto (e_2 @ q)]
\end{array}$$

$$\begin{array}{c}
\text{STORE-EQ-HIGH-1} \\
n; \Sigma_1 \vdash q \not\sqsubseteq \mathcal{A} \quad n; \Sigma_1; \Sigma_2 \vdash \phi_1 \simeq_{\mathcal{A}}^{\emptyset} \phi_2 \\
\hline
n; \Sigma_1; \Sigma_2 \vdash \phi_1 [a \mapsto e @ q] \simeq_{\mathcal{A}}^{\emptyset} \phi_2
\end{array}
\qquad
\begin{array}{c}
\text{STORE-EQ-HIGH-2} \\
n; \Sigma_2 \vdash q \not\sqsubseteq \mathcal{A} \quad n; \Sigma_1; \Sigma_2 \vdash \phi_1 \simeq_{\mathcal{A}}^{\emptyset} \phi_2 \\
\hline
n; \Sigma_1; \Sigma_2 \vdash \phi_1 \simeq_{\mathcal{A}}^{\emptyset} \phi_2 [a \mapsto e @ q]
\end{array}$$

Figure B.18: \mathcal{A} -equivalence for memories

\mathcal{A} -equivalence defined in Figure B.18. That is, we write $n; \Sigma_1; \Sigma_2 \vdash \Phi \simeq_{\mathcal{A}}^{\emptyset} \Psi$ when $\forall m \in \mathcal{N} . n; \Sigma_1; \Sigma_2 \vdash \Phi(m) \simeq_{\mathcal{A}}^{\emptyset} \Psi(m)$. Note that $n; \emptyset; \emptyset \vdash \phi \simeq_{\mathcal{A}} \psi$ if and only if $m; \emptyset; \emptyset \vdash \phi \simeq_{\mathcal{A}} \psi$, i.e., all nodes without any local delegations agree on \mathcal{A} -equivalence of memories. Thus, we write $\phi \simeq_{\mathcal{A}} \psi$ to mean $n; \emptyset; \emptyset \vdash \phi \simeq_{\mathcal{A}} \psi$ for some n .

We can now express that Flamio programs satisfy termination-insensitive noninterference as follows.

Theorem B.7.1 (Global noninterference). *Let n be a node, Γ be a global typing environment, e be an expression such that $n; \Gamma \vdash e : \tau$ for some type τ .*

If $(n, \emptyset, S) \xrightarrow{\mathcal{A} \rightsquigarrow t^}$ for $S(m) = \langle \Phi(m) \mid \llbracket e \rrbracket_n(m) \rangle$ such that $\mathcal{A} \vdash \text{good}(t)$ then $k_{\mathcal{A}}^n(e, t) \supseteq k_{\mathcal{A}}^{\downarrow n}(\Phi, e)$.*

Before proving this theorem, we state a number of needed lemmas and definitions.

Given two pairs of names and environments $(n_2, \Sigma_1), (n_2, \Sigma_2)$ we write $(n_1, \Sigma_1) \simeq_{\mathcal{A}} (n_2, \Sigma_2)$ when

1. Authorization results that only depend on delegations that the attacker \mathcal{A} can observe are similar:

$$\begin{aligned}
& (n_1; \Sigma_1 \vdash \ell \sqsubseteq \mathcal{A} \wedge n_1; \Sigma_1 \vdash p \succcurlyeq q : \ell) \iff \\
& (n_2; \Sigma_2 \vdash \ell \sqsubseteq \mathcal{A} \wedge n_2; \Sigma_2 \vdash p \succcurlyeq q : \ell)
\end{aligned}$$

2. When the label on the context is observable to the \mathcal{A} , computation happens on the same node in both executions, and the labels on the

contexts (and the strategies) in the two executions are equal:

$$n_1; \Sigma_1 \vdash \Sigma_1(n_1).lbl \sqsubseteq \mathcal{A} \implies \begin{cases} n_1 = n_2 \\ \Sigma_1(n_1).lbl = \Sigma_2(n_2).lbl \\ \Sigma_1(n_1).strat = \Sigma_2(n_2).strat \end{cases}$$

3. The executions always agrees on what is \mathcal{A} -observable:

$$n_1; \Sigma_1 \vdash \Sigma_1(n_1).lbl \sqsubseteq \mathcal{A} \iff n_2; \Sigma_2 \vdash \Sigma_2(n_2).lbl \sqsubseteq \mathcal{A}.$$

We write $n; \Sigma_1; \Sigma_2 \vdash \langle \phi_1 \mid es_1 \rangle \simeq_{\mathcal{A}}^{\theta} \langle \phi_2 \mid es_2 \rangle$ if

1. $n; \Sigma_1; \Sigma_2 \vdash \phi_1 \simeq_{\mathcal{A}}^{\theta} \phi_2$
2. $n; \Sigma_1; \Sigma_2 \vdash es_1 \simeq_{\mathcal{A}}^{\theta} es_2$

Finally, we lift \mathcal{A} -equivalence to global configurations in Figure B.19. Note how rule G-EQ-Low uses the \mathcal{A} -equivalence defined above.

Lemma B.7.2 (Pure step in one run implies pure step in another). *If*

1. $n; \Sigma_i \vdash \Sigma_i(n).lbl \sqsubseteq \mathcal{A} : i = 1, 2$
2. $n; \Sigma_1; \Sigma_2 \vdash es_1 \simeq_{\mathcal{A}} es_2$
3. $es_1 \rightarrow es'_1$.
4. $n; \Sigma_2 \vdash S_2 \rightarrow S'_2 : \sigma'_2$ where

$$S_2(n) = \langle \phi_2 \mid es_2 \rangle$$

$$S'_2(n) = \langle \phi'_2 \mid es'_2 \rangle$$

then

1. $es_2 \rightarrow es'_2$
2. $\Sigma'_2 = \Sigma_2$
3. $\phi'_2 = \phi_2$
- 4.

$$S'_2(m) = \begin{cases} \langle \phi_2 \mid es'_2 \rangle & m = n \\ S_2(m) & \text{otherwise} \end{cases}$$

$$\begin{array}{c}
\text{G-EQ-EMPTY-L} \\
\frac{\mathcal{N}_i \subseteq \mathcal{N} \quad i = 1, 2 \quad m \notin \mathcal{N}_1 \quad \text{dom}(\Sigma_i) = \text{dom}(S_i) = \mathcal{N}_i \quad \langle ns_1, \Sigma_1, S_1 \rangle \simeq_{\mathcal{A}} \langle ns_2, \Sigma_2, S_2 \rangle}{\langle ns_1, \Sigma_1 [m \mapsto \sigma], S_1 [m \mapsto \langle \phi \mid \bullet \rangle] \rangle \simeq_{\mathcal{A}} \langle ns_2, \Sigma_2, S_2 \rangle} \\
\\
\text{G-EQ-EMPTY-R} \\
\frac{\mathcal{N}_i \subseteq \mathcal{N} \quad i = 1, 2 \quad m \notin \mathcal{N}_2 \quad \text{dom}(\Sigma_i) = \text{dom}(S_i) = \mathcal{N}_i \quad \langle ns_1, \Sigma_1, S_1 \rangle \simeq_{\mathcal{A}} \langle ns_2, \Sigma_2, S_2 \rangle}{\langle ns_1, \Sigma_1, S_1 \rangle \simeq_{\mathcal{A}} \langle ns_2, \Sigma_2 [m \mapsto \sigma], S_2 [m \mapsto \langle \phi \mid \bullet \rangle] \rangle} \\
\\
\text{G-EQ-HIGH-L} \\
\frac{\mathcal{N}_i \subseteq \mathcal{N} \quad i = 1, 2 \quad m \notin \mathcal{N}_1 \quad \text{dom}(\Sigma_i) = \text{dom}(S_i) = \mathcal{N}_i \quad n_1; \Sigma_1 \vdash \mathcal{A}^{\rightarrow} \not\geq \sigma.lbl^{\rightarrow} : \text{fail} \quad \langle ns_1, \Sigma_1, S_1 \rangle \simeq_{\mathcal{A}} \langle ns_2, \Sigma_2, S_2 \rangle}{\langle n_1, \Sigma_1 [m \mapsto \sigma], S_1 [m \mapsto \langle \phi \mid e \rangle] \rangle \simeq_{\mathcal{A}} \langle ns_2, \Sigma_2, S_2 \rangle} \\
\\
\text{G-EQ-HIGH-R} \\
\frac{\mathcal{N}_i \subseteq \mathcal{N} \quad i = 1, 2 \quad m \notin \mathcal{N}_2 \quad \text{dom}(\Sigma_i) = \text{dom}(S_i) = \mathcal{N}_i \quad n_2; \Sigma_2 \vdash \mathcal{A}^{\rightarrow} \not\geq \sigma.lbl^{\rightarrow} : \text{fail} \quad \langle ns_1, \Sigma_1, S_1 \rangle \simeq_{\mathcal{A}} \langle ns_2, \Sigma_2, S_2 \rangle}{\langle ns_1, \Sigma_1, S_1 \rangle \simeq_{\mathcal{A}} \langle ns_2, \Sigma_2 [m \mapsto \sigma], S_2 [m \mapsto \langle \phi \mid e \rangle] \rangle} \\
\\
\text{G-EQ-LOW} \\
\frac{\mathcal{N}_i \subseteq \mathcal{N} \setminus \{m\} \quad i = 1, 2 \quad \langle ns_1, \Sigma_1, S_1 \rangle \simeq_{\mathcal{A}} \langle ns_2, \Sigma_2, S_2 \rangle \quad n_1; n_2; \Sigma_1 [m \mapsto \sigma_1]; \Sigma_2 [m \mapsto \sigma_2] \vdash \langle \phi_1 \mid es_1 \rangle \simeq_{\mathcal{A}} \langle \phi_2 \mid es_2 \rangle}{\langle ns_1, \Sigma_1 [m \mapsto \sigma_1], S_1 [m \mapsto \langle \phi_1 \mid es_1 \rangle] \rangle \simeq_{\mathcal{A}} \langle ns_2, \Sigma_2 [m \mapsto \sigma_2], S_2 [m \mapsto \langle \phi_2 \mid es_2 \rangle] \rangle} \\
\\
\text{G-EQ-NIL} \\
\frac{}{\langle ns_1, \emptyset, \emptyset \rangle \simeq_{\mathcal{A}} \langle ns_2, \emptyset, \emptyset \rangle}
\end{array}$$

Figure B.19: \mathcal{A} -equivalence for global configurations.

Proof. Induction in $es_1 \rightarrow es'_1$. \square

Lemma B.7.3 (Local monadic step in run one implies local monadic step in run two). *If*

1. $n \cdot ns_1; \Sigma_1 \vdash S_1 \xrightarrow{ev_1} S'_1 : \sigma'_1$ where $\forall n. S_1(n) = \langle \phi_1 \mid es_1 \rangle$.
2. $\langle n \cdot ns_2, \Sigma_2, S_2 \rangle \xRightarrow{ev_2} \langle ns'_2, \Sigma'_2, S'_2 \rangle$ where $\forall n. S_2(n) = \langle \phi_2 \mid es_2 \rangle$.
3. $n; \Sigma_i \vdash \Sigma_i(n).lbl \sqsubseteq \mathcal{A} : i = 1, 2$
4. $n; \Sigma_1; \Sigma_2 \vdash es_1 \simeq_{\mathcal{A}} es_2$

then

1. $n; \Sigma_2 \vdash S_2 \xrightarrow{ev_2} S'_2 : \sigma'_2$
2.
$$\Sigma'_2(m) = \begin{cases} \sigma'_2 & m = n \\ \Sigma_2(n) & \text{otherwise} \end{cases}$$
3.
$$S'_2(m) = \begin{cases} \sigma'_2 & \text{if } m = n \\ S_2(m) & \text{otherwise} \end{cases}$$

Proof. Induction in $n; \Sigma_1 \vdash S_1 \xrightarrow{ev_1} S'_1 : \sigma'_1$. \square

Lemma B.7.4 (RPC step in run one implies RPC step in run two). *If $\langle n \cdot ns_1, \Sigma_1, S_1 \rangle \xRightarrow{ev_1} \langle o \cdot m \cdot ns'_1, \Sigma'_1, S'_1 \rangle$ where $\forall n. S_1(n) = \langle \phi_1^n \mid es_1^n \rangle$ and*

1. $S'_1(n) = \langle \phi_n^1 \mid E_1[\text{wait}(\tau)] \rangle$
2. $S'_1(o) = \langle \phi_o^1 \mid (\text{toLabeled}_{\Sigma_1(m).lbl} \ p \ (e_{\text{body}}^1[e_{\text{arg}}^1/x])); es_o^1 \rangle$
3. $e_n^1 = E_1[(\lambda_{\tau@p}^o \ x. e_{\text{body}}^1) \ e_{\text{arg}}^1]$
4. $e_o^1 = es_o^1$
5. $\langle n \cdot ns_2, \Sigma_2, S_2 \rangle \xRightarrow{t}^* \langle ns''_2, \Sigma''_2, S''_2 \rangle \not\Rightarrow$ where $\forall n. S_2(n) = \langle \phi_n^2 \mid e_n^2 \rangle$
6. $\forall m \in \mathcal{N}$ it holds that $n; \Sigma_1; \Sigma_2 \vdash es_m^1 \simeq_{\mathcal{A}} es_m^2$

then there exists $es_{\text{body}}^2, es_{\text{arg}}^2, es_2, E_2$ such that

$$\begin{aligned} \langle n \cdot ns_2, \Sigma_2, S_2 \rangle &\xRightarrow{ev_2} \langle o \cdot n \cdot ns_2, \Sigma'_2, S'_2 \rangle \\ &\xRightarrow{t'}^* \langle n''_2, \Sigma''_2, S''_2 \rangle \end{aligned}$$

where

$$S'_2(m) = \begin{cases} \langle \phi_n^2 \mid E_2[\text{wait}(\tau)] \rangle & m = n \\ \langle \phi_o^2 \mid (\text{toLabeled}_{\Sigma_2(n).lbl} \text{ p } (e_{\text{body}}^2[e_{\text{arg}}^2/x])); es_2 \rangle & m = o \\ S_2(n) & \text{otherwise} \end{cases}$$

and

1. $e_m^2 = E_2[(\lambda_{\tau@p}^o x. e_{\text{body}}^2) e_{\text{arg}}^2]$
2. $e_o^2 = \overline{e_2}$
3. $ev_2 = (\text{call}((e_{\text{body}}^2[e_{\text{arg}}^2/x]), o), \Sigma_2, n).$
4. $\Sigma'_2 = \Sigma_2[o \mapsto \Sigma_2(o)[lbl \mapsto \Sigma_2(o).lbl \sqcup \Sigma_2(n).lbl]].$
5. $t = ev_2 \cdot t'.$

Proof. Follows immediately from the hypotheses $n; \Sigma_1; \Sigma_2 \vdash es_m^1 \simeq_{\mathcal{A}} es_m^2$ and $e_n^1 = E_1[(\lambda_{\tau@p}^o x. e_{\text{body}}^1) e_{\text{arg}}^1]$ \square

Lemma B.7.5. If $\langle n \cdot m \cdot ns_1, \Sigma_1, S_1 \rangle \xRightarrow{ev_1} \langle ns'_1, \Sigma'_1, S'_1 \rangle$ and

1. $S_1(n) = \langle \phi_n^1 \mid v_1; es_1 \rangle$
2. $S_1(m) = \langle \phi_m^1 \mid E_1[\text{wait}(\tau)] \rangle$
3. $S'_1(m) = \langle \phi_m^1 \mid E_1[v_1] \rangle$
4. $S'_1(n) = \langle \phi_n^1 \mid es_1 \rangle$
5. For all $m \in \mathcal{N}$ it holds that $n; \Sigma_1; \Sigma_2 \vdash es_m^1 \simeq_{\mathcal{A}} es_m^2$

then there exists es_2 such that $\langle n \cdot m \cdot ns_2, \Sigma_2, S_2 \rangle \xRightarrow{ev_2} \langle m \cdot ns, \Sigma_2, S'_2 \rangle$ where

$$S'_2(o) = \begin{cases} \langle \phi_n^2 \mid es_2 \rangle & o = n \\ \langle \phi_m^2 \mid E_1[v_2] \rangle & o = m \\ S_2(o) & \text{otherwise} \end{cases}$$

and

1. $e_m^2 = E_2[\text{wait}(\tau)]$
2. $e_n^2 = v_2; es_2$
3. $ev_2 = (\text{ret}(v_2, m), \Sigma_2, n)$

We first prove a termination-insensitive noninterference theorem for local monadic reduction steps.

Lemma B.7.6 (Local noninterference). *If*

1. $n; \Sigma_i \vdash \Sigma_i(n).lbl \sqsubseteq \mathcal{A}$ for $i = 1, 2$
2. $(n, \Sigma_1, S_1) \simeq_{\mathcal{A}} (n, \Sigma_2, S_2)$
3. $n; \Sigma_1 \vdash S_1(n) \xrightarrow{ev_1} s'_1 : \sigma'_1$
4. $n; \Sigma_2 \vdash S_2(n) \xrightarrow{ev_2} s'_2 : \sigma'_2$

then there exists $\theta' \supseteq \theta$ such that

1. $n; \Sigma_1; \Sigma_2 \vdash ev_1 \simeq_{\mathcal{A}}^{\theta'} ev_2$
2. $(n, \Sigma'_1, S'_1) \simeq_{\mathcal{A}} (n, \Sigma'_2, S'_2)$

where $\Sigma'_i = \Sigma_i [n \mapsto \sigma'_i]$ and $S'_i = S_i [n \mapsto s'_i]$ for $i = 1, 2$.

Proof. Induction in $n; \Sigma_1 \vdash S_1(n) \xrightarrow{ev_1} s'_1 : \sigma'_1$. □

Lemma B.7.7. *If*

1. $\langle n \cdot ns, \Sigma_1, S_1 \rangle \Rightarrow \langle ns'_1, \Sigma'_1, S'_1 \rangle$ where $S_1(n) = \langle \phi_1 \mid es_1 \rangle$
2. $\langle n \cdot ns, \Sigma_2, S_2 \rangle \xRightarrow{t}^k \langle ns''_2, \Sigma''_2, S''_2 \rangle \not\Rightarrow$ where $S_2(n) = \langle \phi_2 \mid es_2 \rangle$
3. $n; \Sigma_i \vdash \Sigma_i(n).lbl \sqsubseteq \mathcal{A} : i = 1, 2$
4. $n; \Sigma_1; \Sigma_2 \vdash es_1 \simeq_{\mathcal{A}} es_2$

then

1. $k > 0$
2. $\langle n \cdot ns, \Sigma_2, S_2 \rangle \xRightarrow{ev} \langle ns'_2, \Sigma'_2, S'_2 \rangle \xRightarrow{t'}^{k-1} \langle ns''_2, \Sigma''_2, S''_2 \rangle$

3. $t = ev \cdot t'$

Proof. Induction in $\langle n \cdot ns_1, \Sigma_1, S_1 \rangle \Rightarrow \langle ns'_1, \Sigma'_1, S'_1 \rangle$. \square

We now prove Theorem B.7.1.

Proof. We generalize the statement and prove that

$H_1 \langle n \cdot ns, \Sigma_1, S_1 \rangle \xrightarrow{\mathcal{A} \rightsquigarrow t_1^k} \langle ns'_1, \Sigma'_1, S'_1 \rangle \not\Rightarrow$ where $S_1(n) = \langle \phi_n^1 \mid es_n^1 \rangle$ and $S'_1(n) = \langle \phi_n^{1'} \mid es_n^{1'} \rangle$.

$H_2 \langle n \cdot ns, \Sigma_2, S_2 \rangle \xrightarrow{\mathcal{A} \rightsquigarrow t_2^*} \langle ns'_2, \Sigma'_2, S'_2 \rangle \not\Rightarrow$ where $S_2(n) = \langle \phi_n^2 \mid es_n^2 \rangle$ and $S'_2(n) = \langle \phi_n^{2'} \mid es_n^{2'} \rangle$

$H_3 \langle n \cdot ns, \Sigma_1, S_1 \rangle \simeq_{\mathcal{A}} \langle n \cdot ns, \Sigma_2, S_2 \rangle$

$H_4 \langle n, \Sigma_1 \rangle \simeq_{\mathcal{A}} \langle n, \Sigma_2 \rangle$

$H_5 n; \Sigma_i \vdash \Sigma_i(n).|bl| \sqsubseteq \mathcal{A}$ for $i = 1, 2$.

H_6 It holds that $\forall p, q, e \cdot \text{toLabeled}_p \ q \ e \notin es_n^i$ for $i = 1, 2$.¹

H_7 It holds that $\forall \tau, m \cdot \text{wait}(\tau) \notin es_m^i$ for $i = 1, 2$.

implies

$G_1 t_1 \simeq_{\mathcal{A}} t_2$

$G_2 \langle ns'_1, \Sigma'_1, S'_1 \rangle \simeq_{\mathcal{A}} \langle ns'_2, \Sigma'_2, S'_2 \rangle$

$G_3 (ns'_1, \Sigma'_1) \simeq_{\mathcal{A}} (ns'_2, \Sigma'_2)$

G_4 It holds that $\forall m, p, q, e \cdot \text{toLabeled}_p \ q \ e \notin es_m^{i'}$ for $i = 1, 2$.

G_5 It holds that $\forall \tau, m \cdot \text{wait}(\tau) \notin es_m^{i'}$ for $i = 1, 2$.

The proof proceeds by induction in k .

$k = 0$: Then $t_1 = \varepsilon$ and $es_n^1 = \bullet$ for all $n \in \mathcal{N}$. Thus $e_n^2 = \bullet$ for all $n \in \mathcal{N}$ and so $t_2 = \varepsilon$. The remaining goals all holds by assumption.

¹We use $e_1 \in e_2$ and $e_1 \notin e_2$ to mean that e_1 is a subexpression of e_2 and e_1 is not a subexpression of e_2 respectively. We lift these and write $e \in es$ to mean that $e \in es_i$ for some i and $e \notin es$ to mean that $e \notin es_i$ for all i , respectively.

$k > 0$: Then

$$\langle ns_1, \Sigma_1, S_1 \rangle \xRightarrow{ev_1} \langle ns'_1, \Sigma'_1, S'_1 \rangle \xRightarrow{\mathcal{A} \rightsquigarrow t'_1}^{k-1} \langle ns''_1, \Sigma''_1, S''_1 \rangle$$

where $t_1 = ev_1 \cdot t'_1$ by H_5 . By Lemma B.7.7 it holds that

$$\langle ns_2, \Sigma_2, S_2 \rangle \xRightarrow{ev_2} \langle ns'_2, \Sigma'_2, S'_2 \rangle \xRightarrow{\mathcal{A} \rightsquigarrow t'_2}^* \langle ns''_2, \Sigma''_2, S''_2 \rangle$$

where $t_2 = ev_2 \cdot t'_2$.

The proof proceeds by induction in $\langle n \cdot ns_1, \Sigma_1, S_1 \rangle \xRightarrow{ev_1} \langle ns'_1, \Sigma'_1, S'_1 \rangle$.

G-STEP-LOCAL-EV: Follows by Lemma B.7.3 and Lemma B.7.6.

G-STEP-APP-EV: Follows by Lemma B.7.4 and the induction hypothesis.

G-STEP-RET-EV: Follows by Lemma B.7.5 and the induction hypothesis.

□

Bibliography

- [1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin.
Dynamic Typing in a Statically-typed Language.
In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–227.
Cited on page [77](#).
- [2] Martin Abadi.
Access Control in a Core Calculus of Dependency.
In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 263–273.
Cited on page [110](#).
- [3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti.
Control-flow Integrity Principles, Implementations, and Applications.
In *ACM Transactions on Information and System Security*, 13, pages 4:1–4:40, 2009.
Cited on page [27](#).
- [4] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault.
Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation.
In *Proceedings of the 32th IEEE Computer Security Foundations Symposium*.
Cited on page [141](#).
- [5] Johan Agat.
Transforming out Timing Leaks.
In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53.
Cited on page [35](#).
- [6] Amal Ahmed and David Walker.
The Logical Approach to Stack Typing.

In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 74–85.

Cited on pages 78, 83 and 101.

- [7] Andrew W Appel.
Compiling with continuations.
Cambridge University Press, 2006.
Cited on pages 80 and 100.
- [8] O. Arden and A. C. Myers.
A Calculus for Flow-Limited Authorization.
In *2016 IEEE 29th Computer Security Foundations Symposium*, pages 135–149.
Cited on pages 9, 19, 28 and 137.
- [9] Owen Arden, Jed Liu, and Andrew C. Myers.
Flow-Limited Authorization.
In *Proceedings of the IEEE 28th Computer Security Foundations Symposium*, pages 569–583.
Cited on pages 14, 16, 28, 105, 106, 108, 110, 112, 119, 133, 136 and 139.
- [10] Owen Arden, Jed Liu, and Andrew C. Myers.
Flow-limited Authorization: Technical Report.
Technical Report 1813-40138, Cornell University Computing and Information Science, 2015.
Cited on pages 125 and 137.
- [11] Aslan Askarov and Andrew C. Myers.
Attacker Control and Impact for Confidentiality and Integrity.
In *Logical Methods in Computer Science*, 7, 2011.
Cited on pages 125 and 137.
- [12] Aslan Askarov and Andrei Sabelfeld.
Gradual Release: Unifying Declassification, Encryption and Key Release Policies.
In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 207–221.
Cited on pages 17, 19, 64, 127, 132 and 176.
- [13] 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*, pages 333–348.
Cited on pages 19, 25, 61 and 132.

- [14] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers.
Predictive Black-box Mitigation of Timing Channels.
In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 297–307.
Cited on pages [43](#) and [77](#).
- [15] Mounir Assaf and David A. Naumann.
Calculational Design of Information Flow Monitors (extended version).
In *CoRR*, abs/1605.02778, 2016.
Cited on page [59](#).
- [16] 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*, pages 1–12.
Cited on page [17](#).
- [17] Anindya Banerjee and David A. Naumann.
Secure Information Flow and Pointer Confinement in a Java-like Language.
In *Proceedings of the 15th IEEE Workshop on Computer Security Foundations*, pages 253–267.
Cited on pages [19](#), [50](#) and [131](#).
- [18] Gilles Barthe, Pedro R. D’argenio, and Tamara Rezk.
Secure Information Flow by Self-composition.
In *Mathematical Structures in Computer Science*, 21, pages 1207–1252, 2011.
Cited on page [6](#).
- [19] E. D. Bell and J. L. La Padula.
Secure Computer System: Unified Exposition and Multics Interpretation, 1976.
Cited on page [11](#).
- [20] Daniel J. Bernstein.
Cache-timing attacks on AES.
Technical report, The University of Illinois at Chicago, Department of Mathematics, Statistics, and Computer Science, Chicago, IL 60607–7045, 2005.
Cited on page [21](#).
- [21] J. K. Biba.
Integrity Considerations for Secure Computer Systems, 1977.

Cited on page 11.

- [22] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds.
Local Reasoning About a Copying Garbage Collector.
In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–231.
Cited on page 103.
- [23] Joachim Breitner.
dup – Explicit un-sharing in Haskell.
In *CoRR*, abs/1207.2017, 2012.
Cited on page 25.
- [24] Niklas Broberg, Bart van Delft, and David Sands.
The Anatomy and Facets of Dynamic Policies.
In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 122–136.
Cited on page 17.
- [25] Pablo Buiras and Alejandro Russo.
Lazy Programs Leak Secrets.
In *Proceedings of the 18th Nordic Conference on Secure IT Systems - Volume 8208*, pages 116–122.
Cited on pages 24, 25, 32 and 67.
- [26] Pablo Buiras and Bart van Delft.
Dynamic Enforcement of Dynamic Policies.
In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, pages 28–41.
Cited on page 107.
- [27] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo.
HLIO: Mixing Static and Dynamic Typing for Information-flow Control in Haskell.
In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 289–301.
Cited on pages 9 and 107.
- [28] Ethan Cecchetti, Andrew C. Myers, and Owen Arden.
Nonmalleable Information Flow Control.
In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 1875–1891.
Cited on pages 6, 9, 11, 19, 22 and 129.

- [29] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng.
Secure Web Applications via Automatic Partitioning.
In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 31–44.
Cited on pages [125](#), [126](#), [137](#) and [139](#).
- [30] Stephen Chong, K. Vikram, and Andrew C. Myers.
SIF: Enforcing Confidentiality and Integrity in Web Applications.
In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16.
Cited on page [139](#).
- [31] Michael R. Clarkson and Fred B. Schneider.
Hyperproperties.
In *Journal of Computer Security*, 18, pages 1157–1210, 2010.
Cited on pages [5](#) and [19](#).
- [32] Ezra E.K. Cooper and Philip Wadler.
The RPC Calculus.
In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 231–242.
Cited on pages [134](#) and [138](#).
- [33] Zhenyue Deng and Geoffrey Smith.
Lenient Array Operations for Practical Secure Information Flow.
In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, pages 115–124.
Cited on page [55](#).
- [34] Dorothy E. Denning and Peter J. Denning.
Certification of Programs for Secure Information Flow.
In *Communications of the ACM*, 20, pages 504–513, 1977.
Cited on pages [6](#), [8](#), [55](#) and [103](#).
- [35] Dominique Devriese and Frank Piessens.
Noninterference Through Secure Multi-execution.
In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 109–124.
Cited on page [67](#).
- [36] Catalin Dima, Constantin Enea, and Radu Gramatovici.
Nondeterministic Nointerference and Deducible Information Flow.
Technical report, University of Paris, 2006.
Cited on pages [17](#) and [64](#).

- [37] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris.
Labels and Event Processes in the Asbestos Operating System.
In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30.
Cited on pages [28](#) and [107](#).
- [38] Federal Information Processing Standards Publication.
Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.
Cited on page [21](#).
- [39] Mattias Felleisen.
The Theory and Practice of First-class Prompts.
In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 180–190.
Cited on page [117](#).
- [40] J. S. Fenton.
Memoryless subsystems.
In *The Computer Journal*, 17, pages 143–147, 1974.
Cited on pages [10](#) and [16](#).
- [41] Jeffrey S. Fenton.
Information Protection Systems.
PhD thesis, University of Cambridge, 1973.
Cited on page [10](#).
- [42] David F. Ferraiolo, John F. Barkley, and D. Richard Kuhn.
A Role-based Access Control Model and Reference Implementation Within a Corporate Intranet.
In *ACM Transactions on Information and System Security*, 2, pages 34–64, 1999.
Cited on page [106](#).
- [43] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh.
Hyperflow: A Processor Architecture for Nonmalleable, Timing-Safe Information-Flow Security.
In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*.
Cited on pages [22](#) and [138](#).

- [44] Daniel Frampton, David F. Bacon, Perry Cheng, and David Grove.
Generational Real-time Garbage Collection: A Three-part Invention
for Young Objects.
In *Proceedings of the 21st European Conference on Object-Oriented Pro-
gramming*, pages 101–125.
Cited on page 66.
- [45] Peter Gammie, Antony L. Hosking, and Kai Engelhardt.
Relaxing Safely: Verified On-the-fly Garbage Collection for x86-TSO.
In *Proceedings of the 36th ACM SIGPLAN Conference on Programming
Language Design and Implementation*, pages 99–109.
Cited on pages 68 and 103.
- [46] Joseph A. Goguen and José Meseguer.
Security Policies and Security Models.
In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–
20.
Cited on pages 5, 65, 89, 107, 133 and 176.
- [47] M. Gordon.
The semantic challenge of Verilog HDL.
In *Proceedings of 10th Annual IEEE Symposium on Logic in Computer Sci-
ence*, pages 136–145.
Cited on page 22.
- [48] James Gosling and Greg Bollella.
The Real-Time Specification for Java.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA,
2000.
Cited on page 66.
- [49] Simon Gregersen, Søren Eller Thomsen, and Aslan Askarov.
A Dependently Typed Library for Static Information-Flow Control in
Idris.
In *Principles of Security and Trust*, pages 51–75.
Cited on page 103.
- [50] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling
Wang, and James Cheney.
Region-based Memory Management in Cyclone.
In *Proceedings of the ACM SIGPLAN Conference on Programming Lan-
guage Design and Implementation*, pages 282–293.
Cited on page 102.

- [51] Yu Guo, Xinyu Feng, Zhong Shao, and Peizhi Shi.
Modular Verification of Concurrent Thread Management.
In *Programming Languages and Systems*, pages 315–331.
Cited on page [103](#).
- [52] J. Halpern and K. O’Neill.
Secrecy in multiagent systems.
In *Proceedings 15th IEEE Computer Security Foundations Workshop*, pages 32–46.
Cited on page [17](#).
- [53] Robert Harper and Greg Morrisett.
Compiling Polymorphism Using Intensional Type Analysis.
In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141.
Cited on pages [72](#) and [77](#).
- [54] Daniel Hedin and Andrei Sabelfeld.
A Perspective on Information-Flow Control.
In *Software Safety and Security - Tools for Analysis and Verification*, 2012, pages 319–347.
Cited on page [19](#).
- [55] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld.
JSFlow: Tracking Information Flow in JavaScript and its APIs.
In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*.
Cited on pages [19](#) and [33](#).
- [56] Nevin Heintze and Jon G. Riecke.
The SLam Calculus: Programming with Secrecy and Integrity.
In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377.
Cited on page [24](#).
- [57] Andrew K. Hirsch, Pedro H. Azevedo de Amorim, Ethan Cecchetti, Ross Tate, and Owen Arden.
First-Order Logic for Flow-Limited Authorization (Technical Report).
In submission, 2019.
Cited on pages [14](#) and [28](#).
- [58] José A. Joao, Onur Mutlu, and Yale N. Patt.
Flexible Reference-counting-based Hardware Acceleration for Garbage Collection.

- In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 418–428.
Cited on page [66](#).
- [59] Richard Jones, Antony Hosking, and Eliot Moss.
The Garbage Collection Handbook: The Art of Automatic Memory Management.
Chapman & Hall/CRC, 1st edition, 2011.
Cited on pages [36](#), [94](#) and [95](#).
- [60] Simon L. Peyton Jones.
Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine.
In *Journal of Functional Programming*, 2, pages 127–202, 1992.
Cited on page [100](#).
- [61] Aleksandr Karbyshev, Kasper Svendsen, Aslan Askarov, and Lars Birkedal.
Compositional Non-interference for Concurrent Programs via Separation and Framing.
In *Principles of Security and Trust*, pages 53–78.
Cited on pages [19](#), [20](#), [27](#) and [99](#).
- [62] Paul C. Kocher.
Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems.
In *Advances in Cryptology*, pages 104–113.
Cited on page [20](#).
- [63] Elisavet Kozyri, Fred B. Schneider, Andrew Bedford, Josée Desharnais, and Nadia Tawbi.
Beyond Labels: Permissiveness for Dynamic Information Flow Enforcement.
In *Proceedings of the 32th IEEE Computer Security Foundations Symposium*.
Cited on page [142](#).
- [64] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris.
Information Flow Control for Standard OS Abstractions.
In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 321–334.
Cited on pages [28](#), [107](#) and [138](#).

- [65] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber.
Authentication in Distributed Systems: Theory and Practice.
In *ACM Transactions on Computer Systems*, 10, pages 265–310, 1992.
Cited on page [110](#).
- [66] Butler W. Lampson.
A Note on the Confinement Problem.
In *Communications of the ACM*, 16, pages 613–615, 1973.
Cited on pages [5](#), [9](#), [18](#) and [161](#).
- [67] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Wayne, and Andrew C. Myers.
Fabric: A Platform for Secure Distributed Computation and Storage.
In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 321–334.
Cited on pages [14](#), [15](#), [134](#) and [139](#).
- [68] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers.
Fabric: Building open distributed systems securely by construction.
In *Journal of Computer Security*, 25, pages 367–426, 2017.
Cited on pages [14](#) and [15](#).
- [69] Luísa Lourenço and Luís Caires.
Dependent Information Flow Types.
In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–328.
Cited on pages [9](#), [19](#), [83](#) and [103](#).
- [70] Heiko Mantel and Andrei Sabelfeld.
A Generic Approach to the Security of Multi-Threaded Programs.
In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, pages 126–142.
Cited on page [93](#).
- [71] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li.
A General Framework for Certifying Garbage Collectors and Their Mutators.
In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 468–479.
Cited on page [103](#).
- [72] Andrew McCreight, Tim Chevalier, and Andrew Tolmach.
A Certified Framework for Compiling and Executing Garbage-collected Languages.

- In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 273–284.
Cited on page 68.
- [73] Daryl McCullough.
Noninterference and the Composability of Security Properties.
In *Proceedings of the IEEE Conference on Security and Privacy*, pages 177–186.
Cited on page 63.
- [74] Robin Milner.
A Theory of Type Polymorphism in Programming.
In *Journal of Computer and System Sciences*, 17, pages 348–375, 1978.
Cited on page 9.
- [75] Benoît Montagu, Benjamin C. Pierce, and Randy Pollack.
A Theory of Information-Flow Labels.
In *Proceedings of the IEEE 26th Computer Security Foundations Symposium*, pages 3–17.
Cited on page 11.
- [76] Scott Moore, Aslan Askarov, and Stephen Chong.
Precise Enforcement of Progress-sensitive Security.
In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 881–893.
Cited on page 61.
- [77] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong.
Extensible Access Control with Authorization Contracts.
In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 214–233.
Cited on page 116.
- [78] Greg Morrisett, Matthias Felleisen, and Robert Harper.
Abstract Models of Memory Management.
In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 66–77.
Cited on pages 35, 51 and 54.
- [79] Greg Morrisett, Karl Crary, Neal Glew, and David Walker.
Stack-based Typed Assembly Language.
In *Types in Compilation*, pages 28–52.
Cited on pages 79 and 101.

- [80] Greg Morrisett, David Walker, Karl Crary, and Neal Glew.
From System F to Typed Assembly Language.
In *ACM Transactions on Programming Languages and Systems*, 21, pages 527–568, 1999.
Cited on page 101.
- [81] Stefan Muller and Stephen Chong.
Towards a Practical Secure Concurrent Language.
In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 57–74.
Cited on pages 9, 19, 43 and 122.
- [82] Andrew C. Myers and Barbara Liskov.
Protecting Privacy Using the Decentralized Label Model.
In *ACM Transactions on Software Engineering and Methodology*, 9, pages 410–442, 2000.
Cited on pages 12, 13 and 56.
- [83] Andrew C. Myers and 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.
Cited on pages 9, 19, 55, 72, 86, 107, 129 and 139.
- [84] Andrew C. Myers, Andrew C. Myers, and Barbara Liskov.
A Decentralized Model for Information Flow Control.
In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 129–142.
Cited on page 11.
- [85] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic.
Enforcing Robust Declassification.
In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, pages 172–186.
Cited on pages 9 and 129.
- [86] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom.
Jif 3.0: Java Information Flow.
Software release, <http://www.cs.cornell.edu/jif>, 2006.
Cited on page 33.
- [87] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic.

- CETS: Compiler Enforced Temporal Safety for C.
In *Proceedings of the International Symposium on Memory Management*,
pages 31–40.
Cited on pages 75 and 102.
- [88] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg.
Verification of Information Flow and Access Control Policies with De-
pendent Types.
In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 165–
179.
Cited on page 67.
- [89] Kensuke Narita and Shin-ya Nishizaki.
A Parallel Abstract Machine for the RPC Calculus.
In *Informatics Engineering and Information Science*, pages 320–332.
Cited on page 138.
- [90] 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*, pages 1–20.
Cited on pages 21 and 32.
- [91] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria.
Multi-run Side-Channel Analysis Using Symbolic Execution and Max-
SMT.
In *Proceedings of the IEEE 29th Computer Security Foundations Sympo-
sium*, pages 387–400.
Cited on page 33.
- [92] Mathias V. Pedersen and Aslan Askarov.
From Trash to Treasure: Timing-Sensitive Garbage Collection.
In *Proceedings of the IEEE Symposium on Security and Privacy*.
Cited on pages 9, 19, 25, 26, 27, 70, 71, 78, 89, 93, 94, 98, 102 and 141.
- [93] Mathias V. Pedersen and Aslan Askarov.
Static Enforcement of Secure Runtime Systems.
In *Proceedings of the 32th IEEE Computer Security Foundations Sympo-
sium*.
Cited on pages 9, 17, 19, 25, 26 and 141.
- [94] Mathias V. Pedersen and Stephen Chong.
Programming with Flow-Limited Authorization: Coarser is Better
(Technical Report).
Technical report, 2018.

Cited on page 133.

- [95] Mathias V. Pedersen and Stephen Chong.
Flamio implementation in Haskell. <https://www.dropbox.com/s/zxy991pjeepl8nn/FLAMinLIO.zip>, 2018.
Cited on pages 107 and 133.
- [96] Mathias V. Pedersen and Stephen Chong.
Programming with Flow-Limited Authorization: Coarser is Better.
In *Proceedings of the 4th IEEE European Symposium on Security and Privacy*.
Cited on pages 19, 25, 27 and 141.
- [97] Pedersen, Mathias V., Askarov, Aslan.
Static enforcement of secure runtime systems: technical report.
Technical report, 2018.
Cited on pages 82 and 97.
- [98] Colin Percival.
Cache missing for fun and profit.
In *Proceedings of BSDCan 2015*.
Cited on page 21.
- [99] Frances Perry, Chris Hawblitzel, and Juan Chen.
Simple and Flexible Stack Types.
In *Proceedings of the 3rd International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming*, page 20.
Cited on pages 78 and 83.
- [100] Simon Peyton Jones et al.
The Haskell 98 Language and Libraries: The Revised Report.
In *Journal of Functional Programming*, 13, pages 0–255, 2003.
Cited on page 27.
- [101] Benjamin C. Pierce.
Types and Programming Languages.
The MIT Press, 1st edition, 2002.
Cited on page 77.
- [102] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek.
Schism: Fragmentation-tolerant Real-time Garbage Collection.
In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–159.
Cited on page 66.

- [103] François Pottier and Vincent Simonet.
Information Flow Inference for ML.
In *ACM Transactions on Programming Languages and Systems*, 25, pages 117–158, 2003.
Cited on pages 6, 9, 19, 55 and 107.
- [104] Vineet Rajani and Deepak Garg.
Types for Information Flow Control: Labeling Granularity and Semantic Models.
In *Proceedings of the IEEE 31st Computer Security Foundations Symposium*.
Cited on pages 9, 19, 131, 138 and 141.
- [105] R. L. Rivest, A. Shamir, and L. Adleman.
A Method for Obtaining Digital Signatures and Public-key Cryptosystems.
In *Communications of the ACM*, 21, pages 120–126, 1978.
Cited on page 21.
- [106] Alejandro Russo and Andrei Sabelfeld.
Dynamic vs. Static Flow-Sensitive Security Analysis.
In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, pages 186–199.
Cited on page 59.
- [107] A. Sabelfeld and A. C. Myers.
Language-based Information-flow Security.
In *IEEE Journal on Selected Areas in Communications*, 21, pages 5–19, 2006.
Cited on pages 11, 31, 43, 70 and 103.
- [108] A. Sabelfeld and D. Sands.
Probabilistic noninterference for multi-threaded programs.
In *Proceedings 13th IEEE Computer Security Foundations Workshop*.
Cited on pages 23, 27, 99 and 102.
- [109] Andrei Sabelfeld.
The Impact of Synchronisation on Secure Information Flow in Concurrent Programs.
In *Lecture Notes in Computer Science*, 2001, pages 225–239.
Cited on pages 70 and 99.
- [110] Andrei Sabelfeld and David Sands.
A Per Model of Secure Information Flow in Sequential Programs.
In *Higher-Order and Symbolic Computation*, 14, pages 59–91, 2001.

Cited on pages [19](#) and [24](#).

- [111] J. H. Saltzer, D. P. Reed, and D. D. Clark.
End-to-end Arguments in System Design.
In *ACM Transactions on Computer Systems*, 2, pages 277–288, 1984.
Cited on page [3](#).
- [112] Vincent Simonet.
The Flow Caml System: Documentation and User’s Manual.
Technical Report 0282, Institut National de Recherche en Informatique
et en Automatique (INRIA), 2003.
Cited on page [33](#).
- [113] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal.
Reasoning About a Machine with Local Capabilities.
In *Programming Languages and Systems*, pages 475–501.
Cited on page [70](#).
- [114] Geoffrey Smith and Dennis Volpano.
Secure Information Flow in a Multi-threaded Imperative Language.
In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364.
Cited on pages [9](#), [23](#), [99](#) and [122](#).
- [115] 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*, pages 95–106.
Cited on pages [19](#), [28](#), [33](#), [68](#) and [102](#).
- [116] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazieres.
Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems.
In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, pages 201–214.
Cited on pages [27](#), [70](#), [99](#) and [107](#).
- [117] Deian Stefan, David Mazieres, John C. Mitchell, and Alejandro Russo.
Flexible Dynamic Information Flow Control in the Presence of Exceptions.
In *Journal of Functional Programming*, 27, pages e5, 2017.
Cited on pages [107](#), [108](#), [110](#), [111](#), [119](#) and [138](#).

- [118] Christopher A. Stone and Robert Harper.
Extensional Equivalence and Singleton Types.
In *ACM Transactions on Computational Logic*, 7, pages 676–722, 2006.
Cited on page [78](#).
- [119] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic.
Managing policy updates in security-typed languages.
In *19th IEEE Computer Security Foundations Workshop*, pages 13–216.
Cited on page [15](#).
- [120] The Coq Development Team.
The Coq Proof Assistant, version 8.9.0, 2019.
Cited on page [26](#).
- [121] Gil Tene, Balaji Iyengar, and Michael Wolf.
C4: The Continuously Concurrent Compacting Collector.
In *Proceedings of the International Symposium on Memory Management*,
pages 79–88.
Cited on page [66](#).
- [122] Mads Tofte and Jean-Pierre Talpin.
Region-Based Memory Management.
In *Information and Computation*, 132, pages 109–176, 1997.
Cited on page [102](#).
- [123] Stelios Tsampas, Dominique Devriese, and Frank Piessens.
Temporal safety for stack allocated memory on capability machines.
In *Proceedings of the 32th IEEE Computer Security Foundations Symposium*.
Cited on page [75](#).
- [124] Stephen Tse and Steve Zdancewic.
Run-time Principals in Information-flow Type Systems.
In *ACM Transactions on Programming Languages and Systems*, 30, 2007.
Cited on pages [9](#), [72](#), [83](#), [85](#), [87](#) and [103](#).
- [125] David Ungar.
Generation Scavenging: A Non-disruptive High Performance Storage
Reclamation Algorithm.
In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering
Symposium on Practical Software Development Environments*, pages 157–
167.
Cited on page [36](#).

- [126] Myra Ellen Vaninwegen.
The Machine-assisted Proof of Programming Language Properties.
PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1996.
Cited on page 59.
- [127] M. Vassena, J. Breitner, and A. Russo.
Securing Concurrent Lazy Programs Against Information Leakage.
In *Proceedings of the IEEE 30th Computer Security Foundations Symposium (CSF)*.
Cited on pages 25, 70 and 102.
- [128] Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan.
From Fine- to Coarse-Grained Dynamic Information Flow Control and Back.
In *Proceedings of the ACM on Programming Languages*, 3, pages 1–31, 2019.
Cited on pages 19, 102, 138 and 141.
- [129] Marco Vassena, Gary Soeller, Peter Amidon, Matthew Chan, John Renner, and Deian Stefan.
Foundations for Parallel Information Flow Control Runtime Systems.
In *Principles of Security and Trust*, pages 1–28.
Cited on page 23.
- [130] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith.
A Sound Type System for Secure Flow Analysis.
In *Journal of Computer Security*, 4, pages 167–187, 1996.
Cited on pages 9, 17, 55 and 103.
- [131] Philip Wadler.
Monads for Functional Programming.
In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52.
Cited on page 110.
- [132] Philip L. Wadler.
Analysis of an Algorithm for Real Time Garbage Collection.
In *Communications of the ACM*, 19, pages 491–500, 1976.
Cited on page 66.
- [133] Glynn Winskel.
The Formal Semantics of Programming Languages: An Introduction.
MIT Press, Cambridge, MA, USA, 1993.
Cited on page 43.

- [134] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe.
The CHERI Capability Model: Revisiting RISC in an Age of Risk.
In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, pages 457–468.
Cited on page [75](#).
- [135] A.K. Wright and M. Felleisen.
A Syntactic Approach to Type Soundness.
In *Information and Computation*, 115, pages 38–94, 1994.
Cited on page [60](#).
- [136] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers.
Using Information Flow to Design an ISA that Controls Timing Channels.
In *Proceedings of the 32nd IEEE Computer Security Foundations Symposium*.
Cited on page [23](#).
- [137] Steve Zdancewic and Andrew C. Myers.
Robust Declassification.
In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, pages 5–23.
Cited on pages [11](#), [16](#) and [129](#).
- [138] Steve Zdancewic and Andrew C. Myers.
Observational Determinism for Concurrent Program Security.
In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 29–43.
Cited on pages [5](#), [20](#), [23](#), [32](#), [99](#) and [102](#).
- [139] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers.
Secure Program Partitioning.
In *ACM Transactions on Computer Systems*, 20, pages 283–328, 2002.
Cited on page [15](#).
- [140] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazieres.
Making Information Flow Explicit in HiStar.
In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19.
Cited on pages [28](#), [107](#) and [138](#).

- [141] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres.
Securing Distributed Systems with Information Flow Control.
In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308.
Cited on page [107](#).
- [142] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers.
Predictive Mitigation of Timing Channels in Interactive Systems.
In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 563–574.
Cited on pages [43](#) and [77](#).
- [143] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers.
Language-based Control and Mitigation of Timing Channels.
In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–110.
Cited on pages [19](#), [21](#), [22](#), [35](#), [43](#), [61](#), [67](#) and [77](#).
- [144] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers.
A Hardware Design Language for Timing-Sensitive Information-Flow Security.
In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–516.
Cited on pages [22](#), [83](#) and [103](#).
- [145] Lantian Zheng and Andrew C. Myers.
Dynamic Security Labels and Static Information Flow Control.
In *International Journal of Information Security*, 6, pages 67–84, 2007.
Cited on pages [9](#), [19](#), [83](#) and [103](#).